Mobius

Information Society Technologies

Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable 2.6

Preliminary Report on Advanced Resource Policies

Due date of deliverable: 2008-09-01 (T0+36)

Actual submission date: 2008-10-15

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **UEDIN**

Submitted version

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary

This is Deliverable 2.6 of MOBIUS, an integrated project (FP6-015905) in the European Community Sixth Framework Programme. Full information about MOBIUS is available online at the project website `http://mobius.inria.fr`.

This deliverable reports on initial progress on type systems and static analyses for advanced resource policies and analysis (Task 2.4). The deliverable consists of copies of 13 publications that have appeared elsewhere and includes contributions from all MOBIUS partners involved in Task 2.4, namely INRIA, LMU, UEDIN, and UPM. The results reported here build on previous work in Task 2.3.

- Chapter 1 is a brief introduction outlining the space of type systems and program analyses covered by Task 2.4.
- Chapter 2 presents INRIA's implementation of an inter-procedural relational analysis for Java bytecode that can compute invariants used in the generation of resource certificates.
- Chapter 3 reports on LMU's prototype implementation of a type system for amortised heap-space analysis, and LMU's development of a generic resource extension to the MOBIUS Base Logic. The extended logic serves as a target for translating type derivations into base logic proofs, as demonstrated by interpretations of type systems for constant heap space and for block booking resources.
- Chapter 4 presents UEDIN's approach to block booking: explicit accounting using resource managers. Resource safety is then enforced either dynamically by run-time monitoring, or statically by a type system.
- Chapter 5 summarises UPM's efforts in order to improve the generic framework for cost analysis of Java bytecode (cf. Deliverable 2.3) in several directions: Estimating the usage of specific resources, solving recurrence equations generated by automatic cost analysis, developing cost analysis generic in the concept of resource, and developing new analyses of heap structures in Java bytecode programs.
- Appendix A[1] collates the 13 publications that constitute this deliverable.

This report reflects only the views of the authors and the European Community is not liable for any use that may be made of the information contained therein.

[1] **Save paper:** Avoid printing the appendix (200+ pages).

Version Control History

Revisions

Version	Date	Purpose	Revision code
Draft D2.6	2008-09-10	First version, circulated for partner review	6870
Revised D2.6	2008-09-30	Revised version, prepared for lead site.	6951
Final D2.6	2008-10-15	Submitted to European Project Office	7010

Note: "Revision code" refers to the version control number assigned by the project online document repository. This uniquely identifies all historical versions of MOBIUS documents through drafting and editing.

Authors

Site	Contributed to Chapter
INRIA	2
LMU	3
UEDIN	1,4, editor
UPM	5

Contents

Chapter 1

Introduction

This deliverable reports on initial progress on type systems and static analyses for advanced resource policies and analysis (Task 2.4). The deliverable consists of copies of 13 publications (collated in the appendix for convenience) that have appeared elsewhere and includes contributions from all MOBIUS partners involved in Task 2.4, namely INRIA, LMU, UEDIN, and UPM, cf. following chapters for details.

On mobile phones certain resources, like sending text messages, must be controlled tightly. As an example scenario take *bulk messaging* where the user wants to send a text message to a number of recipients. Because of the cost of sending text messages, the user must authorise each message explicitly. Java MIDP 2.0 implements this requirement by insisting on each message being authorised individually just before it is sent, thus bombarding the user with confirmation screens. A better way to fulfil this requirement would be collective authorisation of all messages in one go, also known as *block booking*. However, this requires tracking the flow of authorised resources from the points of authorisation to the points of use, to ensure that no more messages are sent than authorised.

The type systems and program analyses presented in this deliverable guarantee adherence to resource-related properties of mobile code, like soundness of block booking, for instance. Type systems are an enabling technology for the MOBIUS Proof-Carrying Code (PCC) architecture because they are intuitive, automatic and scalable. Hence improvements of program coverage and language coverage as well as of flexibility and scalability, as they are presented in this deliverable, are important steps to build the MOBIUS PCC-architecture.

The following chapters describe various type systems and program analyses for controlling resources, notably execution time, heap space, and access to external resources. The results reported here build on previous work (Task 2.3) in various ways.

- Chapter 2 presents INRIA's implementation of an inter-procedural relational analysis for Java bytecode that can compute invariants used in the generation of resource certificates (e. g., for applications using block booking).
- Chapter 3 reports on LMU's prototype implementation of a type system for amortised heap-space analysis, and LMU's development of a generic resource extension to the MOBIUS Base Logic. The extended logic serves as a target for translating type derivations into base logic proofs, as demonstrated by interpretations of type systems for constant heap space and for block booking resources.
- Chapter 4 presents UEDIN's approach to block booking: explicit accounting using resource managers. Resource safety is then enforced either dynamically by run-time monitoring, or statically by a type system.
- Chapter 5 summarises UPM's efforts in order to improve the generic framework for cost analysis of Java bytecode (cf. Deliverable 2.3) in several directions: Estimating the usage of specific resources, solving recurrence equations generated by automatic cost analysis, developing cost analysis generic in the concept of resource, and developing new analyses of heap structures in Java bytecode programs.

Chapter 2

Polyhedral Analysis of Java Bytecode for Certificate Generation

In this deliverable, INRIA reports on the following publication.

[1] F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Research Report 6333, IRISA, September 2007.

The publication documents the implementation of a relational (polyhedral) analysis for Java bytecode that can compute invariants used in the generation of resource certificates.

INRIA [1] has developed a relational (polyhedral-based) static analysis for Java bytecode that is capable of automatically computing invariants of the relations between program variables. By introducing variables that represent the amount of resources available (cf. the resource model for Java MIDP resources developed in the project), such an analysis can be used to prove that a program uses its resources correctly in the sense that it does not use more resources than it has been granted. The following code fragment illustrates this with an instance of the block booking idiom. In this example, the computed invariant relation between the length of the array addr, the (global) variable sms holding the number of (permissions to send) SMSs granted to the application and the iterator i combine to prove that the number of available SMSs will never fall below zero.

```
public static void main(String [] addr) {
    int N = 0;                                  // Number of winners
    String [] aux = new String[addr.length];    // Array with winners at the start, losers at the end

    for (int i = 0; i < addr.length; i++)       // Fill the array of winners and losers
      if ( competitor i is a winner )
          { aux[N] = addr[i]; N++; }
      else
          { aux[addr.length−1−(i−N)] = addr[i]; }

    grant(addr.length+N);                       // Allocate 2 messages to send to winners, 1 for losers
                                                // Side effect: sms=addr.length+N;
    // Send the messages themselves
    //@ maintaining addr.length ≤ sms + i ≤ addr.length + N ∧ i ≤ addr.length ≤ sms + 2i − N
    for (int i=0; i < addr.length; i++)
        if (i<N)
            sendSMS(2,aux[i]);                  // Side effect: sms=sms−2;
        else
            sendSMS(1,aux[i]);                  // Side effect: sms=sms−1;
}
```

To infer such invariants, the analysis takes as input Java bytecode and infers for each bytecode instruction a relation between global variables, the parameters to a method, local variables and the numeric values stored on the stack. This relation is represented by elements of the abstract state underlying the analysis. Rather than fixing a particular abstract domain from the outset, we have specified the bytecode analysis with respect to an abstract numeric relational interface made up of a few central operations such as upper bounds, variable renaming and projection. These operations can then be instantiated with standard relational abstract domains such as polyhedra and octagons.

The bytecode analysis is defined by specifying for each bytecode an abstract transfer function which maps abstract states to abstract states. There are several issues that had to be addressed in order to extend basic polyhedral analyses of imperative programs to an analysis of the stack-based, object-oriented, procedural Java bytecode:

- The operand stack of Java bytecode can be costly to model fully in a relational analysis. We enrich the abstract domain to include symbolic expressions that represent the content of a stack element in terms of program variables. This de-compilation of the stack also allows to extend the analysis to handle more specific entities such as eg. array length.

- The analysis currently employs a simple model of the heap of objects in which objects are abstracted to their type and their fields are ignored. The only exception to this are arrays where the size (but still not the content) is represented.

- The analysis is inter-procedural. It handles recursion and computes for each method a set of pre- and post-conditions that represent invariants at the start and end of method execution, respectively.

The correctness of the analysis specification has been proved using a standard approach based on abstract interpretation using Galois connections to formalise the link between elements of the abstract domain and set of program states. A distinctive feature of the proof is that it has been conducted inside the Coq proof assistant, providing a machine-checkable proof of semantic correctness.

An actual solution to the flow equation defining the analysis of a bytecode program can be computed by combining the analyser with an implementation of a relational domain such as Polka or PPL. We have developed an implementation that uses the Polka polyhedral library and which gives satisfactory analysis precision and time for small to medium-sized benchmarks.

The invariants computed by the relational analysis can serve as the basis for generating compact certificates for use in a proof carrying code scenario based on static analysis. The invariants often provide more information than strictly necessary for proving that a program respects a given security policy. There is hence room for pruning such invariants in order to obtain compressed certificates of code correctness.

The analysis specification can be re-used to build a *certificate checker* for Java bytecode that can be used in a PCC scenario where bytecode comes equipped with resource certificates. Such a checker has been developed and installed on a mobile device as part of Work Package 4.

Chapter 3

(Amortised) Heap Space Consumption, and Numeric Correspondence Assertions

In this deliverable, LMU reports on the following publications.

[1] L. Beringer, M. Hofmann, and M. Pavlova. Certification using the Mobius base logic. In *Formal Methods for Components and Objects: Revised Lectures from the 6th International Symposium FMCO 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

[2] M. Hofmann and D. Rodriguez. Implementing a type system for amortised heap-space analysis. 2008.

Implementing a type system for amortised heap-space analysis (paper [2]) The prediction of resource consumption in programs has gained interest in the last years. It is important for a number of areas, notably embedded systems and safety critical systems. Different approaches to achieve bounded resource consumption have been analysed. One of them, based on an amortised complexity analysis, has been studied by Hofmann and Jost for a Java-like language called RAJA.

The present paper describes an automated type-checking algorithm for a modified version of RAJA which we call RAJA+. Moreover we prove that the type checking algorithm we present is sound and complete with respect to the declarative type system of RAJA+. This proves the decidability of the system.

On the other side, we have implemented the type checking algorithm as well as a parser and an interpreter for RAJA+ programs in Ocaml. This prototype implementation supports further investigation in this area.

Certification using the Mobius Base Logic (paper [1]) This paper describes a core component of MOBIUS' Trusted Code Base, the MOBIUS base logic. This program logic facilitates the transmission of certificates that are generated using logic- and type-based techniques and is formally justified w.r.t. the Bicolano operational model of the JVM. The paper motivates major design decisions, presents core proof rules, describes an extension for verifying intensional code properties, and considers applications concerning security policies for resource consumption and resource access.

Of particular relevance for the present deliverable are Section 4 and 5 of the named publication. In Section 4, a type system is presented that guarantees constant bound on heap space. In contrast to the type system discussed above, the system is phrased on bytecode, and has been formally justified with respect to the Bicolano operational semantics. Section 5 presents a solution to the block booking challenge, namely a bytecode-level type system for numeric correspondence assertions, where the authorisation request operation is parametric in a variable, such that the number of authorisations that are requested may depend dynamically on other data.

Chapter 4

Safety Guarantees from Explicit Resource Management

In this deliverable, UEDIN reports on the following publications.

[1] D. Aspinall, P. Maier, and I. Stark. Monitoring external resources in Java MIDP. *Electronic Notes in Theoretical Computer Science*, 197(1):17–30, 2008.

[2] D. Aspinall, P. Maier, and I. Stark. Safety guarantees from explicit resource management. In *Formal Methods for Components and Objects: Revised Lectures from the 6th International Symposium FMCO 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

The publications focus on the compile- and run-time guarantees, e. g., static and dynamic soundness of block booking, cf. Chapter 1, of our approach to the explicit (i. e., manifest in code) management of resources such as text messages.

In [1], we present a Java library for MIDP devices which tracks and controls at run-time the use of potentially costly resources, such as sending text messages. The library supports block booking of resources while maintaining the security guarantee that attempted resource abuse is trapped. Tracking of resources is done by *resource managers*, special objects encapsulating multisets of authorised resources. This allows for fine-grained tracking; for instance, we are able to track not just the total number of text messages sent by an application, but the number of messages sent to each individual recipient. To reduce the run-time overhead of tracking multisets, resource managers can easily be erased without altering an application's behaviour if that application is *dynamically resource safe*, i. e., cannot be caught abusing resources. Additionally, the library introduces a flexible notion of *policy* for deciding which resources to grant.

The resource manager library for trapping attempts to abuse resources at run-time can be viewed as a language-based mechanism enforcing resource safety at run-time. In [2], we complement this with a type system for proving that a given program (in a functional language with resource managers) does not attempt to abuse resources. The type system derives logical constraints (in a generic logical constraint language) approximating the effects of evaluating program expressions. Typability of functions in the effect type system induces a notion of *static resource safety*, and in particular implies dynamic resource safety. As a consequence, resource managers can always be erased from well typed programs.

The decidability of type checking in the above type system rests on the decidability of satisfiability in the underlying constraint language. As examples like the bulk messaging application reveal, meaningful types require a constraint language able to express properties of multisets (to model resource managers) and container data structures (like vectors and dictionaries). We are currently investigating decidable fragments of such expressive constraint languages.

Chapter 5

Cost Analysis

In this deliverable, UPM reports on the following publications.

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java bytecode. In *International Conference on Formal Methods for Open Object-based Distributed Systems*, Lecture Notes in Computer Science 5051, pages 2–18. Springer-Verlag, 2008.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Static Analysis Symposium*, Lecture Notes in Computer Science 5079, pages 221–237. Springer-Verlag, 2008.

[3] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap space analysis of Java bytecode. In *International Symposium on Memory Management*, pages 105–116. ACM Press, 2007.

[4] S. Genaim and F. Spoto. Constancy analysis. In M. Huisman, editor, *10th Workshop on Formal Techniques for Java-like Programs*, July 2008.

[5] M. Marron, M. V. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *Compiler Construction*, Lecture Notes in Computer Science, volume 4959, pages 245–259. Springer-Verlag, 2008.

[6] M. Méndez-Lojo and M. V. Hermenegildo. Precise set sharing analysis for Java-style programs. In *Verification, Model Checking and Abstract Interpretation*, Lecture Notes in Computer Science, volume 4905, pages 172–187. Springer-Verlag, 2008.

[7] E. Mera, P. López-García, M. Carro, and M. Hermenegildo. Towards execution time estimation in abstract machine-based languages. In *Principle and Practice of Declarative Programming*, pages 174–184. ACM Press, 2008.

[8] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable resource usage analysis for Java bytecode. Technical Report UNM TR-CS-2008-02 - CLIP1/2008.0, University of New Mexico, January 2008.

The above publications summarise the effort done by UPM in order to improve the generic framework for cost analysis of Java bytecode, that have been reported in the previous year, in several directions: (1) applying it to estimate the usage of specific resources; (2) developing a practical solver for recurrence equations generated by automatic cost analysis; (3) developing a cost analysis that is generic in the concept of resource at the user level, and (4) developing new analyses for inferring properties of heap structures in Java bytecode programs, which are used to improve the precision of cost analysis.

As regards (1), we have developed a heap consumption analysis for Java bytecode [3], which is based on the generic framework for cost analysis that has been reported last year (Deliverable D2.3). The analysis generates *heap space cost relations* which define at compile-time the heap consumption of a program as a function of its input data size. These relations can be used to obtain upper bounds on the heap space

allocated during the execution of the different methods. In addition, we have described a possible refinement of the analysis, by relying on *escape analysis*, in order to take into account the heap space that can be safely deallocated by the garbage collector upon exit from a corresponding method. We have reported a prototype implementation and its application for the inference of heap usage proportional to the data size including constant, polynomial and exponential heap consumption. We have also presented a framework for estimating upper and lower bounds on the execution times of logic programs, which works at the level of the corresponding bytecode-based abstract machine [7]. This framework is based on a program independent profiling stage which calculates constants or functions bounding the execution time of each abstract machine instruction. In addition, we have continued the study of the connection between termination and cost analysis [1], and developed a framework for termination analysis for Java bytecode programs which makes extensive use of the components of the cost analysis framework. Given a bytecode program, our approach produces a *constraint logic* program, whose termination entails termination of the bytecode program. This allows applying the large body of work in termination of constraint logic programs to termination of Java bytecode. A prototype analyser has been described and initial experiments were reported in [1].

As regards (2), in a more general direction, in [2], we have studied the features of recurrence relations generated by automatic cost analysis and explained why existing computer algebra systems are not appropriate for automatically obtaining closed form solutions nor upper bounds of them. Then, we have presented and implemented a practical framework for the fully automatic generation of reasonably accurate upper bounds of recurrence relations originating from cost analysis of a wide range of programs. The approach is based on programming language techniques such as the inference of *ranking functions* and *loop invariants* and on *partial evaluation*.

As regards (3), we have developed, implemented, and benchmarked a cost analysis for Java bytecode [8] that is generic in the concept of resource at the user level. An assertion language allows users to define application-dependent notions of resources such as bytes sent or received by an application, number of files left open, number of SMSs sent or received, number of accesses to a database, money spent, energy consumption, etc. This language is also used to define the resource consumption of some relevant elementary or library operations. From these definitions our analysis derives in an automated way functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. We have also implemented and studied this analysis experimentally on an interesting set of user-defined resources.

As regards (4), we have developed several analyses for inferring properties of heap structures in the context of Java bytecode. This includes an analysis for inferring possible sharing between heap structures [6], an analysis for approximating the shape of heap structures [5], and an analysis for inferring immutability of heap structures [4]. These analyses are extensively used in the cost analysis framework developed at UPM.

Appendix A

Copies of Publications

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Result certification for relational program analysis

Frédéric Besson — Thomas Jensen — David Pichardie — Tiphaine Turpin

N° 6333

October 2007

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--6333--FR+ENG

Result certification for relational program analysis

Frédéric Besson*, Thomas Jensen† , David Pichardie*, Tiphaine Turpin‡

Thème SYM — Systèmes symboliques
Projet Lande

Rapport de recherche n° 6333 — October 2007 — 29 pages

Abstract: We define a generic relational program analysis for an imperative, stack-oriented byte code language with procedures, arrays and global variables and instantiate it with an abstract domain of polyhedra. The analysis has automatic inference of loop invariants and method pre-/post-conditions, and efficient checking of analysis results by a simple checker. Invariants, which can be large, can be specialized for proving a safety policy using an automatic pruning technique which reduces their size. The result of the analysis can be checked efficiently by annotating the program with parts of the invariant together with certificates of polyhedral inclusions, which allow to avoid certain complex polyhedral computation such as the convex hull of two polyhedra. Small, easily checkable inclusion certificates are obtained using Farkas lemma for proving the absence of solutions to systems of linear inequalities. The resulting checker is sufficiently simple to be entirely certified within the Coq proof assistant.

Key-words: Static analysis, abstract interpretation, bytecode Java, Coq

* INRIA Rennes - Bretagne Atlantique/IRISA
† CNRS/IRISA
‡ Université Rennes I/IRISA

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 —Télécopie : +33 2 99 84 71 71

Certification de résultat pour l'analyse de programme relationnelle

Résumé : Nous proposons une analyse générique de programme relationnelle pour un langage de bytecode impératif avec pile d'opérande, procédures, tableaux et variables globales. Cette analyse est instanciée avec un domaine abstrait de polyèdres. Elle propose une inférence automatique d'invariants de boucle et de préconditions/postconditions de procédures, ainsi qu'une vérification efficace du résultat de l'analyse par un vérificateur simple. Les invariants, qui peuvent être grands, peuvent être spécialisés pour prouver une propriété de sûreté en utilisant une technique automatique de compression de taille de certificat. Le résultat de l'analyse peut être vérifié efficacement en annotant le programme avec une partie des invariants et quelques certificats d'inclusion de polyèdre, qui permettent d'éviter certaines calculs polyédriques complexes comme le calcul de l'enveloppe convexe de deux polyèdres. Nous obtenons des certificats d'inclusion petits et facilement vérifiables grâce au lemme de Farkas pour prouver l'absence de solution dans un système d'inégalités linéaires. Le vérificateur ainsi obtenu est suffisamment simple pour être entièrement certifié avec l'assistant à la preuve Coq.

Mots-clés : Analyse statique, interprétation abstraite, bytecode Java, Coq

1 Introduction

Logic-based, static program verification, be it in form of abstract interpretation, symbolic model checking or interactive proving of programs, is used in a number of ways to improve the confidence in safety-critical systems and for protecting host machines from malicious code, as *e.g.*, done by the Java byte code verifier. As applications and the program logics grow in complexity, an automated technique for verifying program invariants based on a program logics should ideally meet all of the following three requirements:

- *Automatic Inference*: the complexity of both programs and the underlying logic can quickly make it burdensome to conduct program proofs manually. Automatic inference of program properties is necessary to obtain a technique that scales.
- *Result certification*: when inference is available, it often relies on advanced deductive methods for inferring an invariant whose size and complexity make it difficult to ascertain its validity manually. Efficient checking of the result of the inference or of any proposed invariant in general becomes important.
- *Small Trusted Computing Base (TCB)*: the result checker becomes the cornerstone of the reliability of the verification framework. In order to reduce the part of the code base that needs to be trusted without proof, the checker should be kept sufficiently simple and small in order to be able to verify the checking algorithmics mechanically.

Program verification based on general Hoare-style program logics may follow the Verification Condition Generator (VCGen) approach of *e.g.*, Extended Static Checking by Flanagan, Leino *et al.* [14] or use expressive type systems such as the dependent type systems of Xi and Pfenning [27] for proving properties of programs. The approaches based on VCGens are generally complete for partial correctness and will produce a set of verification conditions which, when satisfied, will allow to conclude that a given program property holds in the logic. Verification conditions often fall into fragments of logic that require them to be proved by dedicated decision procedures or theorem provers. VCGens and the type-based approaches are primarily concerned with invariant checking and discard part of the inference problem by relying on loop invariants and pre-post-condition of methods to be provided by the programmer. In terms of small TCB, the VCGens remain complex software which are hard to prove correct *in extenso*. The machine-checked formalizations e.g., by Nipkow, Wildmoser *et al.* [25, 26] show that this is indeed possible to certify an entire VCGen inside a proof assistant but also that this remains a major software certification challenge.

Another strand of program verification is based on abstract interpretation. Abstract interpretation is an automatic technique for inferring program properties in the form of fixpoints of monotone data flow functions. As a theory of proving programs it has strong semantic foundations. At the same time it should be noted that the algorithmics of the domains underlying the more advanced analyses such as polyhedral analysis (initially described by Cousot and Halbwachs [13]) is highly non-trivial. Checking an invariant is in theory simple as it only requires one more iteration to check that a property is indeed a

fixpoint but, as said, this computation does in certain cases rely on non-trivial algorithmics that forms part of what must be trusted. In previous work [9, 21], some of the authors formalised the theory of abstract interpretation inside the proof assistant Coq and extracted Caml implementations of a variety of program analyses. This Certified Abstract Interpretation approach represents a systematic way of reducing the TCB of static analyzers and fulfills the three requirements listed above. However, a fully mechanised correctness proofs of more advanced program analysers such as an optimised, polyhedral-based analysis would require an enormous effort in terms of program certification.

The purpose of this paper is to demonstrate that by focusing on certifying the *result* of the analysis rather than the analysis itself, it is possible to develop a verification framework for advanced program properties that satisfies all of the three desired properties and, at the same time, requires a significantly smaller effort in order to be proved correct. This idea was previously used by Wildmoser *et al* [24] who use the result of an untrusted interval analysis in a VCGen for byte code and by Leroy [17] in his certification of a compiler back-end where he, rather than certifying the complex graph-coloring algorithms for register allocation, proves the correctness of a checker that verifies a given coloring returned by an untrusted graph-coloring algorithms. Here, we generalise this idea by developing a relational analysis framework together with a certified checker. The basic observation is that an abstract interpretation can be decomposed into an abstract domain of properties, a generic program logic for reasoning about these properties and a fixpoint engine for solving recursive equations over the abstract domains. The inference does not need to use certified abstract domain operations and fixpoint engines, and the checking of invariants does not need to use a fixpoint engine at all. We take advantage of this to design a checker that re-uses the program logic but replaces the more complex domain operations with simpler ones, at the expense of providing some extra information in the certificate accompanying a program.

2 Overview

In the first part of this paper, we will develop a fully relational, interprocedural analyser which automatically infers an invariant for each control point in the program, a pre-condition that must hold at the point of calling a procedure and a post-condition that is guaranteed to hold when the procedure returns. Relational analyses are useful for finding loop invariants needed for proving program safety, e.g. when verifying the resource usage of programs or verifying safety properties related to safe memory access such as checking that all array accesses are within bounds. We will take Safe Array Access as an example safety policy and illustrate our approach with the Binary Search example given in Fig. 1, showing how the analysis will prove that the instruction that accesses the array `vec` with index `mid` will not index out of bounds.

We have annotated the code of Binary Search with the invariants that have been inferred automatically. Invariants refer to values of local and global variables and can also refer to the length of an array. For example, the invariant (I_3) asserts among other properties that when entering the while loop, the relation $0 \leq \texttt{low} < \texttt{high} < |\texttt{vec}|$ is satisfied. Similarly, the post-condition ensures that the result is a valid index into the array being searched, or -1, indicating that

```
  //     PRE: 0 ≤ |vec0|
static int bsearch(int key, int[] vec) {
  // (I1) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ |vec0|
   int low = 0, high = vec.length - 1;
  // (I2) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec0|
   while (0 < high-low) {
  // (I3) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low < high < |vec0|
      int mid = (low + high) / 2;
  // (I4) key0 = key ∧ |vec0| = |vec| ∧ 0 ≤ low < high < |vec0| ∧ low + high − 1 ≤ 2 · mid ≤
low + high
      if (key == vec[mid]) return mid;
      else if (key < vec[mid]) high = mid - 1;
      else low = mid + 1;
   // (I5) key0 = key ∧ |vec0| = |vec| ∧ −2 + 3 · low ≤ 2 · high + mid ∧ −1 + 2 · low ≤
high + 2 · mid ∧ −1 + low ≤ mid ≤ 1 + high ∧ high ≤ low + mid ∧ 1 + high ≤ 2 · low + mid ∧
1 + low + mid ≤ |vec0| + high ∧ 2 ≤ |vec0| ∧ 2 + high + mid ≤ |vec0| + low
   }
  // (I6) key0 = key ∧ |vec0| = |vec| ∧ low − 1 ≤ high ≤ low ∧ 0 ≤ low ∧ high < |vec0|
   return -1;
} //    POST: −1 ≤ res < |vec0|
```

Figure 1: Binary search

the element was not found. In addition, the analysis introduces a 0-indexed variable (such as *e.g.* $\texttt{key}_0$ in the example) for each parameter (and also for the global variables, of which there are none in the example) in order to refer to its value when entering the procedure. The effect of this is that the invariant on exit of the program defines a relation between the input and the output of the procedure, thus yielding a *summary relation* for the procedure.

2.1 Compressing invariants

Abstract interpretations may give you more information than you need for proving a particular property. In the case of the Binary Search example, if we are only interested in proving the validity of array accesses, there are a number of relations between variables in the invariants that can be forgotten. Reducing the number of constraints and the number of variables under consideration can lead to a significant gain in execution time when it comes to checking a proposed invariant. For example, pruning the invariants in Fig. 1 with respect to this property yields the simpler invariant shown in Fig. 2:

Notice that the inferred loop invariant I_3' is close to what a specifying programmer of Binary Search might have come up with, but here produced automatically. We explain pruning of procedures in Section 7.

2.2 Analysing a stack-based language

Polyhedral analysis of While languages is well understood but we want our framework to be able to analyse byte code programs and not only source code. We could in theory avoid the problem by transforming the program into three-address code and treat each stack location as a local variable but this transformation is expensive from an algorithmic point of view, as it increases the number of times that the relation has to be updated. Instead, we achieve the effect of this transformation by defining an analysis for stack-oriented byte code

```
  //     PRE: True
static int bsearch(int key, int[] vec) {
  // (I'_1) |vec_0| = |vec| ∧ 0 ≤ |vec_0|
   int low = 0, high = vec.length - 1;
  // (I'_2) |vec_0| = |vec| ∧ 0 ≤ low ≤ high + 1 ≤ |vec_0|
   while (0 < high-low) {
  // (I'_3) |vec_0| = |vec| ∧ 0 ≤ low < high < |vec_0|
      int mid = (low + high) / 2;
  // (I'_4) |vec| − |vec_0| = 0 ∧ low ≥ 0 ∧ mid − low ≥ 0∧
  //        2 · high − 2 · mid − 1 ≥ 0 ∧ |vec_0| − high − 1 ≥ 0
      if (key == vec[mid]) return mid;
      else if (key < vec[mid]) high = mid - 1;
      else low = mid + 1;
  // (I'_5) |vec_0| = |vec| ∧ −1 + low ≤ high ∧ 0 ≤ low ∧ 5 + 2 · high ≤ 2 · |vec|
   }
  // (I'_6) 0 ≤ |vec_0|
   return -1;
} //    POST: −1 ≤ res < |vec_0|
```

Figure 2: Binary search after invariant pruning

that combines relational abstract interpretation with symbolic execution, following an idea previously used for analysing Java byte code by Xi and Xia [28] and Wildmoser *et al* [24]. This technique abstracts the environment of local variables by a relation (e.g., a polyhedron) and replace the operand stack with a stack of symbolic expressions used to "decompile" the operations on the operand stack. For example, the comparison of variables `low` and `high` will be compiled to the byte codes below, which are analysed in a state consisting of the relation I_2 as defined in Fig. 1 and an abstract stack that evolves as values are pushed onto the stack.

		[]	I_2
7 :	ipush 0	0	I_2
8 :	iload high	`high` :: 0	I_2
9 :	iload low	`low` :: `high` :: 0	I_2
10 :	isub	(`high`−`low`) :: 0	I_2
11 :	if_icmpge 56	[]	I_3

Before the comparison in instruction 11, the stack top contains the expression `high`−`low`, reflecting that in the real execution the stack top at this point will contain the value of this expression. When we learn from the test that the expression `high`>`low` evaluates to true in the state immediately following the comparison (and only then), we update the relation accordingly to obtain invariant I_3. Similarly, we have to update the relation when assigning a new value to a variable. For example, the instruction that assigns (`high`+`low`)/2 to `mid` is compiled and analysed as shown below. Again, the relation I_3 is only updated when the assignment to `mid` is done, to yield relation I_4.

		[]	I_3
14 :	iload low	low	I_3
15 :	iload high	high :: low	I_3
16 :	iadd	(high+low)	I_3
17 :	ipush 2	2 :: (high+low)	I_3
18 :	idiv	((high+low)/2)	I_3
19 :	istore mid	[]	I_4

More generally, with the abstract stack of expressions, only the comparisons and assignment to variables require updating the relation. In a polyhedron-based analysis this is a substantial saving.

2.3 Result checking with certificates

Checking an invariant obtained by computing a post-fixpoint of an abstract interpretation is in theory simple as it only requires one more iteration to check that it is indeed a post-fixpoint. In addition, only invariants at certain program points such as loop headers are required for re-building an entire invariant in one iteration. Lightweight Bytecode Verification by Rose [22] and the more general Abstraction-Carrying Code by Albert, Puebla and Hermenegildo [1] exploit this to construct efficient checkers for invariant-based program certificates. For the code in Fig. 2, only I'_2 is required.

The inference of invariants using our relational analysis uses an iterative fixpoint solver over an abstract domain of polyhedra and is in principle amenable to the same technique. However, despite efficient implementations of basic polyhedral operations, the algorithmic complexity of operations such a computing the least upper bound (*i.e.* the convex hull) of two polyhedra remains high, and certifying them in a proof assistant would be a major undertaking.

Instead, we propose an enriched certificate format which has the virtue of being simpler to check, at the cost of sending more information than in basic fixpoint reconstruction. We exploit that, for the checker, the only important property of the convex hull operators is that it produces an upper bound of two polyhedra and therefore can be replaced by inclusion checks with respect to an upper bound that is proposed by the certificates. Upper bounds are computed at join points so in Fig. 2 we would also supply I'_5.

Safety checks also reduces to inclusions of polyhedra as verifying the array access $\texttt{vec}[\texttt{mid}]$ amounts to ensuring that I'_4 implies $0 \leq \texttt{mid} < |\texttt{vec}|$. By simple propositional reasoning, this reduces to proving that the linear systems of constraints $-\texttt{mid} - 1 \geq 0 \wedge I'_4$ and $\texttt{mid} - |\texttt{vec}| \geq 0 \wedge I'_4$ have no solution. Due to a result by Farkas, such problems can be checked efficiently using certificates by a simple matrix computation. The key insight is that unsolvability follows from the existence of a *positive* combination of the constraints which yield a strict negative constant. This would lead to a contradiction because the sum and product of positive quantities cannot be strictly negative. The certificate is therefore a vector which records the coefficients of the positive combination. For example, the certificate $[\mathbf{2}; \mathbf{2}; \mathbf{0}; \mathbf{0}; \mathbf{1}; \mathbf{2}]$ proves that the constraints $\texttt{mid} - |\texttt{vec}| \geq 0 \wedge I'_4$ are unsatisfiable, as the expression

$$\mathbf{2} \cdot (\texttt{mid} - |\texttt{vec}|) + \mathbf{2} \cdot (|\texttt{vec}| - |\texttt{vec}_0|) + \mathbf{0} \cdot \cdots + \mathbf{0} \cdot \cdots + \\ \mathbf{1} \cdot (2 \cdot \texttt{high} - 2 \cdot \texttt{mid} - 1) + \mathbf{2} \cdot (|\texttt{vec}_0| - \texttt{high} - 1)$$

evaluates to -2. We explain these certificates in detail in Section 8.

2.4 Certified certificate checkers

The result checking technique explained above already drastically reduces the TCB of the analysis result which only rely on the result checker. To further reduce the TCB, we have machine-checked the result checker of our analysis in the Coq proof assistant. The main components of the formalisation are

1. a predicate `Safe:program→Prop` which models the safe programs with respects to the the semantics described in Section 4,
2. a function `checker:program→certificate→bool`, which checks the safety of a program using a certificate containing a (partial) result of an analysis and some inclusion certificates,
3. a machine checked proof establishing the correctness of the checker:

```
Theorem checker_correct :
  ∀ p cert, checker p cert = true → Safe p.
```

The Trusted Computed Base is hence reduced to the Coq type checker and the formal definition of program safety.

Once the certified result checker is verified (by the Coq type checker) and installed by the code consumer, two scenarios can be envisaged to verify the safety of programs sent by producers. In the first one, the consumer may use an efficient Ocaml version of the checker, extracted from the Coq version thanks to the Coq extraction mechanism. The other alternative is related to *proof by reflection*. For each program `p` and certificate `cert` the consumer may build a foundational Coq proof of `Safe p`. To do so he only has to check in Coq the term `checker_correct p cert refl_eqtrue` where `refl_eqtrue` denotes a proof of `true=true`. It is the role of the Coq reduction engine to verify during type checking if `true=true` is equivalent to `checker p cert = true` by running the checker inside Coq. In this way we combine two desirable features which are often difficult to reconcile in state-of-the art Proof Carrying Code: foundational proofs and small certificates.

3 Notations

Let A and B be sets. If A and B are disjoint then $A + B$ is the disjoint sum of A and B. We write $A_\perp$ the set $A + \{\perp\}$. For $f \in A \to B_\perp$, $dom(f) = \{a \in A \mid f(x) \neq \perp\}$. Let $f \in A \to B$, $f[x \mapsto v]$ is the function identical to f everywhere except for x for which it returns v. The notation $[x_1 \mapsto v_1; \ldots; x_n \to v_n]$ stands for a function f of domain $\{x_1, \ldots, x_n\}$ such that $f(x_i) = v_i$. A^* is the set of lists of elements of A. We write $[]$ for the empty list and $a_0 :: \ldots :: a_{n-1}$ is a list l of length n ($|l| = n$) whose head (resp. tail) is a_0 (resp. a_{n-1}). $l[i]$ is the i-th element of l. We write a^i the list that is the repetition of a, i times. Let V, W be totally ordered sets. For $x \in V$, $\iota_V(x)$ is the index of x in set V and ι_V^{-1} is the inverse function. We abuse notations and identify $A^{|V|}$ with $V \to A$ *i.e.,* given a finite ordered set, $V = \{x_1, \ldots, x_n\}$ such that $x_1 < \ldots < x_n$, we identify the finite mapping $[x_1 \mapsto v_1, \ldots x_n \mapsto v_n]$ with the n-tuple $(v_1, \ldots, v_n)$.

We will write A^V to denote both $A^{|V|}$ and $V \to A$. Let $\rho \in A^V$ and $V' \subseteq V$, $\rho_{|V'} \in A^{V'}$ is the restriction of e over the variables of V' such that for all $x \in V'$, $\rho_{|V'}(x) = e(x)$. Given V and W disjoint set of variables, $\rho_1 \in A^V$ and $e_2 \in A^W$, we write $\rho_1 \oplus \rho_2 \in A^{V+W}$ for the finite mapping such that $(\rho_1 \oplus \rho_2)_{|V} = \rho_1$ and $(\rho_1 \oplus \rho_2)_{|W} = \rho_2$. Let W and W' ordered sets of same cardinality. If $\rho \in A^{V+W}$, then $\rho_{W \to W'} \in A^{V+W'}$ is obtained by renaming the variables of W to the variables in W'. Formally, we have $\rho_{W \to W'}(x) = \rho(x)$ if $x \in V$ and $\rho_{W \to W'}(x) = \rho(\iota_W^{-1}(\iota_{W'}(x)))$ if $x \in W'$. To make the distinction clear between syntactic expressions and values, syntactic expressions are bracketed ($\llcorner \cdot \lrcorner$). For example, we write $\llcorner 1 + e \lrcorner$ a syntactic expression built by applying the $+$ operator to the constant 1 and the syntactic expression e.

4 A byte code language and its semantics

We use a simple stack-based byte code language to illustrate our ideas. Features include integers, dynamically created (unidimensional) array of integers, static methods (procedures) and static fields (global variables).

Programs are lists of methods and a method consists of a name, a number of arguments and a list of instructions. In the following, f ranges over the set S of static field names, r ranges over the set $R = \{r_0, \ldots, r_{|R|}\}$ of local variables and id ranges over the set *MethId* of method names. Moreover, i and n range over $\mathbb{N}$ or $\mathbb{Z}$ depending on the context and p is used for control points.

$$
\begin{array}{rcrcl}
P & \in & Prog & = & Meth^* \\
m & \in & Meth & = & Sig \times Code \\
 & & Sig & = & MethId \times \mathbb{N} \\
c & \in & Code & = & Instr^* \\
instr & \in & Instr & & \\
instr & ::= & \multicolumn{3}{l}{Nop \mid Ipush\ n \mid Iinc\ r\ n \quad \text{where } n \in \mathbb{Z}} \\
 & & \multicolumn{3}{l}{Pop \mid Dup \mid Ineg \mid Iadd \mid Isub \mid Imult \mid Idiv} \\
 & & \multicolumn{3}{l}{Load\ r \mid Store\ r} \\
 & & \multicolumn{3}{l}{Getstatic\ f \mid Putstatic\ f} \\
 & & \multicolumn{3}{l}{Newarray \mid Arraylength \mid Iaload \mid Iastore} \\
 & & \multicolumn{3}{l}{Goto\ p \mid If\_icmp\ cond\ p} \\
 & & \multicolumn{3}{l}{\quad \text{where } cond \in \{=, \neq, <, \leq\}} \\
 & & \multicolumn{3}{l}{Invoke\ sig \quad \text{where } sig \in Sig} \\
 & & \multicolumn{3}{l}{Iinput \mid Return}
\end{array}
$$

The instruction set has operators for integer arithmetic and for manipulating local variable, static fields and an operand stack. Instructions on arrays permit to create, obtain the size of, access and update arrays. The flow of control can be modified unconditionally (with *Goto*), and conditionally with the family of conditional instructions *If_icmp cond* which compare the top elements of the run-time stack and branch according to the outcome. Input of data is modelled with the instruction *Iinput*. The inter-procedural layer of the language contains an instruction *Invoke* for invoking a method and an instruction *Return* which transfers control to the calling method, and, at the same time returns the top of the operand stack as result by pushing it onto the operand stack of the caller (see the operational semantics below).

A program state is composed of a frame stack, the value of static fields and a heap of arrays and has the form $<(m, p, s, l)^*, g, h>$. Each frame is a triple composed of a method m, a control point p to be executed next, an operand stack s local to a frame and l a partial mapping from local variables to values. The global heap h is used for storing allocated arrays and is modelled as a partial function from memory locations to arrays. A special error state *Error* models the run-time error arising from indexing an array outside its bounds.

$$\begin{array}{rcrcl} ref & \in & Location & & \\ v & \in & Val & = & \mathbb{Z} + Location \\ s & \in & Stack & = & Val^* \\ l & \in & LocVar & = & R \rightarrow Val \\ a & \in & Array & = & \mathbb{Z}^* \\ h & \in & Heap & = & Location \rightarrow Array_\perp \\ g & \in & Static & = & S \rightarrow Val \\ & & Frame & = & Meth \times \mathbb{N} \times Stack \times LocVar \\ & & State & = & Frame^* \times Static \ \times Heap \\ & & & & + \{Error\} \end{array}$$

The byte code language is given an operational semantics via a transition relation $\rightarrow$ between states. Some of the rules of the definition of $\rightarrow$ are shown in Fig. 3. In the semantics, for a method $m = ((id, n), c)$, we write $m[p]$ for $c[p]$. Note that the language is untyped: registers and fields may (and will) point successively to values of different types during execution. Instructions that require arguments with a certain type get stuck in case of error. Also, the number of registers $|R|$ is the same for all methods. Unused registers and uninitialised fields have the value 0. Finally, we only consider states $<st, g, h>$ such that every location appearing in st, g is in $dom(h)$, which is clearly preserved by the semantics in Fig. 3.

5 Relational analysis of byte code

In this section, we describe a generic, relational analysis for byte code, parameterised with respect to a numeric relational domain used to abstract the values of the local and global variables of the program.

5.1 Symbolic analysis of the stack

Rather than treating each stack location as a new local variable and include this variable in the numeric abstraction describing the state, we integrate a *symbolic de-compilation* into the analysis that abstracts a stack location by a symbolic expression describing how the value at that stack location is computed from the values of the local variables. The operand stack is hence abstracted by a stack of symbolic expressions which represents relation between operands, static fields and local variables.

The following definition of expressions and guards has two purposes: they form the basis of the abstract domain for stacks (*Expr* only), which is specific to stack-based byte code, and they serve as the interface with the numeric relational domain, which is parametric. Note that those two aspects of the analysis are

$$\frac{}{s,l,g,h \xrightarrow{Ipush\ n} n :: s,l,g,h} \qquad \frac{}{n_2 :: n_1 :: s,l,g,h \xrightarrow{Iadd} n_1 + n_2 :: s,l,g,h}$$

$$\frac{l(r) = n}{s,l,g,h \xrightarrow{Iinc\ r\ i} s,l[r \mapsto n+i],g,h} \qquad \frac{}{s,l,g,h \xrightarrow{Load\ r} l(r) :: s,l,g,h}$$

$$\frac{}{v :: s,l,g,h \xrightarrow{Store\ r} s,l[r \mapsto v],g,h} \qquad \frac{}{s,l,g,h \xrightarrow{Getstatic\ f} g(f) :: s,l,g,h}$$

$$\frac{h(ref) = \bot \quad n \geq 0}{n :: s,l,g,h \xrightarrow{Newarray} ref :: s,l,g,h[ref \mapsto 0^n]}$$

$$\frac{h(ref) = a \quad 0 \leq i < |a|}{i :: ref :: s,l,g,h \xrightarrow{Iaload} a[i] :: s,l,g,h} \qquad \frac{h(ref) = a \quad \neg\ 0 \leq i < |a|}{i :: ref :: s,l,g,h \xrightarrow{Iaload} Error}$$

$$\frac{m[p] = instr \quad s,l,g,h \xrightarrow{instr} s',l',g',h'}{<(m,p,s,l) :: st,g,h> \rightarrow_P <(m,p+1,s',l') :: st,g',h'>}$$

$$\frac{m[p] = If\_icmp\ cond\ p' \quad n_1\ cond\ n_2}{<(m,p,n_2 :: n_1 :: s,l) :: st,g,h> \rightarrow_P <(m,p',s,l) :: st,g,h>}$$

$$\frac{m[p] = If\_icmp\ cond\ p' \quad \neg\ n_1\ cond\ n_2}{<(m,p,n_2 :: n_1 :: s,l) :: st,g,h> \rightarrow_P <(m,p+1,s,l) :: st,g,h>}$$

$$\frac{m[p] = Invoke\ (mn,n) \quad m' = ((mn,n),c) \in P}{\begin{array}{l}<(m,p,(v_{n-1} :: \ldots :: v_0 :: s),l) :: st,g,h> \rightarrow_P \\ \quad <m',0,[],[r_0 \mapsto v_0;\ldots;r_{n-1} \mapsto v_{n-1};r_n \mapsto 0;\ldots;r_{|R|} \mapsto 0] :: (m,p,s,l) :: st,g,h>\end{array}}$$

$$\frac{m[p] = Return}{<(m,p,v :: s,l) :: (m',p',s',l') :: st,g,h> \rightarrow_P <(m',p'+1,v :: s',l') :: st,g,h>}$$

Figure 3: Operational semantics of the byte code language

completely independent apart from that.

$$\begin{array}{rll} Expr_V \ni e ::= & n \mid x \mid ? \mid e \diamond e & x \in V, \diamond \in \{+,-,\times,/\} \\ Guard_V \ni t ::= & e \bowtie e & \bowtie \in \{=,\neq,<,\leq,>,\geq\} \end{array}$$

The expression ? represents an unknown value and is responsible for the non-deterministic evaluation of expressions. Analyses will use this expression to model interactive inputs and abstract away numeric quantities not in the scope of the analysis. For instance, our analysis will not keep track of values stored in arrays.

The semantics $[\![e]\!]_\rho$ and $[\![t]\!]_\rho$ of expressions and guards with respect to an environment $\rho \in \mathbb{V} \to \mathbb{Z}$ are given below.

$$\begin{array}{rcl} [\![n]\!]_\rho = \{n\} \qquad [\![x]\!]_\rho = \{\rho(x)\} \qquad [\![?]\!]_\rho = \mathbb{Z} & & \\ [\![e_1 \diamond e_2]\!]_\rho & = & \{n_1 \diamond n_2 \mid n_1 \in [\![e_1]\!],\ n_2 \in [\![e_2]\!]\} \\ [\![e_1 \bowtie e_2]\!]_\rho & \iff & \exists\ n_1 \in [\![e_1]\!]_\rho,\ n_2 \in [\![e_2]\!]_\rho \quad n_1 \bowtie n_2 \end{array}$$

Note that this is not the whole concretisation function for symbolic expressions, which is described later (see Fig. 4).

Symbolic stacks Concrete operand stacks are abstracted by lists of symbolic expressions. To deal correctly with values which are returned after a method

call we use auxiliary variables in a given set A, so the symbolic abstract domain for stacks is $Expr^*_{R+S+A}$.

5.2 Numeric relational domain specification

Apart from symbolic expressions stacks, the byte code analysis is specified with respect to an abstract numeric relational interface (defined below) that can be instantiated with standard relational abstract domains. We thus assume a domain $\mathbb{D}$ parameterised over a (finite) totally ordered set of variables V.

Language independent operators An abstract element is mapped to a set of environments in $\mathbb{Z}^V$ by the concretisation function $\gamma : \mathbb{D}_V \to \mathcal{P}(\mathbb{Z}^V)$. To manage sets of variables, $\mathbb{D}$ is equipped with a projection operator $\exists_{V'} : \mathbb{D}_{V+V'} \to \mathbb{D}_V$, an extension operator $\mathbb{E}_{V'} : \mathbb{D}_V \to \mathbb{D}_{V+V'}$ and a renaming operator $\cdot_{W\to W'} : \mathbb{D}_{V+W} \to \mathbb{D}_{V+W'}$.The abstract domain is also equipped with a partial order $\sqsubseteq \subseteq \mathbb{D}_V \times \mathbb{D}_V$ and meet and upper bound operators $\sqcap, \sqcup : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$. These components are language-independent.

Language dependent operators The abstract assignment of an expression $e \in Expr_V$ to a variable $x \in V$ is modelled by the operator $[\![x := e]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$. A guard $t \in Guard_V$ may be abstracted by two operators $assume^\sharp(t), ensure^\sharp(t) : \mathbb{D}_v$: the $assume^\sharp$ operator computes an over-approximation of the guard, while $ensure^\sharp$ computes an under-approximation.

Definition 5.1 states formally the requirements over the operators of abstract domain $\mathbb{D}_V$.

Definition 5.1. *An abstract domain $\mathbb{D}$ is a family of sets $\mathbb{D}_V$ with:*

- *a concretisation function $\gamma : \mathbb{D}_V \to \mathcal{P}(\mathbb{Z}^V)$,*
- *a decidable ordering relation $\sqsubseteq \subseteq \mathbb{D}_V \times \mathbb{D}_V$ such that*

$$d \sqsubseteq d' \Rightarrow \gamma(d) \subseteq \gamma(d'),$$

- *a projection $\exists_{V'} : \mathbb{D}_{V+V'} \to \mathbb{D}_V$, an extension $\mathbb{E}_{V'} : \mathbb{D}_V \to \mathbb{D}_{V+V'}$ and a renaming $\cdot_{W\to W'} : \mathbb{D}_{V+W} \to \mathbb{D}_{V+W'}$ operators such that:*

$$\begin{aligned}\gamma(\exists_{V'}(d)) &= \{\rho_{|_V} \mid \rho \in \gamma(d)\}\\ \gamma(\mathbb{E}_{V'}(d)) &= \{\rho \mid \rho_{|_V} \in \gamma(d)\}\\ \gamma(d_{W\to W'}) &= \{\rho_{W\to W'} \mid \rho \in \gamma(d)\},\end{aligned}$$

- *a meet operator $\sqcap : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$ such that*

$$\gamma(d \sqcap d') = \gamma(d) \cap \gamma(d'),$$

- *an upper bound operator $\sqcup : \mathbb{D}_V \times \mathbb{D}_V \to \mathbb{D}_V$ such that*

$$\gamma(d \sqcup d') \supseteq \gamma(d) \cup \gamma(d'),$$

- *an abstract assignment operator* $[\![x := e]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$ s.t.

$$\gamma([\![x := e]\!]^\sharp(d)) \supseteq \{\rho[x \mapsto v] \mid \rho \in \gamma(d) \wedge v \in [\![e]\!]_\rho\},$$

- $assume^\sharp, ensure^\sharp : Guard_V \to \mathbb{D}_V$ *such that*

$$\gamma(ensure^\sharp(t)) \subseteq \{\rho \mid [\![t]\!]_\rho\} \subseteq \gamma(assume^\sharp(t)).$$

With the operator $assume^\#$ of the numerical domain we define the abstract test $[\![t]\!]^\sharp : \mathbb{D}_V \to \mathbb{D}_V$ of a guard $t \in Guard_V$ by:

$$[\![e \bowtie e']\!]^\sharp(l^\sharp) = assume^\sharp(e \bowtie e') \sqcap l^\sharp \quad \text{if } \bowtie \in \{=, <, \leq, >, \geq\}$$
$$[\![e \neq e']\!]^\sharp(l^\sharp) = (assume^\sharp(e' < e) \sqcap l^\sharp) \sqcup (assume^\sharp(e < e') \sqcap l^\sharp)$$

The specific rule for $\neq$ is necessary to ensure a good precision with convex polyhedra.

5.3 Analysis specification

The byte code analysis is defined by specifying for each byte code an abstract transfer function which maps abstract states to abstract states (for non-jumping intraprocedural instruction at least). The abstract states are pairs of the form $(s^\sharp, l^\sharp)$ where $l^\sharp$ is a relation between local, global and auxiliary variables and $s^\sharp$ is an abstract stack whose elements are symbolic expressions built from these variables. More precisely, the analysis manipulates the following sets of variables:

R: set of local variables $r_0, \ldots, r_{|L|-1}$ of methods,

R_0: set of old local variables $r_0^{old}, \ldots, r_{|P|-1}^{old}$ of methods, representing their initial values t the beginning of method execution,

S: set of static fields $f_0, \ldots, f_{|S|-1}$ of the program

S_0: set of old static fields $f_0^{old}, \ldots, f_{|S|-1}^{old}$ of the program used to model values of static fields at the beginning of method execution

A: set of auxiliary variable $aux_0, \ldots, aux_{|A|-1}$ used to keep track of results of methods in the symbolic operand stack

Moreover, we use a "primed" version X' of the variable set X for renaming purposes. For each method the analysis computes a signature $Pre \to Post$ whose meaning is

> if the method is called with in a context where its arguments and the static fields satisfy the property *Pre* then if the method returns, then its result, its arguments, and the initial and final values of static fields satisfy the property *Post*.

Preconditions are actually chosen by over-approximating the context in which each method may actually be invoked. Additionally the analysis computes at each control point of each method a local invariant between the current (R) and initial (R_0) values of local variables, the current (S) and initial (S_0) values of static fields, and some auxiliary variables (A) which are used temporarily to remember results of method calls which are still on the stack

instr	F_{instr}
Nop	$(s^\sharp, l^\sharp) \to (s^\sharp, l^\sharp)$
Ipush n	$(s^\sharp, l^\sharp) \to \big(n :: s^\sharp, l^\sharp\big)$
Pop	$(e :: s^\sharp, l^\sharp) \to \big(s^\sharp, l^\sharp\big)$
Dup	$(e :: s^\sharp, l^\sharp) \to \big(e :: e :: s^\sharp, l^\sharp\big)$
Iadd	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(\llcorner e2 + e_1 \lrcorner :: s^\sharp, l^\sharp\big)$
Isub	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(\llcorner e2 - e_1 \lrcorner :: s^\sharp, l^\sharp\big)$
Imult	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(\llcorner e2 \times e_1 \lrcorner :: s^\sharp, l^\sharp\big)$
Idiv	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(\llcorner e2/e_1 \lrcorner :: s^\sharp, l^\sharp\big)$
Ineg	$(e :: s^\sharp, l^\sharp) \to \big(\llcorner 0 - e \lrcorner :: s^\sharp, l^\sharp\big)$
Iinput	$(s^\sharp, l^\sharp) \to (? :: s^\sharp, l^\sharp)$
Load r	$(s^\sharp, l^\sharp) \to \big(\llcorner r \lrcorner :: s^\sharp, l^\sharp\big)$
Store r	$(e :: s^\sharp, l^\sharp) \to \big(s^\sharp[?/r], [\![r := e]\!]^\sharp(l^\sharp)\big)$
Getstatic f	$(s^\sharp, l^\sharp) \to \big(\llcorner f \lrcorner :: s^\sharp, l^\sharp\big)$
Putstatic f	$(e :: s^\sharp, l^\sharp) \to \big(s^\sharp[?/f], [\![f := e]\!]^\sharp(l^\sharp)\big)$
Iinc r n	$(s^\sharp, l^\sharp) \to \big(s^\sharp[\llcorner r - n \lrcorner/r], [\![r := r + n]\!]^\sharp(l^\sharp)\big)$
Newarray	$(e :: s^\sharp, l^\sharp) \to \big(e :: s^\sharp, l^\sharp\big)$
Arraylength	$(e :: s^\sharp, l^\sharp) \to \big(e :: s^\sharp, l^\sharp\big)$
Iaload	$(e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(? :: s^\sharp, l^\sharp\big)$
Iastore	$(e_3 :: e_2 :: e_1 :: s^\sharp, l^\sharp) \to \big(s^\sharp, l^\sharp\big)$

$$\frac{m[p] = instr \notin \{Goto\ p', If\_icmp\ cond\ p', Invoke\ sig, Return\}}{F_{instr}(Loc(m,p)) \sqsubseteq Loc(m, p+1)}$$

$$\frac{m[p] = Goto\ p}{Loc(m,p) \sqsubseteq Loc(m,p)}$$

$$\frac{m[p] = If\_icmp\ cond\ p' \quad Loc(m,p) = (e_2 :: e_1 :: s^\sharp, l^\sharp)}{\big(s^\sharp, [\![e_1\ cond\ e_2]\!]^\sharp(l^\sharp)\big) \sqsubseteq Loc(m,p')}$$

$$\frac{m[p] = If\_icmp\ cond\ p' \quad Loc(m,p) = (e_2 :: e_1 :: s^\sharp, l^\sharp)}{\Big(s^\sharp, [\![e_1\ \overline{cond}\ e_2]\!]^\sharp(l^\sharp)\Big) \sqsubseteq Loc(m,p+1)}$$

$$\frac{m[p] = Invoke\ (mn,n) \quad ((m',n),c') \in P \quad Loc(m,p) = (e_{n-1} :: \cdots :: e_0 :: s^\sharp, l^\sharp)}{\Big(\exists_{R+S_0+A}\big(\textstyle\prod_{i=0}^{n-1} assume^\sharp(e_i = r_i^{old}) \sqcap \exists_{R_0}(l^\sharp)\big)\Big)_{S \to S_0} \sqsubseteq Pre((mn,n),c')}$$

$$\frac{m[p] = Invoke\ sig \quad (sig,c') \in P \quad Loc(m,p) = (e_{n-1} :: \cdots :: e_0 :: s^\sharp, l^\sharp)}{\left(\begin{array}{l} \llcorner aux_j \lrcorner :: s^\sharp[?/aux_j], \\ \exists_{S'+\{res\}} [\![aux_j := res]\!] \left(\begin{array}{l} l^\sharp_{S \to S'} \\ \sqcap\ \exists_{R_0} \left(\begin{array}{l} \prod_{i=0}^{n-1} assume^\sharp(e_i = r_i^{old})_{S \to S'} \\ \sqcap\ Post(sig,c')_{S_0 \to S'} \end{array}\right) \end{array}\right) \end{array}\right) \sqsubseteq Loc(m,p+1)}$$

where p is the index of the $j-$th *Invoke* in m

$$\frac{m[p] = Return \quad Loc(m,p) = (e :: s^\sharp, l^\sharp)}{\exists_{R+A}([\![res := e]\!]^\sharp(l^\sharp)) \sqsubseteq Post(m)}$$

$$\frac{(m,n,c) \in P}{\prod_{i=0}^{|S|-1} assume^\sharp(f_i = f_i^{old}) \prod_{i=0}^{n-1} assume^\sharp(r_i^{old} = r_i) \sqcap Pre(m) \sqsubseteq Loc(m,0)}$$

$$\frac{((\text{main},0),c) \in P}{\top \sqsubseteq Pre((\text{main},0),c)}$$

Figure 4: Relational byte code analysis with stack de-compilation

Definition 5.2 (Abstract domain)**.** *The abstract value for a program P is described by an element $(Pre, Post, Loc)$ of the lattice*

$$\begin{aligned} State^\# = \quad & Meth \to \mathbb{D}_{R_0+S_0} \\ \times\ & Meth \to \mathbb{D}_{R_0+S_0+S+\{res\}} \\ \times\ & Meth \times \mathbb{N} \to \big(Expr_{R+S+A}{}^\star \times \mathbb{D}_{R_0+S_0+R+S+A}\big)_\bot \end{aligned}$$

The analysis is specified as a solution of a constraint (inequation) system associated to each program. The constraint system is formally defined

in Fig 4. Note that extensions are left implicit. For non-jumping intraprocedural instructions, the constraint is defined via a transfer function in $\mathit{Expr}^\star \times \mathbb{D}_{R_0+S_0+R+S+A} \to (\mathit{Expr}^\star \times \mathbb{D}_{R_0+S_0+R+S+A})_\perp$. We (ab)use notation and write $(e :: s^\sharp, l^\sharp) \to \left(e :: e :: s^\sharp, l^\sharp\right)$ for the function that maps a state of the form $(e :: s^\sharp, l^\sharp)$ to the resulting state $\left(e :: e :: s^\sharp, l^\sharp\right)$ and other states to $\perp$. The analysis maintains a symbolic version of the operand stack and most of the transfer functions are defined as symbolic executions. The transfer functions for the stack operations *Nop*, *Pop* and *Dup* mimic the semantics of those operations so *e.g.*, *Dup* will duplicate the expression on top of the (abstract) operand stack and hence is abstracted by the function $(e :: s^\sharp, l^\sharp) \to \left(e :: e :: s^\sharp, l^\sharp\right)$. The abstraction of the instruction *Load r* for fetching the value of local variable r just pushes the expression $\llcorner r \lrcorner$ onto the abstract stack (rather than projecting an abstract value of r from the relation describing the local variables). Similarly, the abstraction of the addition operation *Iadd* pops the two topmost expressions e_1 and e_2 from the abstract stack and replaces them with the symbolic expression $\llcorner e_2 + e_1 \lrcorner$.

The transfer function for the *Store r* operation updates the abstract environment of local variables with the constraint that r is now equal to the value given by the expression e on top of the abstract stack top. Formally, this is done using the operation $[\![x := e]\!]^\sharp$ provided by the interface of the relational domain. By the same token, all occurrences of the sub-expression $\llcorner x \lrcorner$ in the abstract stack become invalid, as r now (potentially) has changed value, and are replaced by the "don't know" expression $\llcorner ? \lrcorner$. The analysis abstracts arrays references by the length of the referenced array, so the transfer functions for *Newarray*(which takes the length as argument and returns a reference to the created array) becomes the identity function. Similarly for *Arraylength*.

For all non-jumping instructions, we generate a constraint saying that the state following the instruction should include the result of applying the transfer function of the instruction to the state preceding the instruction. For the conditional *If_icmp cond p'*, we use the abstract tests provided by the relational domain to take the outcome of the test into account, so *e.g.*, at program point p' we know that the condition *cond* holds between the two top elements of the stack. If these are given by expressions e_1 and e_2 then we know that the symbolic expression $\llcorner e_1 \ \mathit{cond}\ e_2 \lrcorner$ evaluates to true in the current environment. The expression $[\![e_1 \ \mathit{cond}\ e_2]\!]^\sharp(l^\sharp)$ in the rule for conditionals updates the environment of local variables ($l^\sharp$ to take this information into account. A similar constraint is generated for the program point $p+1$ using this time the negation $\overline{\mathit{cond}}$ of the condition *cond*.

The analysis of method calls is the most complicated part. The complications partly arise because we have several kinds of variables (static fields, local and auxiliary variables) whose different scope must be catered for. The analysis gives rise to two constraints: one that relates the state before the call to the pre-condition of the method and one that registers the impact of the call on the state immediately following the call site.

When invoking a method m' from method m, we compute an abstract state that holds before starting executing m' and which constrains the $Pre(m')$ component of the abstract element describing m'. This state registers that the n topmost expressions $e_1, \ldots, e_n$ on the abstract stack corresponds to the actual arguments that will be bound to the local variables of the callee m', by inject-

ing the constraints $e_i = r_i^{old}$ into the relational domain and adding them to the current state as given by $l^\sharp$. Care must be exercised not to confound the parameters R_0 of the caller with the parameters of the callee, hence the projecting out of R_0 before joining the constraints. Furthermore, the local variables R, the initial values of static fields S_0 and the auxiliary variables A of method m have a different meaning in the context of method m' and are removed from the abstract state at the start of m' too. Finally, the current value of static fields S in m at the point of the method call becomes the initial value of the static fields when analysing m', hence the renaming of S into S_0. The entire start state for m' is thus described by the expression

$$\left(\exists_{R+S_0+A}\left(\bigsqcap_i assume^\sharp(e_i = r_i^{old}) \sqcap \exists_{R_0}(l^\sharp)\right)\right)_{S\to S_0}$$

The second rule for *Invoke* describes the impact of the method call on its successor state. We use an auxiliary variable aux_j (chosen to be free in $s^\#$) to name the result of a method call which is pushed onto the stack. This variable is constrained to be equal to the variable *res* which receives the value returned by m'. The rest of the left-hand side expression of the constraint

$$l^\sharp_{S\to S'} \sqcap \exists_{R_0}\left(\bigsqcap_i assume^\sharp(e_i = r_i^{old})_{S\to S'} \sqcap Post(m')_{S_0\to S'}\right)$$

serves to link the post-condition $Post(m')$ of the method with the state $l^\sharp$ of the call site. These are linked via the local variables x_i constrained to be equal to the argument expressions e_i and via the global static fields S. Again, some renaming and hiding of variables is required: *e.g.*, the initial values of the static fields in m', referred to by S_0, correspond to the values of the static fields before the call in the state $l^\sharp$ and in the expressions e_i, referred to by S. The renamings $S_0 \to S'$ and $S \to S'$, respectively, ensures that these values are identified.

Two rules are used to initiate the analysis of a method (constraint on $Loc(m, 0)$) and of the entire program (constraint on $Pre((\text{main}, n), c)$). To initialise the analysis of a method m, the precondition $Pre(m)$ is conjoined with the constraints linking the variable f_i^{old} to the current value of the static field f_i and linking the parameters r_i^{old} with the local variables r_i, in accordance with how parameters are handled in *e.g.* Java byte code. The analysis of the main method starts in the completely unconstrained state $\top$.

5.4 Inference

The constraint system presented in the previous section can be turned into a post-fixpoint problem by standard techniques. Consequently, the solutions of the system can be characterised as the set of post-fixpoints $\{x \mid F^\sharp(x) \sqsubseteq x\}$ of a suitable monotone operator $F^\sharp \in State^\sharp \to State^\sharp$ operating on the global abstract domain $State^\sharp$ of the analysis. Assuming that $State^\sharp$ is a complete lattice[1] we know that the least solution $lfpF^\sharp$ of this problem exists and can be over-approximated by any post-fixpoint of $F^\sharp$. Computing such a post-fixpoint is the role of chaotic iterations [12] which operate on the equation

[1]For the polyhedra abstract domain this assumption is too strong but we can relax it by considering a complete lattice containing $State^\sharp$ and all its upper bounds [13].

system associated with the constraint system and choose a suitable iteration strategy [8]. Iteration is sped up by using widening on well-chosen control points. Neither the iteration strategy nor the widening operators belong to the TCB since the validity of the result can be checked with a post-fixpoint test.

5.5 Safety checks

Once the analysis has inferred correct invariants, this information is used to check if they enforce the suitable safety policy. In a context of array bound checking we must check that each array access is within the bounds of the array. As a consequence, for each occurence of an instruction *Iaload* or *Iastore* at a program point (m, pc), we test if the local invariant $Loc(m, pc)$ computed by the analysis ensures a safe array access.

Definition 5.3 (Abstract safety checks)**.** *We say a set of local invariant* $Loc \in (\mathbb{N} \to (Expr^\star \times \mathbb{D}_{P+S_0+L+S+A})_\perp)$ *verifies all safety checks of a program if and only if*

$$
\begin{array}{l}
\forall m \in P,\ pc \in \mathbb{N}, \\
\quad m[p] = Iaload \ \Rightarrow \\
\qquad Loc(m, pc) = (e_2 :: e_1 :: s^\sharp, l^\sharp) \Rightarrow \\
\qquad l^\sharp \sqsubseteq ensure^\sharp(\llcorner 0 \leq e_2 \lrcorner) \ \wedge \ l^\sharp \sqsubseteq ensure^\sharp(\llcorner e_2 < e_1 \lrcorner) \\
\quad \wedge \\
\quad m[p] = Iastore \ \Rightarrow \\
\qquad Loc(m, pc) = (e_3 :: e_2 :: e_1 :: s^\sharp, l^\sharp) \Rightarrow \\
\qquad l^\sharp \sqsubseteq ensure^\sharp(\llcorner 0 \leq e_2 \lrcorner) \ \wedge \ l^\sharp \sqsubseteq ensure^\sharp(\llcorner e_2 < e_1 \lrcorner)
\end{array}
$$

5.6 Soundness of the analysis

Fig. 5 gives the concretisation functions for the abstract domains. The auxiliary abstraction function β maps everything to an integer, abstracting arrays by their length. γ^{expr} defines concretisation of a symbolic expression with respect to an environment. γ^{Pre} maps pre-conditions to sets of calling contexts, γ^{Post} maps post-conditions to relations between calling contexts and return contexts, and γ^{Loc} maps local invariants to relations between calling contexts and local program states. Note that concretisations contain only states such that all locations that are being referenced are defined in the heap.

Definition 5.4 (Reachable states)**.** *For a method* m *in a program* P*, a heap* h*, a static heap* g*, a set of local variables* l*, a frame stack* st*, the set* $[\![P]\!]^m_{h,g,l,st}$ *of reachable state from an execution of* m *starting in an initial configuration* (h, g, l, st) *is defined by*

$$
[\![P]\!]^m_{h,g,l,st} = \left\{ s \mid <(m, 0, [], l') :: st, g, h> \xrightarrow{\geq st}{}^*_P s \right\}
$$

where $\xrightarrow{\geq st}{}^*_P$ *is the reflexive transitive closure of* $\rightarrow_P$ *restricted to states who have a form* $< \ldots :: (m, \ldots) :: st, \ldots >$*.*

The purpose of $\xrightarrow{\geq st}{}^*_P$ is to collect only the states in between the start and the end of the execution of a particular stack frame.

$$
\begin{array}{rrcl}
\beta_{\mathbb{V}} : & Heap \times (\mathbb{V} \to Val) & \to & (\mathbb{V} \to \mathbb{Z}_\perp) \\
& (h, z) & \mapsto & \lambda x. \begin{cases} z(x) & \text{if } z(x) \in \mathbb{Z} \\ |h(z(x))| & \text{if } z(x) \in dom(h) \end{cases} \\
\gamma^{expr}_{h,g,l,a} : & Expr_{R+S+A} & \to & \mathcal{P}(Val), \quad h \in Heap, g \in Static,\ l \in LocVar \\
& & & \qquad \text{and } a \in Val \to \mathbb{Z} \\
& e & \mapsto & [\![e]\!]_{\beta_{R+S+A}(h, l \oplus g \oplus a)} \\
& & & \cup \left\{ ref \in dom(h) \;\middle|\; |h(ref)| \in [\![e]\!]_{\beta_{R+S+A}(h, l \oplus g \oplus a)} \right\}
\end{array}
$$

$$
\begin{array}{rrcl}
\gamma_{Pre} : & \mathbb{D}_{R_0+S_0} & \to & \mathcal{P}(Static \times Heap \times LocVar) \\
& pre & \mapsto & \{\ (g_0, h_0, l_0) \mid \beta_{R_0+S_0}(h_0, l_0 \oplus g_0) \in \gamma(pre)\ \} \\
\gamma_{Post} : & \mathbb{D}_{R_0+S_0+S+\{res\}} & \to & \mathcal{P}((Static \times Heap \times LocVar) \times (Static \times Heap \times Val)) \\
& post & \mapsto & \left\{ \begin{array}{l} ((g_0, h_0, l_0), (g, h, v)) \mid \\ \quad \beta_{S_0+R_0+S+\{res\}}(h_0, g_0 \oplus l_0 \oplus g \oplus [res \mapsto v]) \in \gamma(post) \end{array} \right\} \\
\gamma_{Loc} : & Expr^\star \times \mathbb{D}_{R_0+S_0+R+S+A} & \to & \mathcal{P}\left(\begin{array}{l} (Static \times Heap \times LocVar) \\ \times\ (Static \times Heap \times Stack \times LocVar) \end{array} \right) \\
& (e_1 :: \cdots :: e_n, loc) & \mapsto & \left\{ \begin{array}{l} ((g_0, h_0, l_0), (g, h, v_1 :: \cdots :: v_n, l)) \mid \\ \quad \exists a \in A \to \mathbb{Z},\ \forall i \in [\![1, n]\!],\ v_i \in \gamma^{expr}_{h,g,l,a}(e_i) \wedge \\ \quad\quad \beta_{S_0+R_0+S+R+A}(h_0, g_0 \oplus l_0 \oplus g \oplus l \oplus a) \in \gamma(loc) \end{array} \right\}
\end{array}
$$

Figure 5: Concretisation functions

Definition 5.5 (Safe method)**.** *A method m in a program P is said to be* safe *wrt. a precondition $Pre \subseteq Heap \times Static \times LocVar$ if for all stack frames st and all $(h, g, l) \in Pre$, $Error \notin [\![P]\!]^m_{h,g,l,st}$.*

Theorem 5.6 (Correctness)**.**
Let P be a program and $(Pre, Post, Loc)$ a solution of the constraint system associated with P. If Loc satisfies all safety checks then every method m in P is safe wrt. to $Pre(m)$. In particular,

$$<(((main, n), c), 0, [], \lambda r.0) :: [], \lambda f.0, \lambda ref.\perp > \not\to_P Error$$

Proof. The proof is divided into two parts. We first prove that each reachable intermediate state at a point (m, p) satisfies the property $\gamma_{Loc}(Loc(m, p))$, that each method m is called in a context satisfying $\gamma_{Pre}(Pre(m))$ and that its return value (if it exists) satisfies $\gamma_{Post}(Loc(m))$. In the second part we prove that if a state at some point (m, p) satisfies $\gamma_{Loc}(Loc(m, p))$ as well as the abstract safety check associated with this point, then no error happens in the next semantic step. To deal with the steps corresponding to procedure calls, the proof makes use of an intermediate big-step operational semantics. Details are omitted for lack of space. □

6 Polyhedral analysis

We now instantiate the relational analysis framework using linear relations in the form of convex polyhedra. Polyhedral program analysis has a well-established theory [13] with several implementations [4, 15]. Here, we recall the basics of this theory.

Definition 6.1. *Convex polyhedra of dimension n ($\mathbb{P}_n \subseteq \mathbb{Q}^n$) are (convex) subsets of $\mathbb{Q}^n$ that can be expressed as a finite intersection of half-planes of $\mathbb{Q}^n$.*

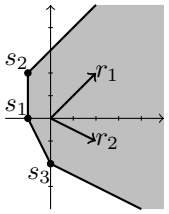


Figure 6: Dual representation of polyhedra

Polyhedra can be represented as sets of linear constraints. It is desirable to keep these sets in normal form *i.e.,* without redundant constraints. For this purpose, polyhedra libraries maintain a dual representation of polyhedra based on *generators* in which a convex polyhedron is the convex hull of a (finite) set of *vertices*, *rays* and *lines*. Vertices, rays and lines are respectively extremal points, infinite directions and bi-directional infinite directions of the polyhedron. Fig. 6 shows a a polyhedron with four constraints whose dual representation is made of three *vertices* (s_1,s_2,s_3) and two *rays* (r_1,r_2).

The efficiency of the algorithm that maintains the normal form of the double description is of crucial importance. For this task, state-of-the-art polyhedral libraries [4, 15] use Chernikova's algorithm [11]. In the worst case, the number of generators is exponential in the number of constraints (and *vice-versa*) but, in practise, the double description offers a good performance. To alleviate further the cost of normalising polyhedra, these libraries switch lazily from one representation to the other.

Polyhedral cannot directly handle expressions that fall outside the linear fragment. It would be sound but unsatisfactory to abstract those expressions towards an arbitrary value *i.e.*, the ? expression. More information can be retained by *linearising* expressions [19]. For instance, the precise analysis of Binary Search (Fig. 1) requires a precise model of euclidean divisions. Given an integer constant n, the guard $y = x/n$ is abstracted by the linear guards $0 \leq x - n \cdot y < n$. Multiplications can also be linearised by using the range of variables.

We now briefly explain how polyhedral algorithms implement the abstract numeric relational domain specified in Definition 5.1. To be implemented efficiently, the double description of polyhedra is needed, using Chernikova's algorithm to reconstruct the coherence of the double representation.

The convex polyhedron can directly be cast into an abstract numeric domain by mapping variables of the domain to dimensions of the polyhedron. Hence, we get $\mathbb{D}_V = \mathbb{P}_{|V|}$ and the concretisation:

$$\gamma(P) = \{\rho \in \mathbb{Z}^V \mid \rho \in P \cap \mathbb{Z}^V\}$$

Renaming of variables consists in applying a permutation to the dimensions of polyhedron. The **extension** operation which add new variables consists in inserting new unconstrained dimensions at the relevant indexes.
Projections can be efficiently performed on the *generator* description of polyhedra in linear time. Each generator is projected by erasing the now irrelevant dimensions.

Intersections are computed by taking the union of the constraints of each polyhedron.
The **convex hull**, *i.e.*, least upper bound, is computed by taking the union of the generators of both polyhedra.
An **assignment** $[\![x := e]\!]^\sharp$ is modelled by the linear transformation (if e is linear) that keeps all the variables unchanged except x which is mapped to e. The transformation is applied to the generators.
Inclusion tests are using both representation at once. Checking the containment of two polyhedra ($P \sqsubseteq Q$) amounts to verifying that the generators of P satisfy the constraints of Q.
Widening operators are used by the fixpoint iterator to ensure convergence. For convex polyhedra, there exist various widening operators [13, 3].
Assume and ensure operators are responsible for interpreting guards of the target language. If the guard t is linear, a polyhedron is built from it and no abstraction takes place. Otherwise, t has to be linearised. In the worst case, universal (resp. empty) polyhedra can be used as sound (though very imprecise) fallbacks.

7 Fixpoint pruning

The result of the polyhedral byte code analysis will be a fixpoint of the transfer functions, representing an invariant of the program under analysis. This invariant will often contain more information than necessary for proving a particular safety policy such as absence of indexing outside array bounds. In the following we show how to *prune* an invariant with respect to a given safety policy, resulting in an invariant that is smaller and cheaper to verify.

7.1 Witnesses and pruning

We have applied the technique described in [7] for pruning constraint-based invariants, with some adaptations allowing to handle our interprocedural polyhedral analysis on byte code better. First we recall the definition of witnesses for this particular analysis.

Definition 7.1. *A witness for a program P is a solution (Pre, Post, Loc) to the constraint system associated with P that satisfies the safety checks of P (see Definition 5.3).*

We use this as the basis for building certificates, relying on the fact that if there exists a witness for P then P is safe (see Theorem 5.6). Part of the witness is sent to the checker in the constraint representation only (see Section 6), so we aim at extracting a weaker witness with fewer linear constraints than the one produced by the inference algorithm of Section 5.4 (if the analysis is accurate enough for the program). Pruning leaves the symbolic expression stacks of the witness unchanged because the checker recomputes them (and hence nothing is transmitted about this part).

It is easy to see that there is generally no unique weakest witness nor a unique witness with the minimum number of constraints (because the analysis is not *distributive*). Also, the idea of starting from the safety requirements to compute backward a witness that satisfies them cannot achieve the same precision as a

prune(w) :=
  let $\overline{w'} = \emptyset$
  while w' is not a witness do
    choose a constraint C and $k \in w'_{|dep(C)}$ *s.t.* $\overline{w'}, \{k\} \not\vdash C$
      (or a check C such that $\overline{w'} \not\vdash C$)
    choose $\overline{x} \subseteq (\overline{w} \setminus \overline{w'})_{|dep(C)}$ such that $\overline{w'} \cup \overline{x}, \{k\} \vdash C$
      (respectively, $\overline{w'} \cup \overline{x} \vdash C$)
    $\overline{w'} := \overline{w'} \cup \overline{x}$
  done
  return w'

Figure 7: Witness pruning algorithm

forward analysis, because intuitively it would have to guess the invariants that a forward analysis naturally discovers. For these reasons we use a technique of pruning that removes as many linear constraints as possible from a given witness.

7.2 Abstract algorithm

We use a variation of the greedy heuristic presented in [7]. In the following we identify polyhedra with sets of constraints. We use

$$\begin{aligned} Var = \quad & \{\mathrm{pre}_m \mid m \in P\} \cup \{\mathrm{post}_m \mid m \in P\} \\ & \cup \{\mathrm{loc}_{m,p} \mid m \in P,\ m = ((mn, n), c),\ p < |c|\} \end{aligned}$$

to denote the set of unknowns of the constraint system associated with P. For an abstract element $x = (Pre, Post, Loc)$ we define the set of linear constraints of x:

$$\begin{aligned} \overline{x} = \bigcup_{m \in P} \quad & \{(\mathrm{loc}_{m,p}, k) \mid Loc(m,p) = (s^\#, l^\#),\ k \in l^\#\} \\ & \cup \{(\mathrm{pre}_m, k) \mid k \in Pre(m)\} \\ & \cup \{(\mathrm{post}_m, k) \mid k \in Post(m)\} \end{aligned}$$

For $V \subseteq Var$ we define $\overline{x}_{|V} = \{(var, k) \in \overline{x} \mid var \in V\}$ and $x_{|V}$ is defined accordingly.

Recall that the constraint system for P is a set of constraints of the form $F(x) \sqsubseteq x_{|\{v\}}$ where $v \in Var$. For a constraint c we note $\overline{x}, \overline{y} \vdash C$ if $F(x) \sqsubseteq y_{|\{v\}}$ (we can do so since the expression stacks are fixed) and $\overline{x} \vdash C$ for $\overline{x}, \overline{x} \vdash C$. We will overload the notation and write also $\overline{x} \vdash C$ if x satisfies the safety check C. Then, for every such constraint C, we define a set $dep(C) \subseteq Var$ that represents the dependencies of this constraint, in the sense that if $\overline{x}, \overline{y} \vdash C$ then $\overline{x}_{|dep(C)}, \overline{y} \vdash C$. The definition of dep is straightforward. For example, if C is the constraint $F_{instr}(Loc(m,p)) \sqsubseteq Loc(m, p+1)$ corresponding to an non-jumping intraprocedural instruction (see the first part of Fig. 4), then $dep(C) = \{\mathrm{loc}_{m,p}\}$. For the constraint $\ldots \sqsubseteq Loc(m, p+1)$ of an *Invoke sig* instruction, $dep(C) = \{\mathrm{loc}_{m,p}, \mathrm{post}_{(sig,c)}\}$ where $(sig, c) \in P$.

The pruning algorithm is shown in Fig. 7. The main issue in this non-deterministic algorithm is the choice of the subset $\overline{x}$: we obviously want a minimal one in the sense of set inclusion (achievable in reasonable time by monotonicity), but it is not unique.

7.3 Efficient pruning for polyhedral byte code analysis

Our strategy is to take a minimal such $\overline{x}$ that almost minimizes a cost function taking into account the number of linear constraints, the number of non-null coefficients in them, and, for *Invoke*, the number of post constraints (as opposed to loc). This allows us to obtain a witness with simpler invariants and signatures. The heuristic blindly applies the definition of $\vdash$ while labelling (part of) the search space. The dependency function *dep* helps by reducing the number of linear constraints to be considered at each step.

Finally, we face a problem specific to the polyhedra domain when pruning an invariant: in order to keep things small, the polyhedra are usually represented in a minimal form in which the relation between a set of dimensions does not necessarily appear as a dedicated linear constraint, but often as a consequence of several other relations. For example, the constraint $x \leq z$ is implicit in $x \leq y \leq z$. For the purpose of finding a small invariant, we may benefit from being able to include such constraints. Our solution is to add some implicit constraints to the invariant before pruning it. More precisely, for a polyhedron in $\mathbb{D}_V$, we add all the projections $\exists_{V \setminus V'}$ (see Section 6) where V' is a subset of V of cardinality at most n. For the maximal number n of dimensions in the implicit constraints to be generated, ∞ seems too costly for non-trivial programs, and unnecessary. It turns out that 3 is enough for all of our examples, which is not surprising because very few correctness proofs actually rely on linear invariants involving more than three variables.

8 Result checking of polyhedral analysis

A result checker for abstract interpretation based static analysis can be reduced to an (optimised) fixpoint checker [1], with the downside that the abstract domains are still part of the TCB. Formally certifying optimised polyhedral libraries [4, 15] is feasible but would require an enormous certification effort. Instead, we propose a lightweight verifier of polyhedral analyses using a result checking methodology which has two advantages: i) the TCB is small, and ii) the checking time is optimised.

8.1 The polyhedral domain revisited

Chernikova's algorithm is at the origin of the computational complexity of convex polyhedra operations, so a first approach would be to design a result checker for Chernikova's algorithm *i*.e., a normal form checker. This has the inconvenience that most of the polyhedral operations would be annotated with their result together with a certificate attesting that it is in normal form. Instead, we develop a checker which only uses the constraint representation of polyhedra and which never need to normalise. Moreover, projections are not computed but delayed using a set of extra *existential* variables. More precisely, our polyhedra are represented by a list of linear expression over two disjoint sets of variables V and E. Variables in $v \in V$ are genuine variables while $e \in E$ are (existential) variables that represent dimensions which have been projected out.

Definition 8.1. *Let V and E be disjoint sets of variables.*

$$\mathbb{P}_V = Lin^*_{V+E}$$

where

$$Lin_{V+E} = \{\llcorner c_1 \times x_1 + \cdots + c_n \times x_n \lrcorner \mid c_i \in \mathbb{Z} \wedge x_i \in V + E\}.$$

Given $es \in \mathbb{P}_V$, the concretisation function is defined by

$$\gamma_V(es) = \{\rho_{|V} \mid \forall k \in es, [\![k \geq 0]\!]_\rho\}$$

In the following, we show how to implement the polyhedral operations using (only) polyhedra in constraint form.

Renaming simply consists in applying the renaming to the expressions within the polyhedron. Because the existential variables belong to a disjoint set, no capture can occur. In addition, for this encoding, **extension** is a no-op because unused variables have no impact on the internal representation.

$$es \in \mathbb{P}_V \Rightarrow \forall W \supseteq V, es \in \mathbb{P}_W$$

Using Fourier-Motzkin elimination (see *e.g.*), [23], **projections** can be computed directly over the constraint representation of polyhedra However, in the worst case, the number of constraints grows exponentially in the number of variables to project. To solve this problem, we delay the projection and simply register them as existentially quantified. This is done by renaming these variables to fresh variables.

To compute **intersections**, care must be taken not to mix up the existential variables. To avoid capture, existentially variables are renamed to variables that are fresh for both polyhedra. Thereafter, the intersection is implemented by taking the union of the expressions.

To implement the **assume** and **ensure** operators, the involved expressions are first linearised and the obtained linear inequality is put into the form $e \geq 0$ which now belongs to the set Lin defined above.

For convex polyhedra, **assignment** is efficiently implemented as an atomic operation. However, it can be expressed in terms of the previous operators: given x' a fresh variable, an assignment can be defined as follows.

$$[\![x := e]\!]^\sharp(P) = \left(\exists_{\{x\}} \left(P \sqcap assume^\sharp(x' = e)\right)\right)_{\{x'\} \to \{x\}}$$

It is this latter definition that we use.

Widening operators are only used during the fixpoint iteration, and are not needed at checking time.

Convex Hull is the typical operation that is straightforward to implement using the generator representation of polyhedra. Using a relaxation technique, it is possible to express the convex hull as the projection of a polyhedron of higher dimension [2] but since this requires to compute projections this does not scale. Even with our delaying of projections, the size of the polyhedron doubles. Instead of computing a convex hull, we follow the result certification methodology and provide a certificate polyhedron that is the result of the convex hull computation. Furthermore, our result checker need not check that the result is exactly the convex hull but only that it is an upper bound by doing a double inclusion test.

$$isUpperBound(P, Q, UB) \equiv P \sqsubseteq UB \wedge Q \sqsubseteq UB$$

To implement **inclusion tests**, we push the result certification methodology further and use inclusion certificates. The form of certificates and their generation are described below.

8.2 Result certification for polyhedral inclusion

Farkas lemma (Lemma 8.2) is a theorem of linear programming (see for instance [23]) which gives a notion of *emptiness* certificate for polyhedra. In this part, we show how this result can be i) lifted to obtain an inclusion checker; ii) extended further to deal with existential variables. Our inclusion checker $\sqsubseteq_{check}$ takes as input a pair of polyhedra (P, Q) and an inclusion certificate. It will only return true if the certificate allows to conclude that P is indeed included in Q ($P \sqsubseteq Q$).

Lemma 8.2 (Farkas Lemma)**.** *Let $A \in \mathbb{Q}^{m\times n}$ and $b \in \mathbb{Q}^n$. The following statements are equivalent:*

- *For all $x \in \mathbb{Q}^n$, $\neg(A \cdot x \geq b)$*
- *There exists $ic \in \mathbb{Q}^m$ satisfying $A^t \cdot ic = \bar{0}$ and $b^t \cdot ic > 0$.*

The soundness ($\Leftarrow$) of certificates is the easy part and is all that is needed in the machine-checked proof. It follows that the existence of a certificate ensures the infeasibility of the linear constraints and therefore that the polyhedron made of these constraints is empty.

Thus, an *inclusion certificate* ic is a vector of $\mathbb{Q}^m$ and checking a certificate consists of 1) computing a matrix-vector product ($A^t \cdot ic$) 2) verifying that the result is a null vector; 3) computing a scalar product ($b^t \cdot ic$); and 4) verifying that the result is strictly positive. All in all, the certificate checker runs in quadratic-time in terms of arithmetic operations.

Certificates generation can be recast as a linear programming problem that can be efficiently solved by either the Simplex or interior point methods. The set of certificates is characterised by the convex polyhedron

$$Cert = \left\{ ic \,\middle|\, ic \geq \bar{0} \wedge b^t \cdot ic > 0 \wedge A^t \cdot ic = \bar{0} \right\}$$

As a result, finding an *extremal* certificate amounts to solving a linear optimisation problem. For instance, the solution of the linear program $min\{c^t \cdot \bar{1} \mid c \in Cert\}$ minimises the sum of the coefficients of the certificate. In theory, such a minimisation might not yield a compact certificate because the optimisation is done over the rationals – there are very small rationals that require many bits. However, in practise, the technique is sufficiently efficient.

From emptiness to inclusion Lemma 8.3 states that in the absence of existential variables an inclusion check amounts to emptiness checks.

Lemma 8.3. *Given $P, P' \in \mathbb{P}_{V+E}$, we have*

$$\forall e' \in P', \gamma_{V+E}(-e'-1 :: P) = \emptyset$$

if and only if

$$\gamma_{V+E}(P) \subseteq \gamma_{V+E}(P').$$

Proof. By construction, polyhedra in $\mathbb{P}_{V+E}$ do not have existential variables. Hence, we have $\gamma_{V+E}(P') = \bigcap_{e'\in P'} \gamma_{V+E}(e'::[])$. Moreover, the complement of a linear constraint $e' \geq 0$ is $-e' - 1 \geq 0$. These facts allow to reduce inclusion to a set of emptiness tests. □

Lemma 8.4 states that to do an inclusion test, it is sound to drop existential variables.

Lemma 8.4. *Let P and P' be polyhedra in constraint form.*

$$\gamma_{V+E}(P) \subseteq \gamma_{V+E}(P') \Rightarrow \gamma_V(P) \subseteq \gamma_V(P')$$

Proof. The Lemma follows from the definition of γ and the fact that the restriction operator on environments is monotone. □

Together, Lemma 8.3 and Lemma 8.4 allow the design of a sound result checker for inclusion tests of form $P \subseteq P'$. In general, the checker is incomplete but this only shows up in cases where P' has existential variables. However, inclusions only need to be certified when P' is a polyhedron computed by the analyser and such a P' *does not* contain existential variables, so the inclusion checker is always used in a context where it is complete.

9 Implementation and Experiments

The relational byte code analysis has been implemented in Caml and instantiated with the efficient NewPolka polyhedral library [15] as its relational abstract domain. As the language presented in Section 4 is compatible with Java byte code, we analyse programs that are compiled from genuine Java programs. The result checker for polyhedral analysis described in Section 8 has been implemented in the Coq proof assistant.

The analyser computes a solution to the constraint system generated from a program and passes it on to the fixpoint pruning algorithm. From the compressed invariants, loop headers and join points are extracted and the inclusion certificates required by the checker are produced using the Simplex algorithm. A binary form of loop headers, join point invariants and their inclusion certificates constitute the final program certificate.

In Fig 8, we give a table of some of the benchmarks used by Xi to demonstrate the dependent type system for Xanadu [27]. HeapSort and QuickSort are particularly good illustrations of the capacities of our analysis for analysing automatically programs mixing recursive procedures and loops. The programs and the analysis results can be found at `http://www.irisa.fr/lande/polycert.html`.

	.class	certificates		checking time	
Program		before	after	before	after
BSearch	515	22	12	0.005	0.007
BubbleSort	528	15	14	0.0005	0.0003
HeapSort	858	72	32	0.053	0.025
QuickSort	833	87	44	0.54	0.25

Figure 8: Class files in bytes, certificates in number of constraints, time in seconds

The two checking times in the last column give the checking time with and without fixpoint pruning Three things are worth noticing. First, invariants that had to be produced manually for the Xanadu examples are now inferred

automatically. Second, the checking time is small, especially given that the checker is extracted from its Coq specification and thus is a purely functional implementation of a polyhedral domain. Third, pruning can halve the number of constraints to verify. This reduction can sometimes but not always produce a similar reduction in checking time. The benchmarks are relatively modest in size and it is well known that full-blown polyhedral analyses have scalability problems. Our analyser will not avoid this but can be instantiated with simpler relational domains such as *e.g.*, octagons, without having to change the checker.

10 Related work

A number of relational abstract domains (octagons [18], convex polyhedra [13], polynomial equalities [20]) have been proposed with various trade-offs between precision and efficiency, and intra-procedural relational abstract interpretation for high-level imperative languages is by now a mature analysis technique. However, to the best of our knowledge the present work is the first extension of this to an inter-procedural analysis for byte code. Dependent type systems for Java-style byte code for removing array bounds checks have been proposed by Xi and Xia [28]. The analysis of the stack uses singleton types to track the values of stack elements, achieving the same as our symbolic stack expressions. The analysis is intra-procedural and does not consider methods (they are added in a later work [27] which also adds a richer set of types). The type checking relies on loop invariants. We have run our analysis on the example Xanadu programs given by Xi and have been able to infer the invariants necessary for verifying safe array access automatically.

The area of certified program verifiers has been an active field recently. Wildmoser, Nipkow *et al.* [25] were the first to develop a fully certified VCGen within Isabelle/HOL for verifying arithmetic overflow in Java byte code. The certification of abstract interpreters has been developed by Cachera, Pichardie *et al.* [9, 21]. for a variety of analyses including class analysis of Java byte code and interval analysis. Lee *et al.* [16] have certified the type analysis of a language close to Standard ML in LF and Leroy [17] has certified some of the data flow analyses of a compiler back-end. Leroy also observes that for certain, more involved analyses such as the register allocation, it is simpler and sufficient to certify a checker of the result than the analysis itself. The same idea is used by Wildmoser *et al.* [24] who certifies a VCGen that uses untrusted interval analysis for producing invariants and that relies on Isabelle/HOL decision procedures to check the verification conditions generated with the help of these invariants. Their technique for analysing byte code is close to ours in that they also use symbolic expressions to analyse the operand stack and the main contribution of the work reported here with respect to theirs is to develop this result checking approach for a fully relational analysis.

The idea of removing useless parts from an invariant was developed independently by Besson *et al.* [7] and by Yang *et al.* [29] who call it abstract value slicing. Both works deal with intra-procedural invariants and both are based on a dependency computation that selects, for every constraint $F(X) \sqsubseteq Y$ of the constraint system of P and every subset of an abstract state Y, a sufficient subset of X that satisfies the constraint. The two methods differ in the way that this choice is done but both have been shown viable for intra-procedural pruning of

relational invariants. The present work is an extension of the principles underlying the non-deterministic algorithm in [7] to handle the pre-/post-conditions arising from the interprocedural analysis. Finally, it should be noted that the fixpoint compression is orthogonal to and compatible with the optimisation of iteration strategies for fixpoint checking underlying Lightweight Bytecode Verification [22] and the more general abstraction-carrying code [1, 6]. Our checker combines both techniques.

11 Conclusions and future work

This paper demonstrates the feasibility of an interprocedural relational analysis which automatically infers polyhedral loop invariants and pre-/post-condition for programs in an imperative byte code language. The machine-generated invariants can be pruned wrt. a particular safety policy to yield compact program certificates. To simplify the checking of these certificates, we have devised a result checker for polyhedra which uses inclusion certificates (issued from a result due to Farkas) instead of computing convex hulls of polyhedra at join points. This checker is much simpler to prove correct mechanically than the polyhedral analyser and provides a means of building a foundational proof carrying code that can make use of industrial strength relational program analysis.

Future work concerns extensions to incorporate richer domains of properties such as disjunctive completion of polyhedra or non-linear (polynomial) invariants. The certificate format and the result checker can accommodate the disjunctive completions, the inclusion certificates from Section 8.2 can be generalised to deal with non-linear inequalities as well [5]. However, the analyses for *inferring* such properties are in their infancy. On a language level, the challenge is to extend the analysis to cover the object oriented aspects of Java byte code. The inclusion of static fields and arrays in our framework provides a first step in that direction but a full extension would notably require an additional analysis to keep track of aliases between objects.

A promising domain of application for our relational analysis technique is to verify the dynamic allocation and consumption of resources and in particular to ensure statically that a program always acquires a necessary amount of resources before consuming them. The approach of Chander *et al.* [10] relies on the programmer to provide loop invariants and pre- and post-conditions for methods in order to link program variables to the amount of resources available and perform powerful transformations such as hoisting resource allocations out of loops. Our inter-procedural byte code analyser could infer the necessary invariants and pre-/post-conditions and in the same vein provide the checker for integrating this into a mobile code resource certification scheme.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-carrying code. In *Proc. of 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, Springer LNAI vol. 3452, pages 380–397, 2004.

[2] B. De Backer and H. Beringer. A clp language handling disjunctions of linear constraints. In *Proc. of the 10th Int. Conf. on Logic Programming (ICLP'93)*, pages 550–563. MIT Press, 1993.

[3] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Proc. of 10th Int. Static Analysis Symposium*, pages 337–354. Springer LNCS vol. 2694, 2003.

[4] R. Bagnara, P.M Hill, and E. Zaffanella. The Parma polyhedral library user's manual, 2006.

[5] F. Besson. Fast reflexive arithmetic tactics: the linear case and beyond. In *Types for Proofs and Programs (TYPES'06)*, pages 48–62. Springer LNCS vol. 4502, 2006.

[6] F. Besson, T. Jensen, and D. Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theoretical Computer Science*, 364(3):273–291, 2006.

[7] F. Besson, T. Jensen, and T. Turpin. Small witnesses for abstract interpretation based proofs. In *Proc. of 16th European Symp. on Programming (ESOP 2007)*, pages 268 – 283. Springer LNCS vol. 4421, 2007.

[8] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications*, pages 128–141. Springer LNCS vol. 735, 1993.

[9] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In *Proc. of 13th European Symp. on Programming (ESOP'04)*, pages 385–400. Springer LNCS vol. 2986, 2004.

[10] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proc. of the 14th European Symposium on Programming (ESOP 2005)*, pages 311–325. Springer LNCS vol. 3444, 2005.

[11] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *U.S.S.R Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[13] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of 5th ACM Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.

[14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. of the ACM Conf. on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.

[15] B. Jeannet and the Apron team. The Apron library, 2007.

[16] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ml. In *Proc. of 34th ACM Symp. on Principles of Programming Languages (POPL'07)*, pages 173–184. ACM Press, 2007.

[17] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of the 33rd ACM Symp. on Principles of programming languages (POPL'06)*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[18] A. Miné. The octagon abstract domain. In *Proc. of Working Conf. on Reverse Engineering 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[19] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI'06*, volume 3855 of *LNCS*, pages 348–363. Springer, 2002.

[20] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. of 31st ACM Symp. on Principles of Programming Languages (POPL'04)*, pages 330–341. ACM Press, 2004.

[21] David Pichardie. *Interprétation abstraite en logique intuitioniste: extraction d'analyseurs Java certifiés.* PhD thesis, Université de Rennes 1, Sept. 2005.

[22] E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.

[23] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1998.

[24] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of 1st Workshop on Bytecode Semantics, Verification and Transformation, Electronic Notes in Computer Science*, 2005.

[25] M. Wildmoser and T. Nipkow. Asserting bytecode safety. In *Proc. of the 15th European Symp. on Programming (ESOP'05)*, 2005.

[26] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In *Exploring New Frontiers of Theoretical Informatics, TC1 3rd Int. Conf. on Theoretical Computer Science (TCS2004)*, pages 333–347. Kluwer, 2004.

[27] Hongwei Xi. Imperative Programming with Dependent Types. In *Proc. of 15th IEEE Symposium on Logic in Computer Science (LICS'00)*, pages 375–387. IEEE, 2000.

[28] Hongwei Xi and Songtao Xia. Towards Array Bound Check Elimination in Java Virtual Machine Language. In *Proc. of CASCOON '99*, pages 110–125, 1999.

[29] H. Yang, S. Seo, K. Yi, and T. Han. Goal-directed weakening of abstract interpretation results. Submitted for publication.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399

Certification using the Mobius Base Logic

Lennart Beringer[1], Martin Hofmann[1], and Mariela Pavlova[2]

[1] Institut für Informatik, Universität München
Oettingenstrasse 67, 80538 München, Germany
[2] Trusted Labs, Sophia-Antipolis, France
{beringer|mhofmann}@tcs.ifi.lmu.de,Mariela.Pavlova@trusted-labs.fr

Abstract. This paper describes a core component of Mobius' Trusted Code Base, the Mobius base logic. This program logic facilitates the transmission of certificates that are generated using logic- and type-based techniques and is formally justified w.r.t. the Bicolano operational model of the JVM. The paper motivates major design decisions, presents core proof rules, describes an extension for verifying intensional code properties, and considers applications concerning security policies for resource consumption and resource access.

1 Introduction: Role of the logic in Mobius

The goal of the Mobius project consists of the development of proof-carrying code (PCC) technology for the certification of resource-related and information-security-related program properties [16]. According to the PCC paradigm, code consumers are invited to specify conditions ("policies") which they require transmitted code to satisfy before they are willing to execute such code. Providers of programs then complement their code with formal evidence demonstrating that the program adheres to such policies. Finally, the recipient validates that the obtained evidence ("certificate") indeed applies to the transmitted program and is appropriate for the policy in question before executing the code.

One of the cornerstones of a PCC architecture is the trusted computing base (TCB), i.e. the collection of notions and tools in whose correctness the recipient implicitly trusts. Typically, the TCB consists of a formal model of program execution, plus parsing and transformation programs that translate policies and certificates into statements over these program executions. The Mobius architecture applies a variant of the *foundational* PCC approach [2] where large extents of the TCB are represented in a theorem prover, for the following reasons.

- Formalising a (e.g. operational) semantics of transmitted programs in a theorem prover provides a precise definition of the model of program execution, making explicit the underlying assumptions regarding arithmetic and logic
- The meaning of policies may be made precise by giving formal interpretations in terms of the operational model
- Theorem provers offer various means to define formal notions of certificates, ranging from proof scripts formulated in the user interface language (including tactics) of the theorem prover to terms in the prover's internal representation language for proofs (e.g. lambda-terms).

In particular, the third item allows one to employ a variety of certificate notions in a uniform framework, and to explore their suitability for different certificate generation techniques or families of policies. In contrast to earlier PCC systems which targeted mostly type- and memory-safety [27, 2], policies and specifications in Mobius are more expressive, ranging from (upper) bounds on resource consumption, via access regulations for external resources and security specifications limiting the flow of information to lightweight functional specifications [16]. Thus, the Mobius TCB is required to support program analysis frameworks such as type systems and abstract interpretation, but also logical reasoning techniques.

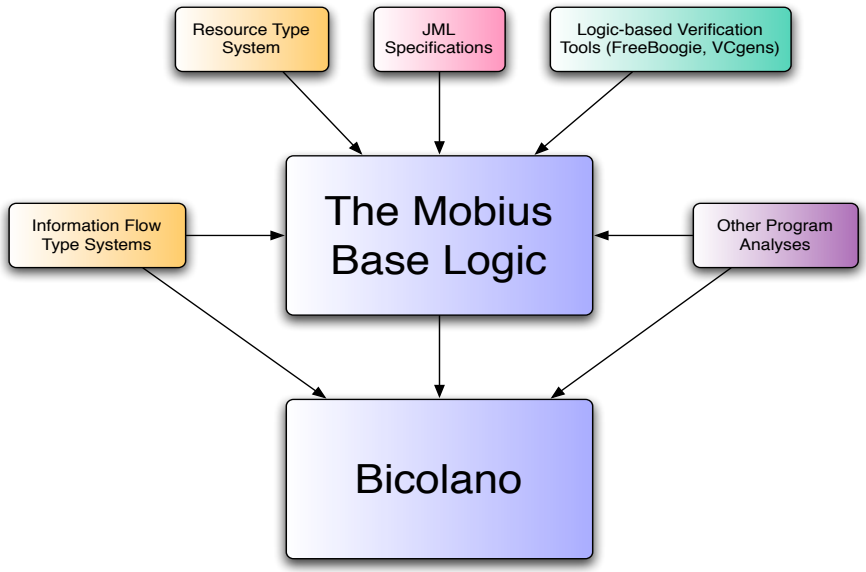


Fig. 1. Core components of the MOBIUS TCB

Figure 1 depicts the components of the Mobius TCB and their relations. The base of the TCB is formed by a formalised operational model of the Java Virtual Machine, Bicolano [30], which will be briefly described in the next section. Its purpose is to define the meaning of JVML programs unambiguously and to serve as the foundation on which the PCC framework is built. In order to abstract from inessential details, a program logic is defined on top of Bicolano. This provides support for commonly used verification patterns such as the verification of loops. Motivated by verification idioms used in higher-level formalisms such as type systems, the JML specification language, and verification condition generators, the logic complements partial-correctness style specifications by two further assertion forms: *local annotations* are attached to individual program points and

are guaranteed to hold whenever the annotated program point is visited during a program execution. *Strong invariants* assert that a particular property will continue to hold for all future states during the execution of a method, including states inside inner method invocations. The precise interpretation of these assertion forms, and a selection of proof rules will be described in Section 3.

We also present an extension of the program logic that supports reasoning about the effects of computations. The extended logic arises uniformly from a corresponding generic extension of the operational semantics. Using different instantiations of this framework one may obtain domain-specific logics for reasoning about access to external resources, trace properties, or the consumption of resources. Polices for such domains are difficult if not impossible to express purely in terms of relations between initial and final states. The extension is horizontal in the sense of Czarnik and Schubert [20] as it is conservative over the non-extended ("base") architecture.

The glue between the components is provided by the theorem prover Coq, i.e. many of the soundness proofs have been formalised. The encoding of the program logics follow the approach advocated by Kleymann and Nipkow [25, 29] by employing a shallow embedding of formulae. Assertions may thus be arbitrary Coq-definable predicates over states. Although the logic admits the encoding of a variety of program analyses and specification constructs, it should be noted that the architecture does not mandate that all analyses be justified with respect to this logic. Indeed, some type systems for information flow, for example, are most naturally expressed directly in terms of the operational semantics, as already the definition of information flow security is a statement over two program executions. In neither case do we need to construct proofs for concrete programs by hand which would be a daunting task in all but the simplest examples. Such proofs are always obtained from a successful run of a type system or program analysis by an automatic translation into the Mobius infrastructure. Examples of this method are given in Sections 4 and 5.2.

Outline We give a high-level summary of the operational model Bicolano [30], restricted to a subset of instructions relevant for the present paper, in Section 2. In Section 3 we present the program logic. Section 4 contains an example of a type-based verification and shows how a bytecode-level type system guaranteeing a constant upper bound on the number of heap allocations may be encoded in the logic. The extended program logic is outlined in Section 5, together with an application concerning a type system for numeric correspondence assertions [34]. We first discuss some related work.

1.1 Related work

The basic design decisions for the base logic were presented in [8], and the reader is referred to loc.cit. for a more in-depth motivation of the chosen format of assertions and rules. In that paper, we also presented a type-system for constant heap space consumption for a functional intermediate language, such that typing

derivations could be translated into program logic derivations over an appropriately restricted judgement form. In contrast, the type system given in the present paper works directly on bytecode and hence eliminates the language translation from the formalised TCB.

The first proposal for a program logic for bytecode we are aware of is the one by Quigley [31]. In order to justify a rule for while loops, Quigley introduces various auxiliary notions for relating initial states to intermediate states of an execution sequence, and for relating states that behave similarly but apply to different classes. Bannwart and Müller [4] present a logic where assertions apply at intermediate states and are interpreted as preconditions of the assertions decorating the successor instructions. However, the occurrence of these local specifications in positive and negative positions in this interpretation precludes the possibility of introducing a rule of consequence. Indeed, our proposed rule format arose originally from an attempt to extend Bannwart and Müller's logic with a rule of consequence and machinery for allowing assertions to mention initial states. Strong invariants were introduced by the Key project [6] for reasoning about transactional safety of Java Card applications using dynamic logics [7].

Regarding formal encodings of type systems into program logics, Hähnle et al. [23], and Beringer and Hofmann [9] consider the task of representing information flow type systems in program logics, while the MRG project focused on a formalising a complex type system for input-dependent heap space usage [10].

Certified abstract interpretation [11] complements the type-based certificate generation route considered in the present paper. Similar to the relationship between Necula-Lee-style PCC [27] and foundational PCC by Appel et al. [2], certified abstract interpretation may be seen as a foundational counterpart to Albert et al.'s Abstraction-carrying code [1]. Bypassing the program logic, the approach chosen in [11] justifies the program analysis directly with respect to the operational semantics. A generic framework for certifying program analyses based on abstract interpretation is presented by Chang et al. [14]. The possibility to view abstract interpretation frameworks as inference engines for invariants and other assertions in program logics in general was already advocated in one of the classic papers by Cousot & Cousot in [18].

Nipkow et al.'s VeryPCC project [33] explores an alternative foundational approach by formally proving the soundness of verification condition generators. In particular, [32] presents generic soundness and completeness proofs for VCGens, together with an instantiation of the framework to a safety policy preventing arithmetic overflows. Generic PCC architectures have recently been developed by Necula et al. [15] and the FLINT group [22].

2 Bicolano

Syntax and States We consider an arbitrary but fixed bytecode program P that assigns to each method identifier M a method implementation mapping instruction labels l to instructions. We use the notation $M(l)$ to denote the instruction at program point l in M, and $init_M$, $suc_M(l)$, and par_M to denote the initial

label of M, the successor label of l in M, and the list of formal parameters of M, respectively. While the Bicolano formalisation supports the full sequential fragment of the JVML, this paper treats the simplified language given by the *basic* instructions

$$basic(M,l) \equiv M(l) \in \left\{ \begin{array}{l} \mathsf{load}\ x, \mathsf{store}\ x, \mathsf{dup}, \mathsf{pop}, \mathsf{push}\ z, \\ \mathsf{unop}\ \mathsf{u}, \mathsf{binop}\ o, \mathsf{new}\ c, \mathsf{athrow}, \\ \mathsf{getfield}\ c\ f, \mathsf{putfield}\ c\ f, \mathsf{getstatic}\ c\ f, \mathsf{putstatic}\ c\ f \end{array} \right\}$$

and additionally conditional and unconditional jumps $\mathsf{ifz}\ l$ and $\mathsf{goto}\ l$, static and virtual method invocations $\mathsf{invokestatic}\ M$ and $\mathsf{invokevirtual}\ M$, and $\mathsf{vreturn}$.

Values and states The domain $\mathcal{V}$ of values is ranged over by $v, w, \ldots$ and comprises constants (integers z and Null), and addresses $a, \ldots \in \mathcal{A}$. States are built from operand stacks, stores, and heaps

$$O \in \mathcal{O} = \mathcal{V}\ list \qquad S \in \mathcal{S} = \mathcal{X} \rightharpoonup_{fin} \mathcal{V} \qquad h \in \mathcal{H} = \mathcal{A} \rightharpoonup_{fin} \mathcal{C} \times (\mathcal{F} \rightharpoonup_{fin} \mathcal{V})$$

where $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{F}$ are the domains of variables, class names, and field names, respectively. In addition to local states comprising operand stacks, stores, and heaps,

$$s, r \in \Sigma = \mathcal{O} \times \mathcal{S} \times \mathcal{H},$$

we consider initial states Σ_0 and terminal states $\mathcal{T}$

$$s_0 \in \Sigma_0 = \mathcal{S} \times \mathcal{H} \qquad t \in \mathcal{T} ::= \mathit{NormState}(h, v) + \mathit{ExcnState}(h, a)$$

These capture states which occur at the beginning and the end of a frame's execution. Terminal states t are tagged according to whether the return value represents a pointer to an unhandled exception object (constructor $\mathit{ExcnState}(.,.)$) or an ordinary return value (constructor $\mathit{NormState}(.,.)$). For $s_0 = (S, h)$ we write $\mathit{state}(s_0) = ([\,], S, h)$ for the local state that extends s_0 with an empty operand stack. For $\mathit{par}_M = [x_1, \ldots, x_n]$ and $O = [v_1, \ldots, v_n]$ we write $\mathit{par}_M \mapsto O$ for $[x_i \mapsto v_i]_{i=1,\ldots,n}$. We write $\mathit{heap}(s)$ to access the heap component of a state s, and similarly for initial and terminal states. Finally, $\mathit{lv}(.)$ denotes the local variable component of a state and $\mathit{getClass}(h, a)$ extracts the dynamic class of the object at location a in heap h.

Operational judgements Bicolano defines a variety of small-step and big-step judgements, with compatibility proofs where appropriate. For the purpose of the present paper, the following simplified setup suffices[3] (cf. Figure 2):

Non-exceptional steps The judgement $\vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r$ describes the (non-exceptional) execution of a single instruction, where l' is the label of the next instruction (given by $\mathit{suc}_M(l)$ or jump targets). The rules are largely standard, so we only give a rule for the invocation of static methods, InvsNorm.

[3] The formalisation separates the small-step judgements for method invocations from the execution of basic instructions and jumps, and then defines a single recursive judgement combining the two. See [30] for the formal details.

Exceptional steps The judgement $\vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a$ describes exceptional small steps where the execution of the instruction at program point M, l in state s results in the creation of a fresh exception object, located at address a in the heap h. In the case of method invocations, a single exceptional step is also observed by the callee if the invoked method raised an exception that could not be locally handled (cf. rule InvsExcn).

Small step judgements Non-exceptional and handled exceptional small steps are combined to the small step judgement $\vdash_M l, s \Rightarrow l', r$ using the two rules NormStep and ExcnStep. The reflexive transitive closure of this relation is denoted by $\vdash_M l, s \Rightarrow^* l', r$

Big-step judgements The judgement form $\vdash_M l, s \Downarrow t$ captures the execution of method M from the instruction at label l onwards, until the end of the method. This relation is defined by the three rules Comp, Vret and Uncaught.

Deep step judgements The judgement $\vdash_M l, s \Uparrow r$ is defined similarly to the big-step judgement, by the rules D-Refl, D-Trans D-Invs, and D-Uncaught. This judgement associates states across invocation boundaries, i.e. r may occur in a subframe of the method M. This is achieved by rule D-Invs which associates a call state of a (static) method with states reachable from the initial state of the callee. A similar rule for virtual methods is omitted from this presentation.

Small and big-step judgements are mutually recursive due to the occurrence of a big-step judgement in hypotheses of the rules for method invocations on the one hand and rule Comp on the other.

3 Base logic

This section outlines the non-resource-extended program logic.

3.1 Phrase-oriented assertions and judgements

The structure of assertions and judgements of the logic are governed by the requirement to enable the interpretation of type systems as well as the representation of core idioms of JML. High-level type systems typically associate types (in contexts) to program phrases. Compiling a well-formed program phrase into bytecode yields a code segment that is the postfix of a JVM method, i.e. all program points without control flow successors contain return instructions. Consequently, judgements in the logic associate assertions to a program label which represents the execution of the current method invocation from the current point (i.e. a state applicable at the program point) onwards. In case of method termination, a partial-correctness assertion (post-condition) applies that relates this current state to the return state. As the guarantee given by type soundness results often extends to infinite computations (e.g. type safety, i.e. absence of type errors), judgements furthermore include assertions that apply to non-terminating

$$\textsc{InvsNorm}\ \frac{M(l) = \mathsf{invokestatic}\ M' \quad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Downarrow NormState(k, v)}{\vdash_M l, (O@O', S, h) \Rightarrow_{\mathsf{norm}} suc_M(l), (v :: O', S, k)}$$

$$\textsc{InvsExcn}\ \frac{M(l) = \mathsf{invokestatic}\ M' \quad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Downarrow ExcnState(k, a)}{\vdash_M l, (O@O', S, h) \Rightarrow_{\mathsf{excn}} k, a}$$

$$\textsc{NormStep}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r}{\vdash_M l, s \Rightarrow l', r} \qquad \textsc{ExcnStep}\ \frac{\vdash_M l, (O, S, h) \Rightarrow_{\mathsf{excn}} k, a \quad getClass(k, a) = e \quad Handler(M, l, e) = l'}{\vdash_M l, (O, S, h) \Rightarrow l', ([a], S, k)}$$

$$\textsc{Comp}\ \frac{\vdash_M l, s \Rightarrow l', s' \quad \vdash_M l', s' \Downarrow t}{\vdash_M l, s \Downarrow t} \qquad \textsc{Vret}\ \frac{M(l) = \mathsf{vreturn}}{\vdash_M l, (v :: O, S, h) \Downarrow NormState(h, v)}$$

$$\textsc{Uncaught}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a \quad getClass(h, a) = e \quad Handler(M, l, e) = \emptyset}{\vdash_M l, s \Downarrow ExcnState(h, a)}$$

$$\textsc{D-Refl}\ \frac{}{\vdash_M l, s \Uparrow s} \qquad \textsc{D-Trans}\ \frac{\vdash_M l, s \Rightarrow l', s' \quad \vdash_M l', s' \Uparrow s''}{\vdash_M l, s \Uparrow s''}$$

$$\textsc{D-Invs}\ \frac{M(l) = \mathsf{invokestatic}\ M' \quad \vdash_{M'} init_{M'}, ([\,], par_{M'} \mapsto O, h) \Uparrow s}{\vdash_M l, (O@O', S, h) \Uparrow s}$$

$$\textsc{D-Uncaught}\ \frac{\vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a \quad getClass(h, a) = e \quad Handler(M, l, e) = \emptyset}{\vdash_M l, s \Uparrow ([a], \emptyset, h)}$$

Fig. 2. Bicolano: selected judgements and operational rules

computations. These *strong invariants* relate the state valid at the subject label to each future state in the current method invocation. This interpretation includes states in subframes, i.e. in method invocations that are triggered in the phrase represented by the subject label.

Infinite computations are also covered by the interpretation of local annotations in JML, i.e. assertions occurring at arbitrary program points which are to be satisfied whenever the program point is visited. The logic distinguishes these explicitly given annotation from strong invariants as the former ones are not necessarily present at all program points. A further specification idiom of JML that has a direct impact on the form of assertions is `\old` which refers to the initial state of a method invocation and may appear in post-conditions, local annotations, and strong invariants.

Formulae that are shared between postconditions, local annotations, and strong invariant, and additionally only concern the relationship between the subject state and the initial state of the method may be captured in pre-conditions.

Thus, the judgement of the logic are of the form $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ where M, l denotes a program point (composed of a method identifier and an instruc-

tion label), and the assertions forms are as follows, where $\mathcal{B}$ denotes the set of booleans.

Assertions $A \in \mathit{Assn} = \Sigma_0 \times \Sigma \to \mathcal{B}$ occur as preconditions A and local annotations Q, and relate the current state to the initial state of the current frame.

Postconditions $B \in \mathit{Post} = \Sigma_0 \times \Sigma \times \mathcal{T} \to \mathcal{B}$ relate the current state to the initial and final state of a (terminating) execution of the current frame.

Invariants $I \in \mathit{Inv} = \Sigma_0 \times \Sigma \times \Sigma \to \mathcal{B}$ relate the initial state of the current method, the current state, and any future state of the current frame or a subframe of it.

The component $\mathcal{G}$ of a judgement represents a proof context and is represented as an association of specification triples $(A, B, I) \in \mathit{Assn} \times \mathit{Post} \times \mathit{Inv}$ to program points.

The behaviour of methods is described using three assertion forms.

Method preconditions $R \in \mathit{MethPre} = \Sigma_0 \to \mathcal{B}$ are interpreted hypothetically, i.e. their satisfaction implies that of the method postconditions and invariants but is not directly enforced to hold at all invocation points.

Method postconditions $T \in \mathit{MethSpec} = \Sigma_0 \times \mathcal{T} \to \mathcal{B}$ constrain the behaviour of terminating method executions and thus relate only initial and final states.

Method invariants $\Phi \in \mathit{MethInv} = \Sigma_0 \times \Sigma \to \mathcal{B}$ constrain the behaviour of terminating and non-terminating method executions by relating the initial state of a method frame to any state that occurs during its execution.

A program specification is given by a method specification table $\mathcal{M}$ that associates to each method a method specification $\mathcal{S} = (R, T, \Phi)$, a proof context $\mathcal{G}$, and a table $\mathcal{Q}$ of local annotations $Q \in \mathit{Assn}$. From now on, let $\mathcal{M}$ denote some arbitrary but fixed specification table satisfying $\mathit{dom}\ \mathcal{M} = \mathit{dom}\ P$.

3.2 Assertion transformers

In order to notationally simplify the presentation of the proof rules, we define operators that relate assertions occurring in judgements of adjacent instructions. The following operators apply to the non-exceptional single-step execution of basic instructions.

$$\begin{aligned}
\mathsf{Pre}(M, l, l', A)(s_0, r) &= \exists\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \wedge A(s_0, s) \\
\mathsf{Post}(M, l, l', B)(s_0, r, t) &= \forall\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \to B(s_0, s, t) \\
\mathsf{Inv}(M, l, l', I)(s_0, r, t) &= \forall\ s.\ \vdash_M l, s \Rightarrow_{\mathsf{norm}} l', r \to I(s_0, s, t)
\end{aligned}$$

These operators resemble WP-operators, but are separately defined for pre-conditions, post-conditions, and invariants.

Exceptional behaviour of basic instructions is captured by the operators

$$\begin{aligned}
\mathsf{Pre}^{\mathsf{excn}}(M,l,e,A)(s_0,r) &= \exists\, s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \wedge \mathit{getClass}(h,a) = e \wedge \\
&\qquad r = ([a], \mathit{lv}(s), h) \wedge A(s_0,s) \\
\mathsf{Post}^{\mathsf{excn}}(M,l,e,B)(s_0,r,t) &= \forall\, s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \rightarrow \mathit{getClass}(h,a) = e \rightarrow \\
&\qquad r = ([a], \mathit{lv}(s), h) \rightarrow B(s_0,s,t) \\
\mathsf{Inv}^{\mathsf{excn}}(M,l,e,I)(s_0,r,t) &= \forall\, s\ h\ a.\ \vdash_M l,s \Rightarrow_{\mathsf{excn}} h,a \rightarrow \mathit{getClass}(h,a) = e\ \rightarrow \\
&\qquad r = ([a], \mathit{lv}(s), h) \rightarrow I(s_0,s,t)
\end{aligned}$$

In the case of method invocations, we replace the reference to the operational judgement by a reference to the method specifications, and include the construction and destruction of a frame. For example, the operators for non-exceptional execution of static methods are

$$\begin{aligned}
&\mathsf{Pre}_{\mathsf{sinv}}(R,T,A,[x_1,\ldots,x_n])(s_0,s) = \\
&\quad \exists\, O\ S\ h\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n, h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n, h), (k,v))) \wedge \\
&\qquad\qquad s = (v :: O, S, k) \wedge A(s_0, ([v_1,\ldots,v_n]@O, S, h)) \\
&\mathsf{Post}_{\mathsf{sinv}}(R,T,B,[x_1,\ldots,x_n])(s_0,r,t) = \\
&\quad \forall\, O\ S\ k\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n, h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n, h), (k,v)))\ \rightarrow \\
&\qquad\qquad r = (v :: O, S, k) \rightarrow B(s_0, ([v_1,\ldots,v_n]@O, S, h), t) \\
&\mathsf{Inv}_{\mathsf{sinv}}(R,T,I,[x_1,\ldots,x_n])(s_0,s,r) = \\
&\quad \forall\, O\ S\ k\ k\ v\ v_i.\ (R([x_i \mapsto v_i]_{i=1}^n, h) \rightarrow T(([x_i \mapsto v_i]_{i=1}^n, h), (k,v)))\ \rightarrow \\
&\qquad\qquad s = (v :: O, S, k) \rightarrow I(s_0, ([v_1,\ldots,v_n]@O, S, h), r)
\end{aligned}$$

The exceptional operators for static methods cover exceptions that are raised during the execution of the invoked method but not handled locally. Due to space limitations we omit the operators for exceptional (null-pointer exceptions w.r.t. the invoking object) and non-exceptional behaviour of virtual methods.

3.3 Selected proof rules

An addition to influencing the types of assertions, type systems also motivate the use of a certain form of judgements and proof rules. Indeed, one of the advantages of type systems is their compositionality i.e. the fact that statements regarding a program phrase are composed from the statements referring to the constituent phrases, as in the following typical proof rule for a language of expressions

$$\frac{\vdash e_1 : \mathbf{int} \quad \vdash e_2 : \mathbf{int}}{\vdash e_1 + e_2 : \mathbf{int}}.$$

Transferring this scheme to bytecode leads to a rule format where hypothetical judgements refer to the control flow successors of the phrase in the judgement's conclusion. In addition to supporting syntax-directed reasoning, this orientation renders the explicit construction of a control flow graph unnecessary, as no control flow predecessor information is required to perform a proof.

Figure 3 presents selected proof rules. These are motivated as follows.

$$\textsc{Instr}\ \frac{\begin{array}{c} basic(M,l) \qquad SC_1 \qquad SC_2 \qquad l'' = suc_M(l) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l'',A)\}\, M, l'\, \{\mathsf{Post}(M,l,l'',B)\}\, (\mathsf{Inv}(M,l,l'',I)) \\ \forall\, l'\ e.\ Handler(M,l,e) = l'\ \rightarrow \\ \mathcal{G} \vdash \{\mathsf{Pre}^{\mathsf{excn}}(M,l,e,A)\}\, M, l'\, \{\mathsf{Post}^{\mathsf{excn}}(M,l,e,B)\}\, (\mathsf{Inv}^{\mathsf{excn}}(M,l,e,I)) \\ \forall\, s_0\ s\ h\ a.\ (\forall\, e.\ getClass(h,a) = e\ \rightarrow Handler(M,l,e) = \emptyset)\ \rightarrow \\ \vdash_M l, s \Rightarrow_{\mathsf{excn}} h, a \rightarrow A(s_0,s) \rightarrow B(s_0,s,(h,a)) \end{array}}{\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\textsc{Goto}\ \frac{\begin{array}{c} M(l) = \mathsf{Goto}\ l' \qquad SC_1 \qquad SC_2 \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l',A)\}\, M, l'\, \{\mathsf{Post}(M,l,l',B)\}\, (\mathsf{Inv}(M,l,l',I)) \end{array}}{\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\textsc{If0}\ \frac{\begin{array}{c} M(l) = \mathsf{ifz}\ l' \qquad SC_1 \qquad SC_2 \qquad l'' = suc_M(l) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l',A)\}\, M, l'\, \{\mathsf{Post}(M,l,l',B)\}\, (\mathsf{Inv}(M,l,l',I)) \\ \mathcal{G} \vdash \{\mathsf{Pre}(M,l,l'',A)\}\, M, suc_M(l)\, \{\mathsf{Post}(M,l,l'',B)\}\, (\mathsf{Inv}(M,l,l'',I)) \end{array}}{\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\textsc{InvS}\ \frac{\begin{array}{c} M(l) = \mathsf{invokestatic}\ M' \quad \mathcal{M}(M') = (R,T,\Phi) \quad SC_1 \quad SC_2 \\ \forall\, s_0\ O\ S\ h\ O'\ r\ v_i.\ (R(par_{M'} \mapsto O, h) \rightarrow \Phi((par_{M'} \mapsto O, h), r))\ \rightarrow \\ A(s_0, (O@O', S, h)) \rightarrow I(s_0, (O@O', S, h), r) \\ A_1 = \mathsf{Pre}_{\mathsf{sinv}}(R,T,A,par_{M'}) \qquad B_1 = \mathsf{Post}_{\mathsf{sinv}}(R,T,B,par_{M'}) \\ \mathcal{G} \vdash \{A_1\}\, M, suc_M(l)\, \{B_1\}\, (\mathsf{Inv}_{\mathsf{sinv}}(R,T,I,par_{M'})) \\ \forall\, l'\ e.\ Handler(M,l,e) = l'\ \rightarrow \\ \mathcal{G} \vdash \{\mathsf{Pre}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,A,e,par_{M'})\}\, M, l'\, \{\mathsf{Post}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,B,e,par_{M'})\} \\ \qquad\qquad (\mathsf{Inv}^{\mathsf{excn}}_{\mathsf{sinv}}(R,T,I,e,par_{M'})) \\ \forall\, s_0\ O\ S\ h\ O'\ k\ a.\ (R(par_{M'} \mapsto O, h) \rightarrow \Phi((par_{M'} \mapsto O, h), (k,a)))\ \rightarrow \\ (\forall\, e.\ getClass(k,a) = e\ \rightarrow Handler(M,l,e) = \emptyset)\ \rightarrow \\ A(s_0, (O@O', S, h)) \rightarrow B(s_0, (O@O', S, h), (k,a)) \end{array}}{\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\textsc{Ret}\ \frac{\begin{array}{c} M(l) = \mathsf{vreturn} \qquad SC_1 \qquad SC_2 \\ \forall\, s_0\ v\ O\ S\ h.\ A(s_0, (v :: O, S, h)) \rightarrow B(s_0, (v :: O, S, h), (h, v)) \end{array}}{\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

$$\textsc{Conseq}\ \frac{\begin{array}{ll} \mathcal{G} \vdash \{A'\}\, \ell\, \{B'\}\, (I') & \forall\, s_0\ s.\ A(s_0,s) \rightarrow A'(s_0,s) \\ \forall\, s_0\ s\ t.\ B'(s_0,s,t) \rightarrow B(s_0,s,t) & \forall\, s_0\ s\ r.\ I'(s_0,s,r) \rightarrow I(s_0,s,r) \end{array}}{\mathcal{G} \vdash \{A\}\, \ell\, \{B\}\, (I)}$$

$$\textsc{Ax}\ \frac{\begin{array}{c} \mathcal{G}(\ell) = (A,B,I) \qquad \forall\, s_0\ s.\ A(s_0,s) \rightarrow I(s_0,s,s) \\ \forall\, Q.\ \mathcal{Q}(\ell) = Q \rightarrow (\forall\, s_0\, s.\ A(s_0,s) \rightarrow Q(s_0,s)) \end{array}}{\mathcal{G} \vdash \{A\}\, \ell\, \{B\}\, (I)}$$

Fig. 3. Program logic: selected syntax-directed rules

Rule INSTR describes the behaviour of basic instructions. The hypothetical judgement for the successor instruction involves assertions that are related to the assertions in the conclusion by the transformers for normal termination. A further hypothesis captures exceptions that are handled locally, i.e. those exceptions e to which the exception handler of the current method associates a handling instruction (predicate $\mathit{Handler}(M, l, e) = l'$). Exceptions that are not handled locally result in abrupt termination of the method. Consequently, these exceptions are modelled in a side condition that involves the method postcondition rather than a further judgemental hypothesis.

Finally, the side conditions SC_1 and SC_2 ensure that the invariant I and the local annotation Q (if existing) are satisfied in any state reaching label l.

$$SC_1 = \forall\, s_0\ s.\ A(s_0, s) \to I(s_0, s, s)$$
$$SC_2 = \forall\, Q.\ \mathcal{Q}(M, l) = Q \to (\forall\, s_0\, s.\ A(s_0, s) \to Q(s_0, s))$$

In particular, SC_2 requires us to prove any annotation that is associated with label l. Satisfaction of I in later states, and satisfaction of local annotations Q' of later program points are guaranteed by the judgement for $\mathit{suc}_M(l)$.

The rules for conditional and unconditional jumps include a hypotheses for the control flow successors, and the same side conditions for local annotations and invariants as rule Instr. No further hypotheses or side conditions regarding exceptional behaviour are required as these instructions do not raise exceptions. These rules also account for the verification of loops which on the level of bytecode are rendered as jumps. Loop invariants can be inserted as postconditions B at their program point. Rule Ax allows one to use such invariants whereas according to Definition 1 they must be established once in order for a verification to be valid.

In rule InvS, the invariant of the callee, namely Φ (more precisely: the satisfaction of Φ whenever the initial state of the callee satisfies the precondition R), and the local precondition A may be exploited to establish the invariant I. This ensures that I will be satisfied by all states that arise during the execution of M', as these states will always conform to Φ. The callee's post-condition T is used to construct the assertions that occur in the judgement for the successor instruction l'. Both conditions reflect the transfer of the method arguments and return values between the caller and the callee. This protocol is repeated in the hypothesis and the side condition for the exceptional cases which otherwise follow the pattern mentioned in the description of the rule Instr.

A similar rule for virtual methods is omitted. The rule for method returns, Ret, ties the precondition A to the post-condition B w.r.t. the terminal state that is constructed using the topmost value of the operand stack.

Finally, the *logical rules* Conseq and Ax arise from the standard rules by adding suitable side conditions for strong invariants and local assertions.

3.4 Behavioural subtyping and verified programs

We say that method specification (R, T, Φ) *implies* (R', T', Φ') if

– for all s_0 and t, $R(s_0) \to T(s_0, t)$ implies $R'(s_0) \to T'(s_0, t)$, and
– for all s_0 and s, $R(s_0) \to \Phi(s_0, s)$ implies $R'(s_0) \to \Phi'(s_0, s)$

Furthermore, we say that $\mathcal{M}$ satisfies *behavioural subtyping* for P if whenever P contains an instruction invokevirtual M' with $\mathcal{M}(M') = (\mathcal{S}', \mathcal{G}', \mathcal{Q}')$, and M overrides M', then there are $\mathcal{S}$, $\mathcal{G}$ and $\mathcal{Q}$ with $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$ such that $\mathcal{S}$ implies $\mathcal{S}'$. Finally, we call a derivation $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ *progressive* if it contains at least one application of a non-logical rule.

Definition 1. *P is* verified *with respect to $\mathcal{M}$, notation $\mathcal{M} \vdash P$, if*

– *$\mathcal{M}$ satisfies behavioural subtyping for P, and*
– *for all M, $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$, and $\mathcal{S} = (R, T, \Phi)$*
  - *a progressive derivation $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ exists for any l, A, B, and I with $\mathcal{G}(M, l) = (A, B, I)$, and*
  - *a progressive derivation $\mathcal{G} \vdash \{A\}\, M, \mathit{init}_M\, \{B\}\, (I)$ exists for*

$$A(s_0, s) \equiv s = \mathit{state}(s_0) \wedge R(s_0)$$
$$B(s_0, s, t) \equiv s = \mathit{state}(s_0) \to T(s_0, t)$$
$$I(s_0, s, r) \equiv s = \mathit{state}(s_0) \to \Phi(s_0, r).$$

As the reader may have noticed, behavioural subtyping only affects method specifications but not the proof contexts $\mathcal{G}$ or annotation tables $\mathcal{Q}$. Technically, the reason for this is that no constraints on these components are required in order to prove the logic sound. Pragmatically, we argue that proof contexts and local annotations tables of overriding methods indeed should not be related to contexts and annotation tables of their overridden counterparts, as both kinds of tables expose the internal structure of method implementations. In particular, entries in proof contexts and annotation tables are formulated w.r.t. specific program points, which would be difficult to interprete outside the method boundary or indeed across different (overriding) implementations of a method.

The distinction between progressive and non-progressive derivations prevents attempts to justify a proof context or method specification table simply by applying the axiom rule to all entries. In program logics for high-level languages, the corresponding effect is silently achieved by the unfolding of the method body in the rule for method invocations [29]. As our judgemental form does not permit such an unfolding, the auxiliary notion of progressive derivations is introduced. In our formalisation, the separation between progressive and other derivations is achieved by the introduction of a second judgement form, as described in [8].

3.5 Interpretation and soundness

Definition 2. *The triple $(\mathcal{Q}, B, I)$ is* valid at (M, l) for (s_0, s) *if*

– *for all r, if $\vdash_M l, s \Downarrow t$ then $B(s_0, s, t)$*
– *for all l' and r, if $\vdash_M l, s \Rightarrow^* l', r$ and $\mathcal{Q}(l') = Q$, then $Q(s_0, r)$, and*
– *for all r, if $\vdash_M l, s \Uparrow r$ then $I(s_0, s, r)$.*

Note that the second clause applies to annotations Q associated with arbitrary labels l' in method M that will be visited during the execution of M from (l, s) onwards. Although these annotations are interpreted without recourse to the state s, the proof of $Q(s_0, r)$ may exploit the precondition $A(s_0, s)$.

The soundness result is then as follows.

Theorem 1. *For $\mathcal{M} \vdash P$ let $\mathcal{M}(M) = (\mathcal{S}, \mathcal{G}, \mathcal{Q})$, $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ be a progressive derivation, and $A(s_0, s)$. Then $(\mathcal{Q}, B, I)$ is valid at (M, l) for (s_0, s).*

In particular, this theorem implies that for $\mathcal{M} \vdash P$ all method specifications in $\mathcal{M}$ are honoured by their method implementations. The proof of this result may be performed in two ways. Following the approach of Kleymann and Nipkow [25, 29, 3], one would first prove that the derivability of a judgement entails its validity, under the hypothesis that contextual judgements have already been validated. For this task, the standard technique involves the introduction of relativised notions of validity that restrict the interpretation of judgements to operational judgements of bounded height. Then, the hypothesis on contextual judgements is eliminated using structural properties of the relativised validity. An alternative to this approach has been developed by Benjamin Gregoire in the course of the formalisation of the present logic. It consists of (i) defining a family of syntax-directed judgements (one judgement form for each instruction form, inlining the rule of consequence), (ii) proving that property $\mathcal{M} \vdash P$ implies that the last step in a derivation of $\mathcal{G} \vdash \{A\}\, M, l\, \{B\}\, (I)$ can be replaced by an application of the syntax-directed judgement corresponding to the instruction at M, l (in particular, an application of the axiom rule is replaced by the derivation for the corresponding code blocks from $\mathcal{G}$), and (iii) proving the main claim of Theorem 1 by treating the three parts of Definition 2 separately, each one by induction over the respective operational judgement.

4 Type-based verification

In this section we present a type system that ensures a constant bound on the heap consumption of bytecode programs. The type system is formally justified by a soundness proof with respect to the MOBIUS base logic, and may serve as the target formalism for type-transforming compilers.

The requirement imposed on programs is similar to that of the analysis presented by Cachera et al. in [13] in that recursive program structures are denied the facility to allocate memory. However, our analysis is presented as a type system while the analysis presented in [13] is phrased as an abstract interpretation. In addition, Cachera et al.'s approach involves the formalisation of the calculation of the program representation (control flow graph) and of the inference algorithm (fixed point iteration) in the theorem prover. In contrast, our presentation separates the algorithmic issues (type inference and checking) from semantic issues (the property expressed or guaranteed) as is typical for a type-based formulation. Depending on the verification infrastructure available at the code consumer side, the PCC certificate may either consist of (a digest of) the

typing derivation or an expansion of the interpretation of the typing judgements into the MOBIUS logic. The latter approach was employed in our earlier work [10] and consists of understanding typing judgements as derived proof rules in the program logic and using syntax-directed proof tactics to apply the rules in an automatic fashion. In contrast to [10], however, the interpretation given in the present section extends to non-terminating computations, albeit for a far simpler type system.

The present section extends the work presented in [8] as the type system is now phrased for bytecode rather than an intermediate functional language and includes the treatment of exceptions and virtual methods.

Bytecode-level type system The type system consists of judgements of the form $\vdash_{\Sigma,\Lambda} \ell : n$, expressing that the segment of bytecode whose initial instruction is located at ℓ is guaranteed not to allocate more than n memory cells. Here, ℓ denotes a program point M, l while signatures Σ and Λ assign types (natural numbers n) to identifiers of methods and bytecode instructions (in particular, when those are part of a loop), respectively.

$$\text{C-New}\ \frac{n \geq 1 \qquad M(l) = \mathsf{New}\ C \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n-1}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-Instr}\ \frac{\begin{array}{c} n \geq 1 \qquad basic(M,l) \qquad \neg M(l) = \mathsf{New}\ C \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \vdash_{\Sigma,\Lambda} M, l' : n-1 \end{array}}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-If}\ \frac{n \geq 0 \qquad M(l) = \mathsf{ifz}\ l' \qquad \vdash_{\Sigma,\Lambda} M, l' : n \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n}{\vdash_{\Sigma,\Lambda} M, l : n}$$

$$\text{C-Invoke}\ \frac{\begin{array}{c} M(l) \in \{\mathsf{invokestatic}\ M', \mathsf{invokevirtual}\ M'\} \qquad \Sigma(M') = k \\ n \geq 1 \qquad k \geq 0 \qquad \vdash_{\Sigma,\Lambda} M, suc_M(l) : n \\ \forall\, l'\ e.\ Handler(M,l,e) = l' \ \rightarrow \vdash_{\Sigma,\Lambda} M, l' : n-1 \end{array}}{\vdash_{\Sigma,\Lambda} M, l : n+k}$$

$$\text{C-Ret}\ \frac{M(l) = \mathsf{vreturn}}{\vdash_{\Sigma,\Lambda} M, l : 0} \qquad \text{C-Sub}\ \frac{\vdash_{\Sigma,\Lambda} \ell : n \qquad n \leq k}{\vdash_{\Sigma,\Lambda} \ell : k} \qquad \text{C-Assum}\ \frac{\Lambda(\ell) = n}{\vdash_{\Sigma,\Lambda} \ell : n}$$

Fig. 4. Type system for constant heap space

The rules are presented in Figure 4. The first rule, C-New, asserts that the memory consumption of a code fragment whose first instruction is $\mathsf{new}\ C$ is the increment of the remaining code. Rule C-Instr applies to all basic instructions (in the case of $\mathsf{goto}\ l'$ we take $suc_M(l)$ to be l'), except for $\mathsf{new}\ C$ – the predicate $basic(m, l)$ is defined as in Section 3.3. The memory effect of these instructions is zero, as is the case for return instructions, conditionals, and (static) method

invocations in the case of normal termination. For exceptional termination, the allocation of a fresh exception object is accounted for by decrementing the type for the code continuation by one unit. The rule C-Assum allows for using the annotation attached to the instruction if it matches the type of the instruction.

A typing derivation $\vdash_{\Sigma,\Lambda} \ell : k$ is called *progressive* if it does not solely contain applications of rules C-Sub and C-Assum. Furthermore, we call P *well-typed* for Σ, notation $\vdash_\Sigma P$, if for all M and n with $\Sigma(M) = n$ there is a local specification table Λ such that a progressive derivation $\vdash_{\Sigma,\Lambda} M, \mathit{init}_M : n$ exists, and for all ℓ with $\Lambda(\ell) = k$ we have a progressive derivation $\vdash_{\Sigma,\Lambda} \ell : k$.

Type checking and inference The tasks of checking and automatically finding (inference) of typing derivations are not our main concern here. Nevertheless, we discuss briefly how this can be achieved.

For this simple type system checking a given typing derivation amounts to verifying the inequations that arise as side conditions. Furthermore, given Σ, Λ a corresponding typing derivation can be reconstructed by applying the typing rules in a syntax-directed fashion. In order to construct Σ, Λ as well (type inference) one writes down a "skeleton derivation" with indeterminates instead of actual numeric values and then solves the arising system of linear inequalities. Alternatively, one can proceed by counting allocation statements along paths and loops in the control-flow graph.

Our main interest here is, however, the use of existing type derivations however obtained in order to mechanically construct proofs in the program logic. This will be described now.

Interpretation of the type system The interpretation for the above type system is now obtained by defining for each number n a triple $[\![n]\!] = (A, B, I)$ consisting of a precondition A, a postcondition B, and an invariant I, as follows.

$$[\![n]\!] \equiv \begin{pmatrix} \lambda\,(s_0, s).\ \mathit{True}, \\ \lambda\,(s_0, s, t).\ |\mathit{heap}(t)| \leq |\mathit{heap}(s)| + n, \\ \lambda\,(s_0, s, r).\ |\mathit{heap}(r)| \leq |\mathit{heap}(s)| + n \end{pmatrix}$$

Here, $|h|$ denotes the size of heap h and $\mathit{heap}(s)$ extracts the heap component of a state. We specialise the main judgement form of the bytecode logic to

$$\mathcal{G} \vdash \ell\ \{n\} \equiv \mathit{let}\ (A, B, I) = [\![n]\!]\ \mathit{in}\ \mathcal{G} \vdash \{A\}\,\ell\,\{B\}\,(I).$$

By the soundness of the MOBIUS logic, the derivability of a judgement $\mathcal{G} \vdash \ell\ \{n\}$ guarantees that executing the code located at ℓ will not allocate more that n items, in terminating (postcondition B) and non-terminating (invariant I) cases, provided that $\mathcal{M} \vdash P$ holds. For $(A, B, I) = [\![n]\!]$ we also define the method specification

$$\mathit{Spec}\ n \equiv (\lambda\ s_0.\ \mathit{True}, \lambda\,(s_0, t).\ B(s_0, \mathit{state}(s_0), t), \lambda\,(s_0, s).\ I(s_0, \mathit{state}(s_0), s)),$$

and for a given Λ we define $\mathcal{G}_\Lambda$ pointwise by $\mathcal{G}_\Lambda(\ell) = [\![\Lambda(\ell)]\!]$.

Finally, we say that $\mathcal{M}$ satisfies Σ, notation $\mathcal{M} \models \Sigma$, if for all methods M, $\mathcal{M}(M) = (\mathit{Spec}\ n, \mathcal{G}_\Lambda, \emptyset)$ holds precisely if $\Sigma(M) = n$, where Λ is the context associated with M in $\vdash_\Sigma P$. Thus, method specification table $\mathcal{M}$ contains for each method the precondition, postcondition and invariant from Σ, the (complete) context determined from Λ, and the empty local annotation table $\mathcal{Q}$.

We can now prove the soundness of the typing rules with respect to this interpretation. By induction on the typing rules, we first show that the interpretation of a typing judgement is derivable in the logic.

Proposition 1. *For $\mathcal{M} \models \Sigma$ let M be provided in $\mathcal{M}$ with some annotation table Λ such that $\vdash_{\Sigma,\Lambda} M, l : n$ is progressive. Then $\mathcal{G}_\Lambda \vdash M, l\ \{n\}$.*

From this, one may obtain the following, showing that well-typed programs satisfy the verified-program property:

Theorem 2. *Let $\mathcal{M} \models \Sigma$ and $\vdash_\Sigma P$, and let $\mathcal{M}$ satisfy behavioural subtyping for P. Then $\mathcal{M} \vdash P$.*

Discussion In order to improve the precision of the analysis, a possibility is to combine the type system with a null-pointer analysis. For this, we would specialise the proof rules for instructions which might throw a null-pointer exception. At program points for which the analysis guarantees absence of such exceptions, we may then use a specialised typing rule. For example, a suitable rule for the field access operation is the following.

$$\text{C-GETFLD1}\ \frac{\mathit{getField}(m,l) \quad \mathit{refNotNull}(m,l) \quad \vdash_{\Sigma,\Lambda} m, \mathit{suc}_m(l) : n}{\vdash_{\Sigma,\Lambda} m, l : n}$$

Program points for which the analysis is unable to discharge the side condition $\mathit{refNotNull}(m,l)$ would be dealt with using the standard rule. Similarly, instructions that are guaranteed not to throw runtime exceptions (like load x, store x, dup) may be typed using the optimised rule

$$\text{C-NORTE}\ \frac{\vdash_{\Sigma,\Lambda} m, \mathit{suc}_m(l) : n \quad \mathit{noExceptionInstr}(m,l)}{\vdash_{\Sigma,\Lambda} m, l : n}$$

We expect that justifying these specialised rules using the program logic would not pose major problems, while the formal integration with other program analyses (such as the null-pointer analysis) is a topic for future research.

5 Resource-extended program logic

In this section we give a brief overview of an extension of the MOBIUS base logic as described in Section 3 for dealing with resources in a generic way. The extension addresses the following shortcoming of the basic logic:

Resource consumption Specific resources that we would like to reason about include instruction counters, heap allocation, and frame stack height. A well-known technique for modelling these resources is *code instrumentation*, i.e. the introduction of (real or ghost) variables and instructions manipulating these. However, code instrumentation appears inappropriate for a PCC environment, as it does not provide an end-to-end guarantee that can be understood without reference to the program at hand. In particular, the overall satisfaction of a resource property using code instrumentation requires an analysis of the annotated program, i.e. a proof that the instrumentation variables are introduced and manipulated correctly. Furthermore, the interaction between additional variables of different domains, and between auxiliary variables and proper program variables is difficult to reason about.

Execution traces Here, the goal is to reason about properties concerning a full terminating or non-terminating execution of a program, for example by imposing that an execution satisfies a formula expressed in temporal logics or a policy given in terms of a security automaton. Such specifications may concern the entire execution history, i.e. be defined over a sequence of (intermediate) Bicolano states, and are thus not expressible in the MOBIUS base logic.

Ghost variables are heavily used in JML, both for resource-accounting purposes as well as functional specifications, but are not directly expressible in the base logic.

In this section we extend the base logic by a generic resource-accounting mechanism that may be instantiated to the above tasks. In addition to the work reported here, we have also performed an analysis of the usage made of ghost variables in JML, and have developed interpretations of ghost variables in native and resource-extended program logics [24]. In particular, loc.cit. contains a formalised proof demonstrating how resource counting using ghost variables in native logics may be effectively eliminated, by translating each proof derivation into a derivation in the resource-extended logic.

5.1 Semantic modelling of generic resources

In order to avoid the pitfalls of code instrumentation discussed above, a semantic modelling of resource consumption was chosen. The logic is defined over an extended operational semantics, the judgements of which are formulated over the same components as the standard Bicolano operational semantics, plus a further resource-accounting component [20]. The additional component is of the a priori unspecified type ACT, and occurs as a further component in initial, final, and intermediate states. In addition, we introduce transfer functions that update the content of this component according to the other state components, including the program counter. The operational semantics of the extended framework is then obtained by embedding each non-extended judgement form in a judgement form over extended states and invoking the appropriate transfer functions on the resource component. While these definitions of the operational semantics

are carried out once and for all, the implementation of the transfer functions themselves is programmable. Thus, realisations of the framework for particular resources may be obtained by instantiating the ACT to some specific type and implementing the transfer functions as appropriate. The program logic remains conceptually untouched, i.e. it is structurally defined as the logic from Section 3, but the definitions of assertion transformers and rules, and the soundness proof, are adapted to extended states and modified operational judgements.

In comparison to admitting the definition of ad-hoc extensions to the program logic, we argue that the chosen approach is better suited to the PCC applications, as the consumer has a single point of reference where to specify his policy, namely the implementation of the transfer functions.

5.2 Application: block-booking

As an application of the resource-extended program logic, we consider a scenario where an application repeatedly sends some data across a network provided that each such operation is sanctioned by an interaction with the user. In order to avoid authorisation requests for individual send operations, a high-level language might contain a primitive **auth**(n) that asks the user to authorise n messages in one interaction. A reasonable resource policy for the code consumer then is to require that no send operation be carried out without authorisation, and that at each point of the execution, the acquired authorisations suffice for servicing the remaining **send** operations. (For simplicity, we assume that refusal by the user to sanction an authorisation request simply blocks or leads to immediate non-termination without any observable effect.)

We note that as in the case of the logic loop constructs from the high-level language are mapped to conditional and unconditional jumps that must be typed using the corresponding rules.

We now outline a bytecode-level type and effect system for this task, for a sublanguage of scalar (integer) values and unary static methods. Effects τ are rely-guarantee pairs (m, n) of natural numbers: a code fragment with this effect satisfies the above policy whenever executed in a state with at least m unused authorisations, with at least n unused authorisations being left over upon termination. The number of authorisations that are additionally acquired, and possibly used, during the execution are unconstrained. Types $C, D, \ldots$ are sets of integers constraining the values stored in variables or operand stack positions. Judgements take the form $\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} \ell : C, \tau$, with the following components:

- the abstract store Δ maps local variables to types
- the abstract operand stack η is represented as a list of types
- Ξ is an equivalence relation relation ranging over identifiers ρ from $dom\ \Delta \cup dom\ \eta$ where $dom\ \eta$ is taken to be the set $\{0, \ldots, |\eta| - 1\}$. The role of Ξ is to capture equalities between values on the operand stack and the store.
- instruction labels $\ell = (M, l)$ indicate the current program point, as before
- the type C describes the return type
- the effect τ captures the pre-post-behaviour of the subject phrase with respect to authorisation and send events

- the proof context Λ associates sets of tuples $(\Delta, \eta, \Xi, C, \tau)$ to labels l (implicitly understood with respect to method M).
- the method signature table Σ maps method names to type signatures of the form $\forall i{\in}I.\ C_i \xrightarrow{(m_i,n_i)} D_i$. Limiting our attention to static methods with a single parameter, such a poly-variant signature indicates that for each i in some (unspecified) index set I, the method is of type $C_i \xrightarrow{(m_i,n_i)} D_i$, i.e. takes arguments satisfying constraint C_i to return values satisfying D_i with (latent) effect (m_i, n_i).

In addition to ignoring virtual methods (and consequently avoiding the need for a condition enforcing behavioural subtyping of method specifications), we also ignore exceptions. Finally, while our example program contains simple objects we do not give proof rules for object construction or field access. We argue that this impoverished fragment of the JVML suffices for demonstrating the concept of certificate generation for effects, and leave an extension to larger language fragments as future work.

For an arbitrary relation R, we let $Eq(R)$ denote its reflexive, transitive and symmetric closure. We also define the operations $\Xi - \rho$, $\Xi + \rho$ and $\Xi[\rho := \rho']$ on equivalence relation Ξ and identifiers ρ and ρ', as follows.

$$\begin{aligned} \Xi - \rho &\equiv \Xi \setminus \{(\rho_1, \rho_2) \mid \rho = \rho_1 \vee \rho = \rho_2\} \\ \Xi + \rho &\equiv \Xi \cup \{(\rho, \rho)\} \\ \Xi[\rho := \rho'] &\equiv Eq((\Xi - \rho) \cup \{(\rho, \rho')\}) \end{aligned}$$

The interpretation of position ρ in a pair (O, S) is given by $[\![x]\!]_{(O,S)} = S(x)$ and $[\![n]\!]_{(O,S)} = O(n)$. The interpretation of a triple Δ, η, Ξ in a pair (O, S) is given by the formula

$$[\![\Delta, \eta, \Xi]\!]_{(O,S)} = \begin{cases} dom\ \Delta \subseteq dom\ S \wedge |\eta| = |O| \wedge \\ \forall x \in dom\ \Delta.\ S(x) \in \Delta(x) \wedge \\ \forall i < |\eta|.\ O(i) \in \eta(i) \wedge \\ \forall (\rho, \rho') \in \Xi.\ [\![\rho]\!]_{(O,S)} = [\![\rho']\!]_{(O,S)} \end{cases}$$

With the help of these operations, the type system is now defined by the rules given in Figure 5. Due to the formulation at the bytecode level, the authorisation primitive does not have a parameter but obtains its argument from the operand stack.

The rule for conditionals, E-If, exploits the outcome of the branch condition by updating the types of all variables associated with the top operand stack position in Ξ. This limited form of copy propagation will be made use of in the verification of an example program below.

In the rule of consequence, E-Sub, subtyping on types is denoted by $C <: D$ and given by subset inclusion, and is extended to abstract stores (notation $\Delta <: \Delta'$) and abstract operand stacks (notation $\eta <: \eta'$) in a pointwise fashion. Sub-effecting is given by the reflexive closure of the rule

$$\frac{k \geq m + d \quad l \leq n + d}{(m, n) <: (k, l)}.$$

$$\text{E-Send}\ \frac{M(l) = \mathbf{send} \qquad \Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m-1, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Auth}\ \frac{M(l) = \mathbf{auth} \qquad \forall i \in C.\ i \geq k \qquad \Delta, \eta, \Xi - |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m+k, n)}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Goto}\ \frac{M(l) = \mathsf{goto}\ \ l' \qquad \Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l' : D, (m, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-If}\ \frac{\begin{array}{c} M(l) = \mathsf{ifz}\ \ l' \qquad \Xi' = \Xi - |\eta| \\ \Delta_1 = \Delta[x \mapsto \Delta(x) \cap (\mathbf{Z} \setminus \{0\})]_{(|\eta|,x) \in \Xi} \\ \eta_1 = \eta[i \mapsto \eta(i) \cap (\mathbf{Z} \setminus \{0\})]_{(|\eta|,i) \in \Xi\ \wedge\ 0 \leq i < |\eta|} \\ \Delta_2 = \Delta[x \mapsto \Delta(x) \cap \{0\}]_{(|\eta|,x) \in \Xi} \\ \eta_2 = \eta[i \mapsto \eta(i) \cap \{0\}]_{(|\eta|,i) \in \Xi\ \wedge\ 0 \leq i < |\eta|} \\ \Delta_1, \eta_1, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : (m, n) \qquad \Delta_2, \eta_2, \Xi' \vdash_{\Sigma,\Lambda} M, l' : D, (m, n) \end{array}}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Store}\ \frac{M(l) = \mathsf{store}\ \ x \qquad \Xi' = (\Xi[x := |\eta|]) - |\eta| \qquad \Delta[x \mapsto C], \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, C :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Load}\ \frac{M(l) = \mathsf{load}\ \ x \qquad \Xi' = \Xi[|\eta| := x] \qquad \Delta, \Delta(x) :: \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Push}\ \frac{M(l) = \mathsf{push}\ \ c \quad \Delta, \{c\} :: \eta, \Xi + |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-Binop}\ \frac{M(l) = \mathsf{binop}\ \oplus \qquad C = \{z | z = x\ \oplus\ y, x \in C_1, y \in C_2\} \qquad \Delta, C :: \eta, ((\Xi - |\eta|) - (|\eta| + 1)) + |\eta| \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (m, n)}{\Delta, C_1 :: C_2 :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

$$\text{E-InvS}\ \frac{\begin{array}{c} M(l) = \mathsf{invokestatic}\ M' \qquad \Sigma(M') = \forall i \in I.\ C_i \xrightarrow{\tau_i} D_i \qquad k \in I \\ \Xi' = (\Xi - |\eta|) + |\eta| \qquad \Delta, D_k :: \eta, \Xi' \vdash_{\Sigma,\Lambda} M, suc_M(l) : D, (n_k, n) \end{array}}{\Delta, C_k :: \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m_k, n)}$$

$$\text{E-Vret}\ \frac{M(l) = \mathsf{vreturn}}{\Delta, D, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (0, 0)} \qquad \text{E-Ax}\ \frac{(\Delta, \eta, \Xi, D, \tau) \in \Lambda(l)}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, \tau}$$

$$\text{E-Sub}\ \frac{\begin{array}{c} \Delta', \eta', \Xi' \vdash_{\Sigma,\Lambda} \ell : C, \tau' \\ \Delta <: \Delta' \qquad \eta <: \eta' \\ C <: D \quad \tau' <: \tau \quad \Xi' \subseteq \Xi \end{array}}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} \ell : D, \tau} \qquad \text{E-Univ}\ \frac{\forall\ O\ S.\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} = \mathit{False}}{\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)}$$

Fig. 5. Type and effect system for block-booking

The final rule, E-Univ, allows us to associate an arbitrary effect and result type to a code segment under the condition that the constraints Δ, η, Ξ on the initial state are unsatisfiable. The main use of this rule is in cases where branch conditions render one branch dead code.

In order to prove the soundness of the type system in the extended program logic, we instantiate the parameter ACT to the type of finite words over the set $\{\mathbf{send}\} \cup \{\mathbf{auth}(z) \mid z \geq 0\}$ and implement the transfer functions such that each execution of the primitives **send** and **auth** results in appending the appropriate action to the trace - in case of authorisation events, the number z is obtained by inspecting the topmost value of the operand stack.

We interpret a judgement $\Delta, \eta, \Xi \vdash_{\Sigma,\Lambda} M, l : D, (m, n)$ as the logic statement

$$[\![\Lambda]\!]_M \vdash \{\lambda\, s_0.\ \mathit{True}\}\, M, l\, \{[\![(\Delta, \eta, \Xi, m, n, D)]\!]\}\, ([\![(\Delta, \eta, \Xi, m)]\!]),$$

with the following components. The postcondition $[\![(\Delta, \eta, \Xi, m, n, D)]\!]$ is

$$\lambda\, (s_0, (O, S, h, X), (h, v, Y)).\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} \rightarrow \\ (\exists Z.\ v \in D \wedge Y = XZ \wedge |Z|_{\mathsf{auth}} + m \geq |Z|_{\mathsf{send}} + n).$$

For any terminating execution starting in an initial store and operand stack conforming to the abstractions Δ and η, and respecting the equivalence relation Ξ, this property guarantees that the return value satisfies D. Furthermore, the sub-traces for authorisation and send events (obtained by projecting from the trace Z of all events encountered during the execution of the phrase) satisfy the inequality interpreting the effect.

A similar explanation holds for the definition of the invariant $[\![(\Delta, \eta, \Xi, m)]\!]$,

$$\lambda\, (s_0, (O, S, h, X), (O', S', h', X')).\ [\![\Delta, \eta, \Xi]\!]_{(O,S)} \rightarrow \\ (\exists Z.\ X' = XZ \wedge |Z|_{\mathsf{auth}} + m \geq |Z|_{\mathsf{send}}).$$

The local proof context $[\![\Lambda]\!]_M$ is given by

$$[(M, l) \mapsto (\mathit{True}, [\![(\Delta, \eta, \Xi, m, n, D)]\!], [\![(\Delta, \eta, \Xi, m)]\!])]_{\Lambda(l)=(\Delta,\eta,\Xi,D,(m,n))},$$

i.e. by translating the entries of Λ pointwise. Finally, each specification entry $\Sigma(M) = \forall i \in I.\ C_i \xrightarrow{(m_i,n_i)} D_i$ results in an entry $\mathcal{M}(M) = (R, T, \Phi)$ in the bytecode logic specification table, where

$$\begin{aligned} R(s_0) &= \mathit{True} \\ T((S, h, X), (h, v, Y)) &= \forall i \in I.\, S(\mathrm{arg}) \in C_i \rightarrow \\ &\qquad (\exists\ Z.\ v \in D_i \wedge Y = XZ \wedge \\ &\qquad\quad |Z|_{\mathsf{auth}} + m_i \geq |Z|_{\mathsf{send}} + n_i) \\ \Phi((S, h, X), (O, S', h', X')) &= \forall i \in I.\, S(\mathrm{arg}) \in C_i \rightarrow \\ &\qquad (\exists\ Z.\ X' = XZ \wedge |Z|_{\mathsf{auth}} + m_i \geq |Z|_{\mathsf{send}}) \end{aligned}$$

where arg is the formal parameter. Based on this interpretation, certificate generation may now be obtained by deriving the typing rules from the program logic

and introducing appropriate notions of progressive derivations and well-typed programs (in the absence of virtual methods: without a behavioural subtyping condition), in a similar way as in Section 4. The formalisation of this is left as future research.

5.3 Example

We assume two builtin integer-valued functions `size_string` yielding the number of SMS messages required to send a given string, and `size_book` which gives the size of an address book. Figure 6 presents Java-style pseudocode for sending a given string to all addresses of a given address book after requiring the necessary permissions. The program first computes the total number of SMS messages

```
public interface Parameters {
  int p=...; //some constant >= 0
}
class BlockBooking {
  static void send () {...};
  static void auth (int p) {...};
  void block_send(Java.lang.String s, addrbook b) {
       int n = size_string(s);
       int m = size_book(b);
       int nb_sms = n * m;
       int j = 0;
       int sent = 0;
       while (nb_sms - sent > 0) {
         if j > 0 {
           //current authorisations suffice
           send();
           sent = sent + 1;
           j = j - 1
         } else {
           //acquire p new authorisations
           auth (Parameters.p);
           j = Parameters.p;
         }
       }
       return 0;
  }
}
```

Fig. 6. Program for sending a message using authorisation chunks of size p

and then sends the messages where authorisations are acquired in blocks of size

p, for arbitrary fixed $p \geq 0$. The primitives for sending and authorising messages are modelled as additional (static) methods.

Figure 7 shows the bytecode for method `block_send`, which comprises six basic blocks. In order to verify that this method does not send more messages

```
0  aload_1 //variable s
1  invokestatic sizestring
4  istore_3 //variable n
5  aload_2 // variable b
6  invokestatic sizebook
9  istore 4 //variable m
11 iload_3
12 iload 4
14 imul
15 istore 5 //variable nbms
17 iconst_0
18 istore 6 //variable j
20 iconst_0
21 istore 7 //variable sent

23 iload 5
25 iload 7
27 isub
28 ifle 64

31 iload 6
33 ifle 54

36 invokestatic send
39 iload 7
41 iconst_1
42 iadd
43 istore 7
45 iload 6
47 iconst_1
48 isub
49 istore 6
51 goto 23

54 iconst_3 // parameter p
55 invokestatic auth
58 iconst_3
59 istore 6
61 goto 23

64 iconst_0
65 ireturn
```

Fig. 7. Bytecode for method `BlockBooking.block_send`.

than authorised, we derive the typing

$$[s \mapsto C, b \mapsto D], [\,], \emptyset \vdash_{\Sigma,\Lambda} \texttt{block\_send}, 0 : \{0\}, (0,0)$$

where C and D are arbitrary and

$$\begin{aligned}
\Sigma &\equiv [\texttt{sizestring} \mapsto \{(C,0,0,\mathbf{Z})\}, \texttt{sizebook} \mapsto \{(D,0,0,\mathbf{Z})\}] \\
\Lambda &\equiv [23 \mapsto \{spec_d \mid 0 \leq d\}] \\
spec_d &\equiv (\Delta_d, [\,], \Xi_d, \{0\}, (d,0)) \\
\Delta_d &\equiv [n \mapsto \mathbf{Z}, m \mapsto \mathbf{Z}, nbsms \mapsto \mathbf{Z}, j \mapsto \{d\}, sent \mapsto \mathbf{Z}^{\geq 0}] \\
\Xi_d &\equiv \{(n,n),(m,m),(nbsms,nbsms),(j,j),(sent,sent)\}.
\end{aligned}$$

The proof context Λ contains a single entry, namely a polyvariant loop invariant for instruction 23. The invariant contains one entry for each $0 \leq d$, where the index specifies precisely the content of variable j and links this value to the pre-effect. The equivalence relation relevant at this program point contains

merely the reflexive entries for all (integer) variables. The verification of the above judgement applies the rules syntax-directedly for instructions $0, \ldots, 21$, and then applies the axiom rule for label 23, guarded by an application of rule E-Sub.

The overall verification complements the verification of the above judgement with a justification of the context Λ, by providing a progressive derivation for the loop invariant. Again, this verification proceeds syntax-directedly through the loop, terminating in (subtyping-protected) applications of the rule E-Ax. At the point where method `send` is invoked (instruction label 36) a case-split is performed on the condition $d = 0$. If this condition holds, a vacuous statement is obtained as the invocation occurs in the branch $j > 0$, and our invariant ensures that j contains the value d. The vacuity is detected as the entry for j in Δ is $\emptyset$ at that point: the load instruction at label 36 inserts $(0, j)$ into Ξ, hence the type associated with j in the fall-through-hypothesis of the branch at label 33 (in particular: at label 36) is $\{d\} \cap (\mathbf{Z} \setminus \{0\}) = \emptyset$ where the term $\{d\}$ was propagated unmodified to instruction 36 from instruction 23. Consequently, the case $d = 0$ may be immediately discharged by an invocation of rule E-Univ. The case $d > 0$ admits the application of the proof rule E-Send, and the remainder of the branch is again proven in a syntax-directed fashion.

Type checking and inference Again, we briefly discuss these issues for this system. The type system is generic in that types may be arbitrary sets of integers. In order to support effective typechecking and inference one must of course restrict these sets themselves and also the sets of types that arise in annotations and method specifications. A popular and for our intended application sufficient way consists of restricting types to convex polyhedra specified by a system of linear inequalities and to confine sets of types to those arising by intersecting a fixed convex polyhedron with a hyperplane specified by one or more additional parameters. Notice that the types in our running example are all of this form.

When we make this restriction (formally by applying the subtyping rule immediately after each rule to bring the types back into the polyhedral format) then type checking amounts to checking inclusion of convex polyhedra which can be efficiently performed by linear programming. Furthermore, Farkas' Lemma also furnishes short, efficiently computable, and efficiently checkable certificates [21, 28]. Indeed, since any convex polyhedron is the intersection of hyperplanes, deciding containment of convex polyhedra reduces to deciding whether a convex polyhedron $H = \{\boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b}\}$ is contained in a hyperplane of the form $P = \{\boldsymbol{x} \mid \boldsymbol{c}^T\boldsymbol{x} \leq d\}$. This, however, is the case iff $\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in H\} \leq d$; a linear programming problem. Now, the latter inequality can be certified by providing a vector $\boldsymbol{r} \geq 0$ (componentwise) such that $\boldsymbol{r}^T A = \boldsymbol{c}^T$ and $\boldsymbol{r}^T\boldsymbol{b} \leq d$. For then, whenever $\boldsymbol{x} \in H$, i.e., $A\boldsymbol{x} \leq \boldsymbol{b}$ then $\boldsymbol{c}^T\boldsymbol{x} = \boldsymbol{r}^T A\boldsymbol{x} \leq \boldsymbol{r}^T\boldsymbol{b} \leq d$. Farkas' lemmas asserts that such a vector $\boldsymbol{r}$ exists whenever $\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in H\} \leq d$. Given its existence we can efficiently compute it by minimising $\boldsymbol{y}^T\boldsymbol{b}$ subject to $\boldsymbol{y}^T A = \boldsymbol{c}^T$ and $\boldsymbol{y} \geq 0$.

Regarding automatic type inference as opposed to type checking one has to find unknown convex polyhedra specified by fixpoint equations. Besson et al. [12]

report that this can be done by iteration using widening heuristics from [19]. The range and efficiency remains, however, unexplored in loc. cit. In our particular application we expect constraints to be sufficiently simple so that these heuristics or those proposed in [26] will be successful. Inference of the equivalence relations Ξ can be achieved by employing standard copy-propagation techniques known from compiler constructions.

6 Discussion

We have described the use of the Mobius base logic as a unified backend for both program analyses and type systems. The Mobius base logic has been formally proved sound with respect to the Bicolano formalisation of the JVM. Compared to direct soundness proofs of type systems and analyses with respect to Bicolano the use of the Mobius base logic as an intermediary offers two distinctive advantages. First, the soundness proof of the Mobius base logic already does much of the work that is common to soundness proofs, in particular inducting on steps in the operational semantics and stack height. The Mobius logic is more transparent and allows for proof by invariant and recursion. Secondly, the standardised format of assertions in the Mobius base logic makes it easier to compare results of different type systems and analyses and also to assess whether the asserted property coincides with the intuitively desired property.

The resource extension to both Bicolano and the Mobius base logic allows for direct specification and certification of resource-related intensional properties without having to go through indirect observations such as values of ordinary program variables that are externally known to reflect some resource behaviour. This is particularly important in the PCC scenario where providers and users of specifications and certificates do not coincide and might have different objectives.

Similarly, the strong invariants enhance the expressive power of the Mobius base logic compared to standard Hoare logics in that resource behaviour of nonterminating programs is appropriately accounted for. In this way, the usual strong guarantees of type systems and program analyses may be adequately reflected in the logic.

We have demonstrated this use of the Mobius base logic on one of the Mobius case studies: a block-booking scheme whose deployment could avoid the inflation of permission requests that lead to social vulnerabilities.

Acknowledgements This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein. We are grateful to all members of the MOBIUS Working Group on work package 3, in particular Benjamin Gregoire, David Pichardie, Aleksy Schubert and Randy Pollack, for the numerous discussions on program logics, JML, and types, and on formalising these in theorem provers. The constructive feedback from the reviewers helped us to improve content and presentation of the paper.

References

1. E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In *Logic for Programming Artificial Intelligence and Reasoning*, number 3452 in Lecture Notes in Computer Science, pages 380–397. Springer-Verlag, 2005.
2. A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.
3. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Theorem Proving in Higher-Order Logic*, volume 3223 of *Lecture Notes in Computer Science*, pages 34–49, Berlin, Sept. 2004. Springer-Verlag.
4. F. Y. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.
5. G. Barthe and C. Fournet, editors. *Trustworthy Global Computing, Third Symposium (TGC'07), Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
7. B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering*, volume 2621 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, Apr. 2003.
8. L. Beringer and M. Hofmann. A bytecode logic for JML and types. In *Asian Programming Languages and Systems Symposium*, Lecture Notes in Computer Science 4279, pages 389–405. Springer-Verlag, 2006.
9. L. Beringer and M. Hofmann. Secure information flow and program logics. In *IEEE Computer Security Foundations Workshop*. IEEE Press, 2007.
10. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452, pages 347–362. Springer-Verlag, 2005.
11. F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 2006.
12. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Inria Research Report 6333, 2007.
13. D. Cachera, T. P. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106. Springer-Verlag, 2005.
14. B. Chang, A. Chlipala, and G. Necula. A framework for certified program analysis and its applications to mobile-code safety. In E. Emerson and K.S.Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference (VMCAI'06), Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 174–189. Springer-Verlag, 2006.
15. B. Chang, A. Chlipala, G. Necula, and R. Schneck. The open verifier framework for foundational verifiers. In J. Morrisett and M. Fähndrich, editors, *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 1–12. ACM Press, 2005.

16. MOBIUS Consortium. Deliverable 1.1: Resource and information flow security requirements. Available online from http://mobius.inria.fr, 2006.
17. MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from http://mobius.inria.fr, 2006.
18. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, NY, ACM SIGPLAN Not. 12(8):1–12, Aug. 1977.
19. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.
20. P. Czarnik and A. Schubert. Extending operational semantics of the java bytecode. In Barthe and Fournet [5], pages 57–72.
21. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.
22. X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*, pages 67–78, New York, NY, USA, January 2007. ACM Press.
23. R. Hähnle, J. Pan, P. Rümmer, and D. Walter. Integration of a security type system into a program logic. In U. Montanari, D. Sannella, and R. Bruni, editors, *Trustworthy Global Computing, Second Symposium (TGC'06), Revised Selected Papers*, volume 4661 of *Lecture Notes in Computer Science*, pages 116–131. Springer-Verlag, 2007.
24. M. Hofmann and M. Pavlova. Elimination of ghost variables in program logics. In Barthe and Fournet [5], pages 1–20.
25. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1998.
26. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Proc. ACM POPL 2004*, pages 330–341, 2004.
27. G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
28. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC Computer Science Laboratory, June 1981.
29. T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer-Verlag, 2002.
30. D. Pichardie. Bicolano – Byte Code Language in Coq. http://mobius.inia.fr/bicolano. Summary appears in [17], 2006.
31. C. L. Quigley. A Programming Logic for Java Bytecode Programs. In D. A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, (TPHOLs'03), Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2003.
32. M. Wildmoser. *Verified Proof Carrying Code*. PhD thesis, Institut für Informatik, Technische Universität München, 2005.
33. M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In J.-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Theoretical Computer Science*, pages 333–347. Kluwer Academic Publishing, Aug. 2004.
34. T. Y. Woo and S. S. Lam. A semantic model for authentication protocols. In *RSP: IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.

Implementing a type system for amortised heap-space analysis

Martin Hofmann Dulma Rodriguez

September 11, 2008

1 Introduction

The prediction of resource consumption in programs has gained interest in the last years. It is important for a number of areas, notably embedded systems and safety critical systems. Different approaches to achieve bounded resource consumption has been analyzed. One of them is the use of sized types as initially proposed by Hughes and Pareto [HP99]. Another approach, based on an amortized complexity analysis, has been studied by Hofmann and Jost for a functional language in [HJ03] and for a Java-like language in [HJ06].

Their analysis in [HJ06] is based on a Java-like class-based object-oriented language without garbage collection, but with explicit deallocation in the style of C's `free()`. Such programs may be evaluated by maintaining a set of free memory units, the freelist. Upon object creation a number of heap units required to store the object is taken from the freelist provided it contains enough units; each deallocated heap unit is returned to the freelist. An attempt to create a new object with an insufficient freelist causes unsuccessful abortion of the program.

The goal is to predict a bound to the initial size that the freelist must have so that a given program may be executed without causing unsuccessful abortion due to penury of memory. They have achieved this using an amortized analysis. The idea is to assign objects a potential. Any object creation must be paid for from the potential in scope and the potential of the initial configuration furnishes an upper bound on the total heap consumption.

While type inference and automated type checking has already been developed for the functional language within the EmBounded Project ([HDF$^+$05], [HBH$^+$07]), most of the properties of the type system for the Java-like language (called Resource Aware JAva – RAJA) are still unknown.

Now we describe automated type-checking for a modified version of RAJA which we call RAJA$^+$. Moreover we prove that the type checking algorithm we present is sound and complete with respect to the declarative type system of RAJA$^+$. This proves the decidability of the system.

On the other side, we have implemented the type checking algorithm as well as a parser and an interpreter for RAJA$^+$ programs in Ocaml. This prototype implementation supports further investigation in this area.

Contents. The following section describes briefly the systems RAJA and RAJA$^+$ explaining the background. In Section 3 we formulate the algorithmic typing rules and verify the soundness (Section 3.1) and completeness (Section

3.2). Then, decidability of the system follows and is proved in Section 3.3. In Section 4 we describe some interesting features of the Ocaml implementation of the type checker like the implementation of coinductive definitions (Section 4.1) and we show examples of code in Section 4.2. Finally, we discuss future and related work in the conclusions.

2 The language RAJA

2.1 Views and potential

We aim at assigning objects a potential so that any object creation must be paid for from the potential in scope and the potential of the initial configuration furnishes an upper bound on the total heap consumption. For an interesting analysis we should be able to assign objects of the same class different potentials. Thus, we need more refined types than classes to assign potential to. We introduce the *views*. A view on an object shall determine its contribution to the potential. A refined type then consists of a class C and a view r and is written C^r.

The meaning of views is given by three maps $\Diamond$ defining potentials, A defining views of attributes, and M defining refined method types. More precisely,

1. $\Diamond$: Class $\times$ View $\to \mathbb{Q}^+$ assigns each class its potential according to the employed view.

2. A : Class $\times$ View $\times$ Field $\to$ View $\times$ View determines the refined types of the fields. A different view may apply according to whether a field is read from (get-view) or written to (set-view).

3. M : Class $\times$ View $\times$ Method $\to$ $\mathcal{P}$(Views of Arguments $\to$ Effect $\times$ View of Result) assigns refined types and effects to methods. The effect is a pair of numbers representing the potential consumed before and released after method invocation.

The overall potential of a runtime configuration is the (possibly infinite) sum over all access paths in scope that lead to an actual object.

2.2 Extending FJEU to RAJA

Our formal model of Java, FJEU, is an extension of Featherweight Java (FJ) [IPW99] with attribute update, conditional and explicit deallocation. It is thus similar to Flatt et al. Classic Java [FKF98].

An FJEU program $\mathscr{C}$ is a partial finite map from class names to class definitions, which we also refer to as *class table*. Each class table $\mathscr{C}$ implies a subtyping relation $<:$ among the class names in the standard way by inheritance.

Each class consists of an optional super-class, a set of attributes (or fields) with their types, and a set of methods with their types and bodies. A method body is an expression in let normal form (nested expressions flattened out using a sequence of let-definitions). Classes have only one implicit constructor that initializes all attributes to a nil-value.

We now extend FJEU to an annotated version, RAJA, (Resource Aware JAva) as announced in the Introduction. A *RAJA program* is an annotation of an FJEU class table $\mathscr{C}$ in the form of a sextuple $\mathscr{R} = (\mathscr{C}, \mathscr{V}, \Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot,\cdot), \mathsf{A}^{\mathrm{set}}(\cdot,\cdot), \mathsf{M}(\cdot,\cdot))$ specified as follows:

1. $\mathscr{V}$ is a possibly infinite set of views. For each class $C \in \mathrm{dom}(\mathscr{C})$ and for each view $r \in \mathscr{V}$ the pair C^r is called a RAJA class. We refer to RAJA classes also as *(refined) types*.

2. $\Diamond(\cdot)$ assigns to each RAJA class C^r a number $\Diamond(C^r) \in \mathbb{D}$, where $\mathbb{D} = \mathbb{R} \cup \infty$. This number will be used to define the potential of a heap configuration under a given static RAJA typing.

3. $\mathsf{A}^{\mathrm{get}}(\cdot,\cdot)$ and $\mathsf{A}^{\mathrm{set}}(\cdot,\cdot)$ assign to each RAJA class C^r and attribute $a \in \mathsf{A}(C)$ two views $q = \mathsf{A}^{\mathrm{get}}(C^r, a)$ and $s = \mathsf{A}^{\mathrm{set}}(C^r, a)$. The intention is that if $D = C.a$ is the FJEU type of attribute a in C then the RAJA type D^q will be the type of an access to a, whereas the (intendedly stronger) type D^s must be used when updating a. The stronger typing is needed since an update will possibly affect several aliases.

4. $\mathsf{M}(\cdot,\cdot)$ assigns to each RAJA class C^r and method $m \in \mathsf{M}(C)$ having method type $E_1, \ldots, E_j \to E_0$ a j-ary polymorphic RAJA method type $\mathsf{M}(C^r, m)$.

   A *j-ary polymorphic RAJA method type* is a (possibly empty or infinite) set of j-ary monomorphic RAJA method types. A *j-ary monomorphic RAJA method type* consists of $j+1$ views and two numbers $p, q \in \mathbb{D}$, written $r_1, \ldots, r_j \xrightarrow{p/q} r_0$.

   The idea is that if m (of FJEU-type $E_1, \ldots, E_j \to E_0$) has (among others) the monomorphic RAJA method type $r_1, \ldots, r_j \xrightarrow{p/q} r_0$ then it may be called with arguments $v_1 : E_1^{r_1}, \ldots, v_j : E_j^{r_j}$, whose associated potential will be consumed, as well as an additional potential of p. Upon successful completion the return value will be of type $E_0^{r_0}$ hence carry an according potential. In addition to this a potential of another q units will be returned.

   We sometimes write $E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{p/q} E_0^{r_0}$ to denote an FJEU method type combined with a corresponding monomorphic RAJA method type.

Potential. The definition of potential of a runtime configuration involves the following concepts which we explain informally here. We write $\|(v : r).\vec{p}\|_\sigma^{\mathrm{stat}}$ for the view on the object reached from v (of view r) via access path $\vec{p}$ when accessed in this way. We write $\Phi_\sigma(v : r)$ for the potentials of the data structures reachable from v in the heap σ, when viewed through r. For example, $\Phi_\sigma(v : r) = \sum_{\vec{p}} \Diamond(D^s)$, where D is the dynamic type of the record reached from v in σ via $\vec{p}$, whereas $s = \|(v : r).\vec{p}\|_\sigma^{\mathrm{stat}}$. The overall potential of a runtime configuration is the (possibly infinite) sum over all access paths in scope that lead to an actual object.

We will now define when such a RAJA-annotation of an FJEU class table is indeed valid; in particular this will require that each method body is typable with each of the monomorphic RAJA method types given in the annotation.

2.2.1 RAJA Subtyping Relation

We use in this paper a definition of subtyping that differs slightly from the definition in [HJ06]. There, a preorder on views is defined, and combined with the usual subtyping of classes based on inheritance to build subtyping of refined types:

$$\mathit{Compat}(<:, C, D, r, s) \iff$$

$$\Diamond(C^r) \geq \Diamond(D^s) \tag{2.1}$$
$$\forall a \in \mathsf{A}(D)\ .\ \mathsf{A}^{\mathrm{get}}(C^r, a) <: \mathsf{A}^{\mathrm{get}}(D^s, a) \tag{2.2}$$
$$\forall a \in \mathsf{A}(D)\ .\ \mathsf{A}^{\mathrm{set}}(D^s, a) <: \mathsf{A}^{\mathrm{set}}(C^r, a) \tag{2.3}$$
$$\forall m \in \mathsf{M}(D)\ .\forall \beta \in \mathsf{M}(D^s, m)\ .\exists \alpha \in \mathsf{M}(C^r, m)\ .\ \alpha <: \beta \tag{2.4}$$

where we extend $<:$ to monomorphic RAJA method types as follows: if $\alpha = r_1, \ldots, r_j \xrightarrow{p/q} r_0$ and $\beta = s_1, \ldots, s_j \xrightarrow{t/u} s_0$ then $\alpha <: \beta$ is defined as $p \leq t$ and $q \geq u$ and $r_0 <: s_0$ and $s_i <: r_i$ for $i = 1, \ldots j$.

The subtyping relation $r \sqsubseteq s$ between views is now defined as the largest relation $\sqsubseteq$ such that

$$r \sqsubseteq s \implies \mathit{Compat}(\sqsubseteq, C, C, r, s) \text{ for all } C$$

We extend subtyping to RAJA-classes by

$$C^r <: D^s \iff C <: D \text{ and } r \sqsubseteq s \tag{2.5}$$

We define a preorder by coinduction on refined types directly. Our definition is more general and permits to type more examples. We have not yet extended the rigorous proof of soundness to this version but do not foresee major obstacles.

Definition 2.1 (Subtyping of RAJA types) *We define a preorder $<:$ on RAJA types as the largest relation $(C^r <: D^s)$ such that if $(C^r <: D^s)$ then:*

$$\Diamond(C^r) \geq \Diamond(D^s) \tag{2.6}$$
$$\forall a \in \mathsf{A}(D)\ .\ C.a^{\mathsf{A}^{\mathrm{get}}(C^r, a)} <: D.a^{\mathsf{A}^{\mathrm{get}}(D^s, a)} \tag{2.7}$$
$$\forall a \in \mathsf{A}(D)\ .\ D.a^{\mathsf{A}^{\mathrm{set}}(D^s, a)} <: C.a^{\mathsf{A}^{\mathrm{set}}(C^r, a)} \tag{2.8}$$
$$\forall m \in \mathsf{M}(D)\ .\forall \beta \in \mathsf{M}(D^s, m)\ .\exists \alpha \in \mathsf{M}(C^r, m)\ .(C.m)^{\alpha} <: (C.m)^{\beta} \tag{2.9}$$

where we extend $<:$ to monomorphic RAJA method types as follows:

Definition 2.2 (Subtyping of RAJA methods) *If $C.m = E_1, \ldots, E_j \to E_0$, $\alpha = r_1, \ldots, r_j \xrightarrow{p/q} r_0$ and $\beta = s_1, \ldots, s_j \xrightarrow{t/u} s_0$ then $(C.m)^{\alpha} <: (C.m)^{\beta}$ is defined as $p \leq t$ and $q \geq u$ and $E_0^{r_0} <: E_0^{s_0}$ and $E_i^{s_i} <: E_i^{r_i}$ for $i = 1, \ldots j$.*

If Γ and Θ are contexts we write $\Gamma <: \Theta$ if $\forall x \in \Theta\,.\,\Gamma_x <: \Theta_x$.

2.2.2 Sharing Relation

The following definition is important for correctly using variables more than once. If a variable is to be used more than once, e.g., as an argument to a method, then the different occurrences must be given different types which are chosen such that the individual potentials assigned to each occurrence add up to the total potential available. For example if we have $x : C^r$ we can use the variable x with the types C^{s_1} and C^{s_2} if $\curlyvee(r\,|s_1, s_2\,)$ holds.

Definition 2.3 (Sharing Relation) *We define the* sharing relation *between a single view r and a multiset of views $\mathcal{D}$ written $\curlyvee(r\,|\mathcal{D}\,)$ [1] as the largest relation $\curlyvee$, such that if $\curlyvee(r\,|\mathcal{D}\,)$ then for all $C \in \mathscr{C}$:*

$$\Diamond(C^r) \geq \sum_{s\in\mathcal{D}} \Diamond(C^s) \tag{2.10}$$

$$\forall s \in \mathcal{D}\,.\, r \sqsubseteq s \tag{2.11}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(C))\,.\,\curlyvee\big(\mathsf{A}^{\mathrm{get}}(C^r, a)\,\big|\mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big)\big) \tag{2.12}$$

where $\mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big) = \{\mathsf{A}^{\mathrm{get}}(C^s, a) \mid s \in \mathcal{D}\}$. When $\mathcal{D} = \{s_1, \ldots, s_i\}$ is a finite multiset, we also write $\curlyvee(r\,|s_1, \ldots, s_i\,)$ for $\curlyvee(r\,|\mathcal{D}\,)$.

2.2.3 Typing RAJA

We now give the formal definition of the RAJA-typing judgement. See Figure 1. The type system allows us to derive assertions of the form $\Gamma \vdash^{n}_{n'} e : C^r$ where e is an expression or program phrase, C is a Java class, r is a view (so C^r is a refined type). Γ maps variables occurring in e to refined types; we often write Γ_x instead of $\Gamma(x)$. Finally n, n' are nonnegative numbers. The meaning of such a judgement is as follows. If e terminates successfully in some environment η and heap σ with unbounded memory resources available then it will also terminate successfully with a bounded freelist of size at least n plus the potential ascribed to η, σ with respect to the typings in Γ. Furthermore, the freelist size upon termination will be at least n' plus the potential of the result with respect to the view r.

The typing rules are standard. The most interesting ones are $(\Diamond Share)$ and $(\Diamond Waste)$. First we notice that they are not syntax directed, thus, they need to be eliminated when it comes to implement the system in the next section.

The uses of $(\Diamond Waste)$ are twofold: on the one hand, it deals with subtyping, and on the other hand, it allows to change the effects in the typing derivation if some conditions are fulfilled.

The purpose of the $(\Diamond Share)$ rule is to ensure that a variable can be used twice without duplication of potential. Imagine we want to check that an expression e has the type C^r in the context $\Gamma, x\!:\!D^s$, and we know that the variable x appears twice in e, i.e. we have $\Gamma, x\!:\!D^s \vdash^{n}_{n'} e[x/y, x/z] : C^r$. This is allowed if there are views q_1 and q_2 with $\curlyvee(s\,|q_1, q_2\,)$. The important point here is that the rule gives no information about how to find those views q_1 and q_2. This is another important aspect of the implementation of the type system that we will discuss in the next section.

[1] Similar to the definition of subtyping on RAJA types it is possible to define sharing on RAJA types too rather than views, eg. $\curlyvee(C^r\,|C^{s_1}, C^{s_2}\,)$. We would need to extend the soundness proof to this version, which remains under investigation.

RAJA Typing $\boxed{\Gamma \vdash^{n}_{n'} e : C^r}$

$$\frac{}{\varnothing \vdash^{\Diamond(C^r)\,+\,\mathsf{Size}\,(C)}_{0} \texttt{new}\ C : C^r}\ (\Diamond New) \qquad \frac{}{x{:}C^r \vdash^{0}_{\Diamond(C^r)\,+\,\mathsf{Size}\,(C)} \texttt{free}(x) : E^r}\ (\Diamond Free)$$

$$\frac{C <: E}{x{:}E^r \vdash^{0}_{0} (C)x : C^r}\ (\Diamond Cast) \qquad \frac{}{\varnothing \vdash^{0}_{0} \mathsf{null} : C^r}\ (\Diamond Null) \qquad \frac{}{x{:}C^r \vdash^{0}_{0} x : C^r}\ (\Diamond Var)$$

$$\frac{s = \mathsf{A}^{\mathrm{get}}(C^r, a) \qquad D = C.a}{x{:}C^r \vdash^{0}_{0} x.a : D^s}\ (\Diamond Access) \qquad \frac{\mathsf{A}^{\mathrm{set}}(C^r, a) = s \qquad C.a = D}{x{:}C^r, y{:}D^s \vdash^{0}_{0} x.a\texttt{<-}y : C^r}\ (\Diamond Update)$$

$$\frac{\Gamma_1 \vdash^{n}_{n'} e_1 : D^s \qquad \Gamma_2, x{:}D^s \vdash^{n'}_{n''} e_2 : C^r}{\Gamma_1, \Gamma_2 \vdash^{n}_{n''} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C^r}\ (\Diamond Let)$$

$$\frac{\left(E_1^{q_1}, \ldots, E_j^{q_j} \xrightarrow{n/n'} E_0^{q_0}\right) \in \mathsf{M}(C^r, m)}{x{:}C^r, y_1{:}E_1^{q_1}, \ldots, y_j{:}E_j^{q_j} \vdash^{n}_{n'} x.m(y_1, \ldots, y_j) : E_0^{q_0}}\ (\Diamond Invocation)$$

$$\frac{x \in \Gamma \qquad \Gamma \vdash^{n}_{n'} e_1 : C^r \qquad \Gamma \vdash^{n}_{n'} e_2 : C^r}{\Gamma \vdash^{n}_{n'} \texttt{if}\ x\ \texttt{instanceof}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : C^r}\ (\Diamond Conditional)$$

$$\frac{\curlyvee(s\,|q_1, q_2\,) \qquad \Gamma, y{:}D^{q_1}, z{:}D^{q_2} \vdash^{n}_{n'} e : C^r}{\Gamma, x{:}D^s \vdash^{n}_{n'} e[x/y, x/z] : C^r}\ (\Diamond Share)$$

$$\frac{n \geq u \quad n + u' \geq n' + u \quad \Theta \vdash^{u}_{u'} e : D^s \quad \Gamma <: \Theta \quad D^s <: C^r}{\Gamma \vdash^{n}_{n'} e : C^r}\ (\Diamond Waste)$$

RAJA Method Typing $\boxed{\vdash m : \alpha\ \mathsf{ok}}$

$$\frac{m \in \mathsf{M}(C) \qquad \alpha = E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in \mathsf{M}(C^r, m) \qquad \curlyvee(r\,|q, s\,)}{\texttt{this}{:}C^q, x_1{:}E_1^{r_1}, \ldots, x_j{:}E_j^{r_j} \vdash^{n\,+\,\Diamond(C^s)}_{n'} \mathsf{M}_{\mathsf{body}}(C, m) : E_0^{r_0}}$$
$$\vdash m : \alpha\ \mathsf{ok}$$

Figure 1: Typing RAJA

Definition 2.4 (Well-typed RAJA-program) *A RAJA-program* $\mathscr{R} = (\mathscr{C}, \mathscr{V}, \Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot,\cdot), \mathsf{A}^{\mathrm{set}}(\cdot,\cdot), \mathsf{M}(\cdot,\cdot))$ *is* well-typed *if for all* $C \in \mathscr{C}$ *and* $r \in \mathscr{V}$ *the following conditions are satisfied:*

1. $\mathsf{S}(C) = D \Rightarrow C^r <: D^r$
2. $\forall a \in \mathsf{A}(C)\ .\ \mathsf{A}^{\mathrm{set}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^r, a)$
3. $\forall m \in \mathsf{M}(C)\ .\ \forall \alpha \in \mathsf{M}(C^r, m)\ .\ \vdash m : \alpha\ \mathsf{ok}$

Next, we define the judgement $\vdash m : \alpha\ \mathsf{ok}$ where m is the name of a method in a RAJA class C^r, and α is a RAJA type for this method. It means that α is a valid RAJA type for m if the method body of m can be typed with the arguments, return type and effects as specified in α. There is another interesting point here: in order to use the potential of this we put it in the context with a modified type. For example, if we have $\mathsf{this} : C^r$ and $\Diamond(C^r) = 1$, if we can find two views such that $\curlyvee(r\,|q, s)$ with $\Diamond(C^q) = 0$ and $\Diamond(C^s) = 1$, then we put this in the context with type $\mathsf{this} : C^q$ and we can use the potential 1 of C^s when typing the method body. Again, the rule gives no information about how to find the views q and s.

2.2.4 Main Result

This is the final corollary of [HJ06] which is a direct consequence of the main result and it is in this form that they intend to use it.

Corollary 2.5 *Suppose that* $\mathscr{C}$ *is an FJEU program containing (in Java notation) a class* `List` *of singly-linked lists with boolean entries, a class* `C` *containing a method* `void C.main(List args)`*, and arbitrary other classes and methods.*

Suppose furthermore, that there exists a RAJA-annotation of this program containing a view a *where* $\Diamond(\texttt{List}^a) = k \in \mathbb{N}$ *and* $\mathsf{A}^{\mathrm{get}}(\texttt{List}^a, \texttt{next}) = a$ *then evaluating* $\texttt{C.main}(args)$ *in a heap where* `args` *points to a linked list of length* l *requires at most* kl *memory cells.*

2.3 Extending RAJA to RAJA$^+$

Now we present a small modification of the system RAJA. Recall the rule $(\Diamond Share)$ and that it contains no information about how to find the views q_1 and q_2 to share with. There are two possibilities of implementing this rule: either we find a way to infer the views or we annotate the variables with them. The last option permits to implement the $(\Diamond Share)$ rule in an easy way because the only task of the type checker is to check that the given sharing is correct. We have chosen this variant for the implementation because it is still not clear how the inference of these intermediate views would work. This falls under type inference as opposed to typechecking and is not studied here.

A RAJA$^+$ program is a RAJA program where all variable occurrences are annotated with a view. The typing rules are modified in the obvious way. For example we have the rule $(\Diamond^+ Access)$:

$$\frac{s = \mathsf{A}^{\mathrm{get}}(C^r, a) \qquad D = C.a}{x{:}C^r \ \vdash_0^0\ x^r.a : D^s}\ (\Diamond^+ Access)$$

or the rule $(\Diamond^+Share)$:

$$\frac{\curlyvee(s\,|q_1,q_2\,) \qquad \Gamma, y{:}D^{q_1}, z{:}D^{q_2} \vdash^{n}_{n'} e : C^r}{\Gamma, x{:}D^s \vdash^{n}_{n'} e[x^{q_1}/y^{q_1}, x^{q_2}/z^{q_2}] : C^r}\ (\Diamond^+Share)$$

In our syntax the method copy of lists found in [HJ06] looks then (simplified) like this:

```
List<b>,0 copy(0) {
   let Cons<b> res = new Cons in
   let List<a> next = [this as c].next in
   let List<b> nres = [next as a].copy() in
   [res as b].next <- [nres as b] in
   return [res as b];
}
```

3 Algorithmic Typing of RAJA$^+$ Programs

Now the question is how we actually typecheck RAJA$^+$ programs. We need to implement the RAJA$^+$ typing rules, in order to decide $\Gamma \vdash^{n}_{n'} e : C^r$. As pointed out before, there are problematic rules in the system that cannot be implemented directly because they are not syntax directed. These are $(\Diamond^+Share)$ and $(\Diamond^+Waste)$.

The main idea of the implementation is thus to eliminate them and to integrate them in the other syntax directed rules. This way we get a deterministic algorithm. Moreover we will prove that the eliminated rules are still admissible in the algorithmic system, proving this way the correctness of the algorithm.

Recall that the effects of the rule $(\Diamond^+Waste)$ are twofold: on the one hand, it deals with subtyping, and on the other hand, it allows to change the effects. Instead of using $(\Diamond^+Waste)$, we will integrate subtyping in every rule. Moreover, we will see n as another input of every rule, and n' as an output. For example in the rule $(\Diamond^+New)$ we have

$$\frac{}{\varnothing \vdash^{\Diamond(C^r)\,+\,\mathsf{Size}\,(C)}_{0} \mathtt{new}\ C : C^r}\ (\Diamond^+New)$$

$\Diamond(C^r) + \mathsf{Size}\,(C)$ is in this case the amount of free units that are needed in order to create a new object of type C^r. In the algorithmic rule $(\vdash New)$ we have

$$\frac{n \geq \Diamond(D^r) + \mathsf{Size}\,(D) \qquad D^r <: C^r}{\emptyset \vdash^{n}_{n\,-\,(\Diamond(D^r)\,+\,\mathsf{Size}\,(D))} \mathtt{new}\ D \leftleftarrows C^r}\ (\vdash New)$$

First, notice the integration of subtyping in the rule. Second, the amount of free units n only needs to be greater or equal to $\Diamond(D^r) + \mathsf{Size}\,(D)$ so that it is possible to create an object of type D^r. The *output* n' is then $n - (\Diamond(D^r) + \mathsf{Size}\,(D))$. If the given expression is not a subtype of the given type or needs more than n heap units for its evaluation the algorithm will fail.

The purpose of the ($\Diamond^{+}Share$) rule is to ensure that a variable can be used twice without duplication of potential. We suppose here w.l.o.g. that a variable can appear twice only in a let expression or in an if expression. This means that we do not allow expressions like $x.m(y, y)$. This is not really a restriction to the system, because we can just define a copy of the variable y with let $z = y$ in $x.m(y, z)$ which is allowed again.

In the Ocaml implementation we do allow such expressions like $x.m(y, y)$. We show the algorithmic rules here with that restriction because it makes them easier to read and understand, i.e. only the algorithmic rules ($\vdash Let$) and ($\vdash Conditional$) deal with variable sharing.

Recall that in RAJA^{+} all variable occurrences are annotated with a view. The idea of the algorithmic sharing is to collect the annotations of a variable, and then to check if the given sharing is correct. For example in $x : D^{s}, y : C^{r} \vdash^{n}_{n'}$ let $z = y^{r_1}$ in $x^{s}.m(y^{r_2}, z^{r_1})$ we have to check $\curlyvee(r\,|r_1, r_2\,)$. In if expressions though we cannot collect both branches. Imagine the following expression: $x{:}D^{s}, y{:}C^{r} \vdash^{n}_{n'}$ if x instanceof C then y^{r} else y^{r}. The variable y is used only once, either in the then branch or in the else branch, thus, there is no duplication of potential. Thus, it would be too restrictive to reject this expression because probably $\curlyvee(r\,|r, r\,)$ does not hold. The solution is to check the two branches separately: $\curlyvee(r\,|r\,)$ and $\curlyvee(r\,|r\,)$ which is trivial.

To resume, during the typechecking process we need to collect views in order to check the sharing, but we cannot use a simple list to collect the views. We need a data type that is flexible enough to specify if we are in a let or in an if expression. We introduce *algorithmic views*, syntactic expressions based on views.

$$\alpha \quad ::= \quad s \,|\, \alpha + \alpha \,|\, \alpha \vee \alpha$$

Moreover, we impose the distributive laws:

1. $\alpha + (\beta \vee \gamma) = (\alpha + \beta) \vee (\alpha + \gamma)$
2. $\alpha \vee (\beta + \gamma) = (\alpha \vee \beta) + (\alpha \vee \gamma)$

For this reason we assume in the following that algorithmic views are in normal form $(s_{11} + \ldots + s_{1i}) \vee \ldots \vee (s_{n1} + \ldots + s_{nj})$. Now we extend the definition of sharing to algorithmic views in normal form.

$$\curlyvee(r\,|(s_{11} + \ldots + s_{1i}) \vee \ldots \vee (s_{n1} + \ldots + s_{nj})\,) \text{ iff}$$

$$\curlyvee(r\,|(s_{11}, \ldots, s_{1i})\,) \text{ and } \ldots \text{ and } \curlyvee(r\,|(s_{n1}, \ldots, s_{nj})\,)$$

Going back to our examples, in the let case we need to check $\curlyvee(r\,|r_1 + r_2\,)$ which is then equivalent to $\curlyvee(r\,|r_1, r_2\,)$ while in the if case we check $\curlyvee(r\,|r \vee r\,)$, i.e. $\curlyvee(r\,|r\,)$ and $\curlyvee(r\,|r\,)$.

Lemma 3.1 *Let s, q, r be views and α_1 and α_2 algorithmic views. Then:*

1. *If $\curlyvee(s\,|\alpha_1 + \alpha_2\,)$ then $\curlyvee(s\,|\alpha_1\,)$ and $\curlyvee(s\,|\alpha_2\,)$.*
2. *If $\curlyvee(s\,|q, r\,)$, $\curlyvee(q\,|\alpha_1\,)$ and $\curlyvee(r\,|\alpha_2\,)$ then $\curlyvee(s\,|\alpha_1 + \alpha_2\,)$.*
3. *$\curlyvee(s\,|\alpha_1 \vee \alpha_2\,)$ iff $\curlyvee(s\,|\alpha_1\,)$ and $\curlyvee(s\,|\alpha_2\,)$.*

We are now ready to define the judgement $\Gamma^{\Psi} \overset{n}{\underset{n'}{\vdash}} e \leftleftarrows C^r$ inductively by the rules in Figure 2, where Γ, n, e and C^r are inputs and Ψ and n' are outputs. Γ is an FJEU context, i.e. a map from variable names to FJEU types. Ψ is a map from variable names to algorithmic views. The notation Γ^{Ψ} means that for every variable $x \in \Gamma$ if $\Gamma_x = C$ and $\Psi_x = \alpha$ then $\Gamma_x^{\Psi} = C^{\alpha}$. We use also the notation $\Gamma^{\Psi_1+\Psi_2}$ for meaning that if $\Gamma_x^{\Psi_1} = C^{\alpha_1}$ and $\Gamma_x^{\Psi_2} = C^{\alpha_2}$ then $\Gamma_x^{\Psi_1+\Psi_2} = C^{\alpha_1+\alpha_2}$. The meaning of $\Gamma^{\Psi_1 \vee \Psi_2}$ is similar. In the following we also write $\Gamma^{\mathcal{D}}$, where Γ is an FJEU context and $\mathcal{D}$ is a map from variable names to views to denote a RAJA$^+$ context. In summary, we define the function $\mathsf{typecheck}(\Gamma, e, C^r, n)$ by:

$$\mathsf{typecheck}(\Gamma, e, C^r, n) = \begin{cases} (\Psi, n') & \text{if } \Gamma^{\Psi} \overset{n}{\underset{n'}{\vdash}} e \leftleftarrows C^r \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

Next, we define the algorithmic judgement $\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}$ based on algorithmic typing. Similar to $\vdash m : \alpha\ \mathsf{ok}$, it means that α is a valid RAJA$^+$ type for m if the method body of m can be typed with the arguments, return type and effects as specified in α. For every argument type $E_i^{r_i}$, the typechecking algorithm returns an algorithmic view β_i, and it should hold $\curlyvee(r_i \,|\beta_i)$. Moreover, u' is also an output of the typechecking algorithm, and it should be greater or equal than the n' specified in α.

In the following we show that the algorithmic typing system we just defined is correct w.r.t. the declarative typing system of RAJA$^+$:

1. Sound: If $\Gamma^{\Psi} \overset{n}{\underset{n'}{\vdash}} e \leftleftarrows C^r$ and for all $x \in \Psi$ with $\Psi_x = \alpha$ one has $\curlyvee(s \,|\alpha)$ then $\Gamma^{\mathcal{D}} \overset{n}{\underset{n'}{\vdash}} e : C^r$, where $\mathcal{D}_x = s$.
2. Complete: If $\Gamma^{\mathcal{D}} \overset{n}{\underset{n'}{\vdash}} e : C^r$ and $u \geq n$ then for all $x \in \mathcal{D}$ with $\mathcal{D}_x = s$ there is an α with $\curlyvee(s \,|\alpha)$ and $\Gamma^{\Psi} \overset{u}{\underset{u'}{\vdash}} e \leftleftarrows C^r$ for some $u' \geq n'$, where $\Psi_x = \alpha$.

3.1 Verification of Soundness

Lemma 3.2 (Soundness of algorithmic RAJA$^+$ typing)
If $\Gamma^{\Psi} \overset{n}{\underset{n'}{\vdash}} e \leftleftarrows C^r$ and for all $x \in \Psi$ with $\Psi_x = \alpha$ there is a view s with $\curlyvee(s \,|\alpha)$ then $\Gamma^{\mathcal{D}} \overset{n}{\underset{n'}{\vdash}} e : C^r$, where $\mathcal{D}_x = s$.

Proof. By induction on algorithmic typing derivations, using the $(\Diamond^{+}Waste)$ rule in most cases, and Lemma 3.1 in the cases $(\vdash Let)$ and $(\vdash Conditional)$. □

Lemma 3.3 (Soundness of algorithmic RAJA$^+$ method typing) *Given a RAJA$^+$ class C, a view r, a method $m \in \mathsf{M}(C)$ and a RAJA$^+$ method type $\alpha \in \mathsf{M}(C^r, m)$, if $\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}$ then $\vdash m : \alpha\ \mathsf{ok}$.*

Proof. Let $\alpha = E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0}$ and $\curlyvee(r \,|q, s), \curlyvee(q \,|\beta), \curlyvee(r_1 \,|\beta_1) \ldots \curlyvee(r_j \,|\beta_j)$. We have

$$\mathtt{this}{:}C^{\beta}, x_1{:}E_1^{\beta_1}, \ldots, x_j{:}E_j^{\beta_j} \overset{n + \Diamond(C^s)}{\underset{u}{\vdash}} \mathsf{M}_{\mathsf{body}}(C, m) \leftleftarrows E_0^{r_0}$$

and $u \geq n'$ and we show

$$\mathtt{this}{:}C^{q}, x_1{:}E_1^{r_1}, \ldots, x_j{:}E_j^{r_j} \overset{n + \Diamond(C^s)}{\underset{n'}{\vdash}} \mathsf{M}_{\mathsf{body}}(C, m) : E_0^{r_0}$$

Algorithmic RAJA$^+$ Typing $\boxed{\Gamma^{\Psi} \vdash^{n}_{n'} e \leftleftarrows C^r}$

$$\frac{n \geq \Diamond(D^r) + \mathsf{Size}(D) \qquad D^r <: C^r}{\emptyset \vdash^{n}_{n-(\Diamond(D^r)+\mathsf{Size}(D))} \mathsf{new}\ D \leftleftarrows C^r}\ (\vdash New) \qquad \frac{E^q <: C^r}{x{:}E^q \vdash^{n}_{n} x^q \leftleftarrows C^r}\ (\vdash Var)$$

$$\frac{}{x{:}C^q \vdash^{n}_{n+\Diamond(C^q)+\mathsf{Size}(C)} \mathsf{free}(x^q) \leftleftarrows E^r}\ (\vdash Free)$$

$$\frac{D <: E \ \ (\text{or} \ \ E <: D) \qquad D^q <: C^r}{x{:}E^q \vdash^{n}_{n} (D)x^q \leftleftarrows C^r}\ (\vdash Cast) \qquad \frac{}{\varnothing \vdash^{n}_{n} \mathsf{null} \leftleftarrows C^r}\ (\vdash Null)$$

$$\frac{\mathsf{A}^{\mathsf{get}}(C^r, a) = q \qquad C.a = E \qquad E^q <: D^s}{x{:}C^r \vdash^{n}_{n} x^r.a \leftleftarrows D^s}\ (\vdash Access)$$

$$\frac{\mathsf{A}^{\mathsf{set}}(E^q, a) = s \qquad E.a = D \qquad F^p <: D^s \qquad E^q <: C^r}{x{:}E^q, y{:}F^p \vdash^{n}_{n} x^q.a \leftarrow y^p \leftleftarrows C^r}\ (\vdash Update)$$

$$\frac{\left(E_1^{q_1}, \ldots, E_j^{q_j} \xrightarrow{p/p'} E_0^{q_0}\right) \in M(D^s, m) \quad n \geq p \qquad \forall i\,.\, F_i^{t_i} <: E_i^{q_i} \quad E_0^{q_0} <: C^r}{x{:}D^s, y_1{:}F_1^{t_1}, \ldots, y_j{:}F_j^{t_j} \vdash^{n}_{p'+n-p} x^s.m\,(y_1^{t_1}, \ldots, y_j^{t_j}) \leftleftarrows C^r}\ (\vdash Invocation)$$

$$\frac{\Gamma_1^{\Psi_1}, \Delta^{\Psi'_3} \vdash^{n}_{n'} e_1 \leftleftarrows D^s \quad \Gamma_2^{\Psi_2}, \Delta^{\Psi''_3}, x{:}D^{\alpha} \vdash^{n'}_{n''} e_2 \leftleftarrows C^r \quad \curlyvee(s\,|\alpha)}{\Gamma_1^{\Psi_1}, \Gamma_2^{\Psi_2}, \Delta^{\Psi'_3+\Psi''_3} \vdash^{n}_{n''} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \leftleftarrows C^r}\ (\vdash Let)$$

$$\frac{\Gamma_1^{\Psi_1}, \Delta^{\Psi'_3} \vdash^{n}_{n'} e_1 \leftleftarrows C^r \qquad \Gamma_2^{\Psi_2}, \Delta^{\Psi''_3} \vdash^{n}_{n''} e_2 \leftleftarrows C^r}{x{:}D^s, \Gamma_1^{\Psi_1}, \Gamma_2^{\Psi_2}, \Delta^{\Psi'_3 \vee \Psi''_3} \vdash^{n}_{\mathsf{min}(n', n'')} \mathsf{if}\ x^s\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \leftleftarrows C^r}\ (\vdash Conditional)$$

Algorithmic RAJA$^+$ Method Typing $\boxed{\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}}$

$$\frac{\begin{array}{c} m \in \mathsf{M}(C) \qquad \alpha = E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in \mathsf{M}(C^r, m) \\ \curlyvee(r\,|q, s) \qquad \curlyvee(q\,|\beta) \qquad \curlyvee(r_1\,|\beta_1)\ldots\curlyvee(r_j\,|\beta_j) \\ \mathtt{this}{:}C^{\beta}, x_1{:}E_1^{\beta_1}, \ldots, x_j{:}E_j^{\beta_j} \vdash^{n+\Diamond(C^s)}_{u'} \mathsf{M}_{\mathsf{body}}(C, m) \leftleftarrows E_0^{r_0} \qquad u' \geq n' \end{array}}{\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}}$$

Figure 2: Algorithmic RAJA$^+$ Typing

We have by soundness of algorithmic typing (Lemma 3.2)

$$\frac{\texttt{this}{:}C^q, x_1{:}E_1^{r_1}, \ldots, x_j{:}E_j^{r_j} \;\vdash^{n+\Diamond(C^s)}_{u}\; \mathsf{M}_{\mathsf{body}}(C,m) : E_0^{r_0} \qquad u \geq n'}{\texttt{this}{:}C^q, x_1{:}E_1^{r_1}, \ldots, x_j{:}E_j^{r_j} \;\vdash^{n+\Diamond(C^s)}_{n'}\; \mathsf{M}_{\mathsf{body}}(C,m) : E_0^{r_0}}\ (\Diamond^+Waste)$$

□

3.2 Verification of Completeness

The completeness proof is a bit more complicated than the soundness proof. The reason for this is that we have eliminated the rules ($\Diamond^+Share$) and ($\Diamond^+Waste$) from the declarative system and we have to show that typing derivations that use these rules are still admissible in the algorithmic system. We need to prove the admissibility of these rules in extra lemmas.

First we prove the following lemma (Waste), which states the admissibility of the ($\Diamond^+Waste$) rule in the algorithmic system. Here is interesting to remark that the algorithmic views returned by the algorithm remain the same if we change the return type and the context, because they are calculated based on the views annotations of the expression.

Lemma 3.4 (Waste) *If $\Theta^\Psi \;\vdash^n_{n'}\; e \leftleftarrows D^s$ and $u \geq n$ and there is a $\Gamma <: \Theta$ and $D^s <: C^r$ then $\Gamma^\Psi \;\vdash^u_{u'}\; e \leftleftarrows C^r$ for some $u' \geq n' + u - n$.*

Proof. By induction on algorithmic typing derivations. □

The next lemma states the admissibility of sharing in the algorithmic system.

Lemma 3.5 (Share) *Let $\Gamma^\Psi, y : D^{\alpha_1}, z : D^{\alpha_2} \;\vdash^n_{n'}\; e \leftleftarrows C^r$ and let s, q_1, q_2 be views with $\curlyvee(s\,|q_1, q_2), \curlyvee(q_1\,|\alpha_1), \curlyvee(q_2\,|\alpha_2)$. Then $\Gamma^\Psi, x : D^\alpha \;\vdash^n_{n'}\; e[x/y, x/z] \leftleftarrows C^r$ for some α with $\curlyvee(s\,|\alpha)$.*

Proof. By induction on algorithmic typing derivations, using Lemma 3.1. □

Lemma 3.6 (Completeness of algorithmic RAJA$^+$ typing)
If $\Gamma^\mathcal{D} \;\vdash^n_{n'}\; e : C^r$ and $u \geq n$ then for all $x \in \mathcal{D}$ with $\mathcal{D}_x = s$ there is an α with $\curlyvee(s\,|\alpha)$ and $\Gamma^\Psi \;\vdash^u_{u'}\; e \leftleftarrows C^r$ for some $u' \geq n'$, where $\Psi_x = \alpha$.

Proof. By induction on typing derivations.

Case ($\Diamond^+Let$), ($\Diamond^+Conditional$) With the induction hypothesis, using Lemma 3.1.

Case ($\Diamond^+Share$) Using Lemma 3.5.

Case ($\Diamond^+Waste$) We have

$$\frac{\Theta^\mathcal{D} \;\vdash^m_{m'}\; e : D^s \quad n \geq m \quad n + m' \geq n' + m \quad \Gamma^\mathcal{E} <: \Theta^\mathcal{D} \quad D^s <: C^r}{\Gamma^\mathcal{E} \;\vdash^n_{n'}\; e : C^r}\ (\Diamond^+Waste)$$

We can apply the induction hypothesis and obtain a Ψ with $\Theta^\Psi \;\vdash^m_{m''}\; e \leftleftarrows D^s$ for some $m'' \geq m'$. By Lemma 3.4 we have $\Gamma^\Psi \;\vdash^u_{u''}\; e \leftleftarrows C^r$ and

$u'' \geq m'' + u - m$. Our goal is to show $u'' \geq n'$, so let us put all our information together:

$$u'' \geq m'' + u - m \tag{3.1}$$
$$m'' \geq m' \tag{3.2}$$
$$u \geq n \geq m \tag{3.3}$$
$$n + m' \geq n' + m \tag{3.4}$$

We can derive:

$$\text{from 3.2 and 3.4 we get } m'' \geq m' \geq n' + m - n \tag{3.5}$$
$$\text{from 3.1, 3.3 and 3.5 we get } u'' \geq (n' + m - n) + u - m \tag{3.6}$$
$$\text{3.6 simplifies to } u'' \geq n' + u - n \tag{3.7}$$

And since $u \geq n$ by assumption, $u - n \geq 0$, thus, $u'' \geq n'$.

□

Lemma 3.7 (Completeness of algorithmic RAJA$^+$ method typing) *Given a RAJA$^+$ class C, a view r, a method $m \in \mathsf{M}(C)$ and a RAJA$^+$ method type $\alpha \in \mathsf{M}(C^r, m)$, if $\vdash m : \alpha$ ok then $\vdash_{\mathsf{a}} m : \alpha$ ok.*

Proof. Let $\alpha = E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0}$ and $\curlyvee(r\,|q, s\,)$. We have

$$\texttt{this}{:}C^q, x_1{:}E_1^{r_1}, \ldots, x_j{:}E_j^{r_j} \;\vdash^{\frac{n\,+\,\Diamond(C^s)}{n'}}\; \mathsf{M}_{\mathsf{body}}(C, m) : E_0^{r_0}$$

and we show that there are algorithmic views $\alpha, \beta_1, \ldots, \beta_j$ with $\curlyvee(q\,|\alpha\,)$, $\curlyvee(r_1\,|\beta_1\,)$, …, $\curlyvee(r_j\,|\beta_j\,)$ and

$$\texttt{this}{:}C^\alpha, x_1{:}E_1^{\beta_1}, \ldots, x_j{:}E_j^{\beta_j} \;\vdash^{\frac{n\,+\,\Diamond(C^s)}{u'}}\; \mathsf{M}_{\mathsf{body}}(C, m) \leftleftarrows E_0^{r_0}$$

for some $u' \geq n'$, and it follows by Lemma 3.6. □

3.3 Decidability of RAJA$^+$ typing

Lemma 3.8 (Decidability of algorithmic RAJA$^+$ typing) *$\Gamma^\Psi \vdash^{\frac{n}{n'}} e \leftleftarrows C^r$ is decidable.*

Proof. The function $\mathsf{typecheck}(\Gamma, e, C^r, n)$ defined before decides the judgement evidently. It is possible to define such a function because the rules are deterministic and syntax-directed. □

Lemma 3.9 *Given a RAJA$^+$ class C, a view r, a method $m \in \mathsf{M}(C)$ and a RAJA$^+$ method type $\alpha \in \mathsf{M}(C^r, m)$, $\vdash m : \alpha$ ok is decidable.*

Proof. By Lemma 3.3 and Lemma 3.7 together we have that $\vdash m : \alpha$ ok is equivalent to $\vdash_{\mathsf{a}} m : \alpha$ ok, and $\vdash_{\mathsf{a}} m : \alpha$ ok is decidable because $\texttt{this}{:}C^\beta, x_1{:}E_1^{\beta_1}, \ldots, x_j{:}E_j^{\beta_j} \;\vdash^{\frac{n\,+\,\Diamond(C^s)}{u}}\; \mathsf{M}_{\mathsf{body}}(C, m) \leftleftarrows E_0^{r_0}$ is decidable by Lemma 3.8. □

Theorem 3.10 (Decidability of RAJA$^+$ typing) *Given a RAJA$^+$-Program $\mathscr{R}$, it is decidable if it is well-typed.*

Proof. We have to show that the three conditions to check are decidable. Conditions 1. and 2. involve subtyping which is clearly decidable. Details about its implementation are found below. $\forall \alpha \in \mathsf{M}(C^r, m)\ .\ \vdash m : \alpha\ \mathsf{ok}$ is decidable by Lemma 3.9. □

4 Ocaml Implementation

In the last section we have presented a type checking algorithm for RAJA$^+$ programs and proved its correctness. This algorithm has been implemented in Ocaml. In this section we show some interesting aspects of the implementation.

Our tool include not only the typechecker, but also a parser and an interpreter for RAJA$^+$ programs. The interpreter is based on the runtime semantics of the system, which are standard. This way we have a complete environment for experimenting with RAJA$^+$ programs.

4.1 Implementation of Coinductive Definitions

One interesting aspect of the implementation is the way we implemented the coinductive definitions of subtyping and sharing. They have been implemented with an algorithm for computing greatest fixpoints, defined and proved correct in [Pie02, Ch. 21], which works for coinductive definitions that fall into a specific scheme, i.e. a goal is *supported* by a set of subgoals in a deterministic way. The idea of the algorithm is to maintain a list of assumptions. Every goal is kept in this list, unless some condition is not fulfilled. Afterwards the algorithm is called with all the subgoals. If a given subgoal is an element of the list of assumptions (which means it has been a goal before) then it is assumed to be already in the coinductive defined relation. The soundness of the algorithm is not trivial, but it has been proved.

The definition of subtyping of RAJA types (Definition 2.1) does not fall completely into this scheme unfortunately, because the condition (2.9) contains an existential quantifier. Thus, we do not have a set of subgoals, but a boolean combination of subgoals. This is a more general scheme and we aim at finding an extension of the algorithm that deals with it.

In the meantime we regard subtyping of RAJA method types (Definition 2.2) as another coinductive definition, mutual with subtyping of RAJA types. This way we do not get any subgoals from (2.9) but we regard it as one of the ground conditions. The implementation uses mutual recursive functions.

4.2 Examples

We have successfully implemented the examples of RAJA$^+$ code found in [HJ06], i.e. copy of lists and doubly linked lists as well as more interesting algorithms like mergesort. In the following we show some examples of code and explain the syntax. This is a fragment of the copy example.

```
views a, b, c, d, n, e
class List implements a, b, c, d, n {
    ...
}
class Cons extends List implements a, b, c, d, n {
    a: pot = 1;
    b: pot = 0;
    c: pot = 0;
    d: pot = 1;
    n: pot = 0;
    a: List<a, a> next;
    b: List<b, b> next;
    c: List<a, a> next;
    d: List<n, a> next;
    n: List<n, a> next;

    a as <c, d> : List<b>,0 copy(0) {
       let Cons<b> res = new Cons in
       let List<a> next = [this as c].next in
       let List<b> nres = [next as a].copy() in
       [res as b].next <- [nres as b] in
       return [res as b];
    }
}
class Main implements e {
  e: pot = 0;
  e as <e,e> : List<b>,m : main(m + 2*k + 1) {
     \\create a list of type List<a> l of length k
     return l.copy();
  }
}
```

The construction "Cons inherits List implements a, b, ..." means that the class Cons inherits from the class List as in Java, and the class is refined with the already defined views a, b, etc. For every view then it is necessary to define a potential with the construction "a : pot = 1;" . This means $\Diamond(Cons^a) = 1$.

For every field we need to define a get and a set view with "d : List<n, a> next;" which means $\mathsf{A}^{\mathrm{get}}\Big(\mathsf{Cons}^d, \mathsf{next}\Big) = n$, $\mathsf{A}^{\mathrm{set}}\Big(\mathsf{Cons}^d, \mathsf{next}\Big) = a$. Refined types (like Cons^a) are written with the syntax Cons<a>. The construction "a as <c, d> : List<b>, 0 copy(0) " means $\curlyvee(a\,|c, d)$ as in the rule for typing a method body, and the refined type of the method copy in Cons^a is $\mathsf{M}(\mathsf{Cons}^a, \mathsf{copy}) = \ . \xrightarrow{0/0} \mathsf{List}^b$. The construction [this as c] means the annotation of the variable "this" with the view "c".

In this prototype implementation we do not allow runtime arguments to the program for reasons of simplicity. Instead, we have to define a main method that will be executed without any arguments. That means that the arguments have to be defined inside of this method.

In the case of the copy example, we define in the main method a list of the desired length and create a copy of it by calling the respective method. Since the potential of a list of type List<a>is 1, for each node we create we need two units of potential: one for the actual object creation, and one for the potential that will be spend afterwards when copying the list. This explains why we need $2*k$ units for a list of length k. We need another unit for creating a nil node, whose potential is 0. Thus, our program will run successfully with a freelist of $2*k+1$ units or more. If the freelist contains m more units, they will remain free after program execution. Any correct instantiation of this scheme will be accepted by the typechecker.[2] Then, it will predict an upper bound of the freelist size needed by the program. In this case it will be $2*k+1$, i.e. prediction and actual cost will coincide. But in other cases, like if we create such a list and we do not call the copy method, the actual cost will be $k+1$ and the prediction will still be $2*k+1$, an upper bound of the actual cost.

5 Conclusions

We have provided a type checking algorithm for RAJA$^+$ programs and we have proved its correctness. This shows decidability of the typing system. Moreover, we have presented a prototype implementation in Ocaml based on this algorithm. The implementation has been successfully tested with code examples like a method for copying lists, sorting algorithms, etc.

This implementation is based on the RAJA typing system that has been described and proved sound in [HJ06]. However, we believe that this system can be improved. For instance, the subtyping definition we present here allows to type more programs than the one in [HJ06]. Similarly, extending the sharing definition to RAJA types will allow more flexible typing rules. We want to extend the proof of soundness to these new features in the future.

Moreover, as discussed in the last section, we aim at finding an extension of the algorithm for implementing coinductive definitions that fits in the scheme needed for the definition of subtyping of RAJA types, i.e. when we have a boolean combination of subgoals. On the other hand, inference of views and annotations remains under investigation.

References

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 171–183, New York, January 1998. Association for Computing Machinery.

[HBH+07] Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert Pointon. Automatic amortised worst-case execution time analysis. In *7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, Proceedings*, pages 13–18, 2007.

[2] It is important to remark that these equations are only for explanation, the typechecker expects actual numbers since no kind of inference has been implemented yet.

[HDF+05] Kevin Hammond, Roy Dyckhoff, Christian Ferdinand, Reinhold Heckmann, Martin Hofmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert F. Pointon, Norman Scaife, Jocelyn SÃl'rot, and Andy Wallace. The embounded project (project start paper). In Marko C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 195–210. Intellect, 2005.

[HJ03] Hofmann and Jost. Static prediction of heap space usage for first-order functional programs. In *POPL: 30th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[HP99] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space:, June 21 1999.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

REM 2007

Monitoring External Resources in Java MIDP

David Aspinall Patrick Maier[1] Ian Stark

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, United Kingdom

Abstract

We present a Java library for mobile phones which tracks and controls at runtime the use of potentially costly resources, such as premium rate text messages. This improves on the existing framework (MIDP — the Mobile Information Device Profile [6]), where for example every text message must be authorised explicitly by the user as it is sent. Our resource management library supports richer protocols, like advance reservation and bulk messaging, while maintaining the security guarantee that attempted resource abuse is trapped.

Keywords: Runtime Monitoring, Resource Control, Java MIDP, Security.

1 Introduction

Modern mobile phones are powerful computers. Their primary task, providing mobile wireless telephone services, is comparatively losing importance as they are being used for a range of other applications, from personal information managers to web browsers, from media players to games. Most of these applications access the network [2], either because it is integral to their functionality (e. g. web browsers, online games), or because networking is adding desired features (e. g. playing streaming media or synchronising diaries).

The cost of the standard computational resources, like execution time or memory space, is determined solely by the computational device (i. e. the hardware of the mobile phone) itself. The cost of network access, however, is determined by external entities, e. g. the business model of the phone operator, which is why we classify network access as an *external resource*. Moreover, it is a resource the spending of which users generally would like to control tightly because it costs them money. The last point actually goes double: If network access is maliciously exploited it could be very expensive, but even if it is not exploited, users care about each 10p [3], i. e. they want to know the exact cost beforehand.

In MIDP [6], the current standard framework for Java applications on mobile phones, monitoring external resources, like communication via text message, is left to the user, as

[1] Email: pmaier@inf.ed.ac.uk

[2] Refers to the operator's mobile phone network; access to other networks (like the Internet) is routed through this one.

[3] The standard cost of sending a text message in the United Kingdom.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

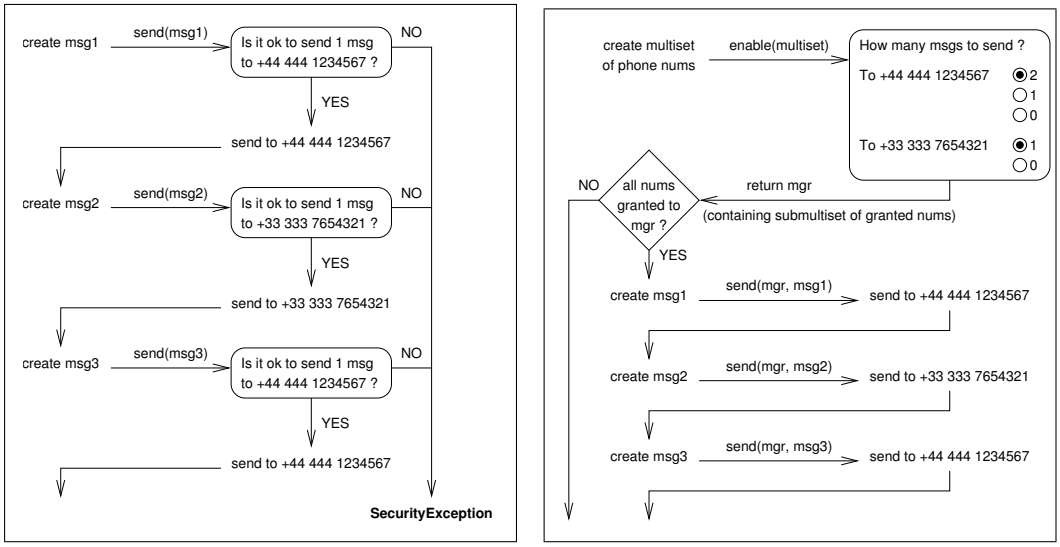


Fig. 1. Transaction sending 3 text messages; in MIDP 2.0 (left) and with explicit resource management (right).

illustrated by the flowchart on the left hand side of Figure 1. For each of the three messages, the application pauses to ask the user for authorisation before sending. This one-shot authorisation is clearly prohibitive for applications wishing to send many messages because users will get annoyed by the many pop-up screens, which malicious applications may exploit to trick users into authorising messages to premium rate numbers. Such social engineering attacks [12] have been reported in the wild [14]. Yet, even if an application sends only few messages, one-shot authorisation can lead to undesirable results, like transactions aborted midway by an exception because the user stops authorising messages (see the left hand side of Figure 1).

We propose explicit accounting and monitoring of external resources to better protect the user from accidental or malicious resource abuse. Our approach revolves around *resource manager* objects, which keep an account of which external resources an application is granted to use and how often. The right hand side of Figure 1 illustrates this on the messaging example. Before sending messages, the application computes a multiset of phone numbers encoding how many messages it will send to which recipients. In a single authorisation dialogue the user then gets to decide how many messages the applications may send to whom. This information (a submultiset of the multiset of requested numbers) is stored in a resource manager. The application only proceeds if all the requested numbers have actually been granted, in which case it calls instrumented methods for sending the messages, taking an extra resource manager argument, which monitors the resources being spent (and would abort the application if it was overspending).

Explicit resource management has additional benefits besides runtime monitoring. It forces the application to determine early on how many resources to request. It provides a clear user interface by centralising the choice of which of the requested resource to grant into a single dialogue. Plus, it enables the application to react flexibly to the amount of resources it has been granted, i. e. the application can choose whether it is feasible to continue with the resources granted or whether it has to abort because of insufficient resources.

The rest of this paper is structured as follows. Section 2 gathers some facts about MIDP which are relevant to us. Section 3 introduces the resource management library,

which Section 4 extends by adding policies. Section 5 describes the security properties that library guarantees and outlines a deployment scenario. Section 6 discusses related work, and Section 7 concludes.

2 Background: The MIDP Security Model

The *Mobile Information Device Profile* (MIDP, current version 2.0 [6]) is the current standard framework for Java applets (also called *MIDlets*) on networked mobile devices. MIDP builds upon the *Connected Limited Device Configuration* (CLDC, current version 1.1 [7]). Together, CLDC and MIDP, which are part of the *Java Micro Edition Platform* (Java ME), define a set of APIs for programming small devices like phones and PDAs. With security in mind, they restrict Java in several ways. In particular, reflection and custom class loading are not supported; all of a MIDlet's classes must be loaded from a single JAR using the standard CLDC class loader, which renders possible to statically check the MIDlet's classes for certain properties (see Section 5.4).

As of MIDP 2.0, access to sensitive APIs and functions (e. g. for sending text messages) is regulated by a permission-based security model. MIDlets are bound to *protection domains* based on whether and by whom they are signed (where a signature expresses the signer's trust in the MIDlet but does not provide any guarantees about the code itself). Each protection domain holds a set of *permissions*, each of which is either flagged as *Allowed* or *User*. The former grants unconditional access whereas the latter requires access to be authorised by the user. How often this authorisation has to be obtained depends on whether a *User* permission is flagged as *Blanket*, *Session* or *OneShot*; the latter requires authorisation for every single access.

According to the MIDP specification, only MIDlets signed by the device manufacturer or the network operator may obtain unconditional access to cost-sensitive functions (e. g. for sending text messages). The protection domains for other MIDlets must insist on OneShot authorisation for access to these functions. As a consequence, MIDlets wishing to use messaging more than just occasionally are faced with the choice of either having to be signed by the operator (or manufacturer) or having to annoy their users with lots of authorisation screens.

3 Basic Resource Management API

This section presents an API for monitoring the use of external resources. The API introduces special objects, called *resource managers*, which encapsulate multisets of resources that a MIDlet may legally use (according to the user's approval) and which are passed as arguments into instrumented MIDP methods that actually use the resources. These methods, e. g. the method for sending text messages, check the resource manager before consuming the resources. If the required resources are not present, the instrumented methods abort the MIDlet with a runtime error.

3.1 Resource Managers

Figure 2 shows a class diagram of resource management package. The core of the API is the final class `ResManager`, which encapsulates a multiset of resources and whose meth-

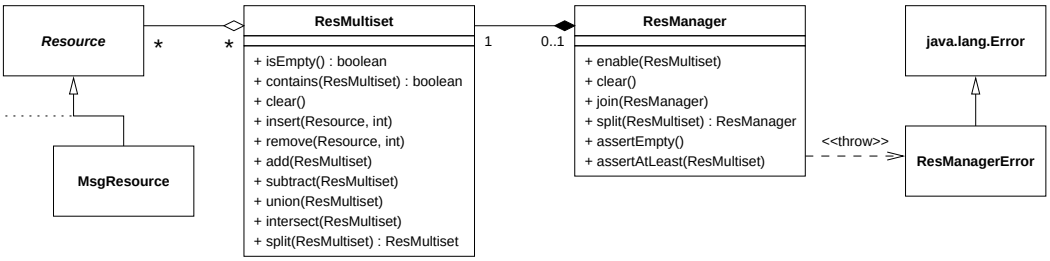


Fig. 2. UML class diagram of the basic resource management API. All terminal (w. r. t. generalisation) classes are final.

ods are explained below. The final class `ResMultiset` provides modifiable multisets of resources, with the usual operations on multisets, including multiset intersection, sum and inclusion. Internally, multisets are realised by hash tables, mapping resources to multiplicities (which may be infinite). Every `ResMultiset` object encapsulates its mutable state, so that it cannot be changed other than by calling its public methods. The abstract class `Resource` serves as an abstract type for resources; actual resources (e. g. the class `MsgResource` representing the permission to send one text message to a given phone number) must be final subclasses. Being used as keys in hash tables, resources must abide by the following contract: They must be immutable objects, and resources constructed from the same arguments must be indistinguishable by the `equals` method.

The class `ResManager` encapsulates a multiset of resources via a private field `rs` of type `ResMultiset`. All public methods are synchronised to avoid races in case different threads access the same resource manager. The table below lists the methods with a JML-style [4] semantics, where the symbols $\subseteq$, $\uplus$ and $\cap$ stand for multiset inclusion, sum and intersection, respectively.

	requires	**ensures**	**modifies**
ResManager()	*true*	this.rs $= \emptyset$	this.rs
void enable(ResMultiset req)	*true*	this.rs $\uplus$ req $=$ \old(this.rs) $\uplus$ \old(req) $\wedge$ req $\subseteq$ \old(req)	this.rs, req
void clear()	*true*	this.rs $= \emptyset$	this.rs
void join(ResManager mgr)	*true*	this.rs $=$ \old(this.rs) $\uplus$ \old(mgr.rs) $\wedge$ mgr.rs $= \emptyset$	this.rs, mgr.rs
ResManager split(ResMultiset bound)	*true*	\fresh(\result) $\wedge$ \result.rs $=$ \old(this.rs) $\cap$ bound $\wedge$ \result.rs $\uplus$ this.rs $=$ \old(this.rs)	this.rs
void assertEmpty()	this.rs $= \emptyset$	*true*	\nothing
void assertAtLeast(ResMultiset bound)	bound $\subseteq$ this.rs	*true*	\nothing

The `enable` method takes a multiset `req` of requested resources and lets the user decide (in a pop-up dialogue) how many of these resources to add to the manager's multiset `rs`. As a side effect, `enable` modifies its argument `req`; upon return from `enable`, the MIDlet should check `req` to learn which of the requested resources it is being denied; in particular, if `req` is empty then all of the requested resources have been granted.

The methods `clear`, `split` and `join` provide some control over the contents of a resource manager, by consuming all its resources, transferring some resources to a new

[4] The \operators generally bear the same meaning as in JML [11], except that `\old(e)` refers to the pre-state of expression `e` in the *pre*-state of the heap.

```
void sendBulk(MessageConnection conn,
              Message msg,
              PhonebookEntry[] grp)
{
  ResMultiset rs = new ResMultiset();
  for (int i=0; i < grp.length; i++) {
    String num = grp[i].getMobileNum();
    rs.insert(new MsgResource(num), 1);
  }

  ResManager mgr = new ResManager();
  mgr.enable(rs);

  if (rs.isEmpty()) {
    for (int i=0; i < grp.length; i++) {
      String num = grp[i].getMobileNum();
      msg.setAddress(num);
      conn.send(mgr, msg);
    }
    mgr.assertEmpty();
  }
  else mgr.clear();
}
```

```
public void send(ResManager mgr, Message msg)
throws IOException, InterruptedIOException
{
  synchorized (msg) {
    String num = msg.getAddress();
    ResMultiset rs = new ResMultiset();
    rs.insert(new MsgResource(num), 1);

    ResManager local_mgr = mgr.split(rs);
    local_mgr.assertAtLeast(rs);

    try {
      send(msg);
      local_mgr.clear(); local_mgr = null;
    } catch (InterruptedIOException e) {
      local_mgr.clear(); local_mgr = null;
      throw e;
    } catch (IOException e) {
      mgr.join(local_mgr); local_mgr = null;
      throw e;
    }
  }
}
```

Fig. 3. Bulk messaging example, left: MIDlet code, right: instrumented MIDP method.

manager, or joining the resources in two managers, respectively. Thanks to `split` and `join`, the MIDlet may keep resource managers thread local, avoiding contention over shared managers.

The assertion methods check whether their preconditions hold. If so they behave like no-ops, otherwise they throw an instance of `ResManagerError`. The latter case must be seen as a violation of the MIDlet's own logic (much like failing an assertion), and the MIDlet should not be allowed attempts at repairing the situation (by catching the error), which is why `ResManagerError` extends `java.lang.Error` rather than `java.lang.Exception`.

3.2 Example: Bulk Messaging MIDlet

We illustrate the use of resource managers by an example application built on top of the *Wireless Messaging API* (WMA, current version 2.0 [8]), a bulk messaging MIDlet, which lets the user send a text message to a group of recipients from his phone book. Figure 3 (left column) shows the MIDlet's method that actually sends the message. The method takes an (already open) message connection, a message and a group of recipients (represented as array of phone book entries). First, the MIDlet builds up a multiset of resources `rs` by iterating over the group of recipients and for each one, extracting the mobile phone number, converting it into a resource by constructing an instance of `MsgResource`, and adding one occurrence of that instance to the multiset. Next, the MIDlet creates an empty resource manager `mgr` and enables it to use the resources in the multiset `rs`. This will pop up a confirmation dialogue box where the user can approve or deny the planned resource usage, modifying `rs` as a side effect. Only if the user approves of all messages to be sent, i. e. if `enable` returns its argument `rs` empty, does the code proceed to the actual send loop. The send loop again iterates over the group of recipients, extracting for each one the mobile phone number, setting the address field of the message and sending the message using the instrumented `send` method, see below. After the loop, `assertEmpty` checks that the resource manager `mgr` is really empty, i. e. all enabled resources have been used. (Instead of checking, the manager could have been cleared explicitly, like in the else branch, to prevent unintended later use of left-over resources.)

3.3 Instrumented Methods

Resources are consumed by specific methods, e. g. in the case of messaging by the method `send(Message)` declared in the WMA interface `MessageConnection`. To monitor whether these methods consume only resources that have been granted, we wrap them with instrumentation code checking whether a given resource manager holds the required resources. These instrumented methods are declared in sub-packages of the resource management package.

To instrument messaging, we have to augment MIDP and WMA in three places. We supplement the WMA interface `MessageConnection` with a new wrapper method `send(ResManager,Message)`, provide a class which implements this extended interface, and revise the MIDP method `Connector.open` to return the new class.

The code for the wrapper method is shown on the right-hand side of Figure 3. It extracts the phone number `num` from the message and constructs a multiset `rs` containing a single occurrence of the resource corresponding to `num`. Then it splits the resources in `rs` off from the resource manager `mgr` and stores them in the new local resource manager `local_mgr`, which is checked for containing at least the resources in `rs`. If this check fails a `ResManagerError` will be thrown, aborting the calling MIDlet; if the check succeeds we know that `local_mgr` holds exactly the resources in `rs`. Finally, the message is actually sent by calling the uninstrumented `send` method. [5] Clearing `local_mgr` and nulling the reference afterwards is not strictly necessary but considered good practise; it signals that the resources in the local manager are now used up and that the manager itself is ready to be reclaimed by garbage collection.

In case of a send failure, the event that actually spends the resources (i. e. delivering the text message to the operator's network) may or may not have happened yet. We assume that an `IOException` is thrown before actually sending the message (e. g. because the connection to the operator's network is down), so the resources are not yet consumed, and the handler can return them to the caller (by joining the local manager to `mgr`) before propagating the exception. However, if an `InterruptedIOException` is raised, we do not know whether the send event has already happened, so we assume that the resources are already spent. In this case, the handler consumes the resources (by clearing the local manager) before propagating the exception.

Note that the instrumented `send` method method must synchronise on `msg`, which is accessed twice, but there is no need to synchronise on `mgr` (for there are no data dependencies between the first and second access) or on **`this`** (for it is accessed only once).

3.4 Runtime Overhead

Monitoring of external resources does cause some runtime overhead. In terms of execution time, the overhead is negligible, as very little time is spent on the instrumentation compared to what is spent on actually consuming the resource (e. g. transmitting a message). Due to the hash table based implementation of multisets, all operations on resource managers take (at most) linear time w. r. t. to the size of the multisets involved. In fact, the overhead of the instrumented send method in Figure 3 is constant because the argument of `assertAtLeast` is a singleton multiset.

[5] Depending on the MIDlet's protection domain, the uninstrumented `send` method may again ask the user to authorise sending the message; Section 5.4 addresses this shortfall.

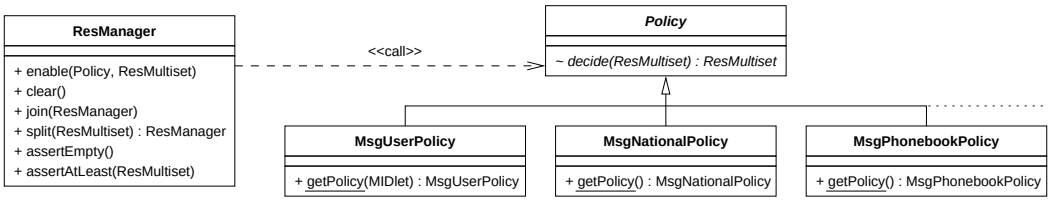


Fig. 4. UML class diagram of policy extension of resource management API. All terminal classes are final.

In terms of memory, the overhead may be more severe, particularly on small devices, because of the memory requirements of the hash tables. Additionally, resource monitoring puts a higher strain on garbage collection because the instrumentation code temporarily allocates resources, multisets and managers. If runtime checking is not necessary or desired, it can be switched off by "erasing" resource managers (see Section 5.2), which reduces the memory overhead significantly.

3.5 Extensibility

By design, the resource management API is extensible. Monitoring new resources (e. g. the number of bytes sent over a TCP/IP connection, or the space available in the persistent record store) simply amounts to adding new resource types plus adding the appropriate instrumentation. New resource types are added by extending the abstract class `Resource` with final subclasses, which abide by the contract on resources. Instrumented methods, which monitor the new resources before calling the corresponding uninstrumented methods, are added to sub-packages of the resource management package.

4 Extending the API with Flexible Policies

So far, the `enable` method involves the user, who is selecting to-be-added resources in a pop-up dialogue. That is, the user is acting as a *policy oracle* deciding which resources to grant and which to deny. In this section, we extend the API to include more flexible policy oracles, not just the user.

4.1 Changes to the API

Figure 4 shows the class diagram of the extension. It adds an abstract class `Policy` providing an abstract, package private method `decide` for deciding which resources to grant and which to deny. The table below shows the formal, non-deterministic semantics of `decide`; granted resources are returned in a new multiset, denied resources are returned via the modified argument.

	requires	**ensures**	**modifies**
ResMultiset decide(ResMultiset req)	$true$	\fresh(\result) $\wedge$ \old(req) $=$ req $\uplus$ \result	req

Actual policies must be final subclasses of `Policy` and must provide a package private implementation of `decide`. The latter requirement ensures that `decide` can be called by the resource management library only, not directly by MIDlets themselves. For a MIDlet to gain access to policies, each subclass of `Policy` provides a static `getPolicy` method

which hands out the requested policy (i. e. an instance of the respective class) or **null** if the calling MIDlet is not authorised to use the requested policy.

MIDlets can only pass policies as arguments to other methods, in particular to the `enable` method of class `ResManager`, which consults its policy argument as an oracle to decide which resources to grant and which to deny, and which interprets a **null** argument as the deny-all policy, see the implementation below. Note that `enable` defers synchronisation on **this** as long as possible (i. e. until accessing the manager's encapsulated multiset `rs`) to avoid locking the manager during a call to `decide`, which may block for a long time (e. g. if the policy consults the user).

```
public void enable(Policy p, ResMultiset req) {
  if (p == null) return;
  synchronized (req) {
    ResMultiset granted = p.decide(req);
    synchronized (this) { rs.add(granted); }
  }
}
```

4.2 *Use of Policies in MIDlets*

The basic resource management API knew only one implicit policy: ask the user. Yet, typically each resource type has its own policy or policies. The policies for `MsgResource` include a `MsgUserPolicy`, which behaves like the implicit policy of the basic API, asking the user how many messages to send to which phone numbers. To use this policy, the call `mgr.enable(rs)` in the bulk messaging MIDlet (Figure 3) must be replaced by `mgr.enable(MsgUserPolicy.getPolicy(this), rs)`.[6]

There could be other policies for `MsgResource`, e. g. a `MsgNationalPolicy`, which grants only messages to national phone numbers. This policy could be combined with `MsgUserPolicy` by chaining calls to `enable` as in the following code snippet.

```
mgr.enable(MsgNationalPolicy.getPolicy(), rs);
mgr.enable(MsgUserPolicy.getPolicy(this), rs);
```

The first call enables all requested messages to national numbers, without asking the user. The second call asks the user to authorise the messages to the remaining (international) numbers. In the end, `rs` contains only those international numbers that the user has denied.

Another interesting policy for messaging could be a `MsgPhonebookPolicy`, which automatically grants all messages to numbers in the user's phone book. If the bulk messaging MIDlet used this policy, the user would not have to confirm anything. In return, the MIDlet could maliciously send more messages than the user intended, but only to phone numbers in the user's phone book, not to premium rate numbers (unless the MIDlet was allowed to modify the phone book).

4.3 *Extensibility*

By design of the API, adding new policies simply amounts to extending the abstract class `Policy` with final subclasses, which abide by the contract on policies: No public fields and methods (in particular, `decide` must be package private) except the static `getPolicy` methods, and the implementation of `decide` must agree with the formal semantics as shown in the table in Section 4.1.

[6] `MsgUserPolicy.getPolicy` requires an argument of type `MIDlet` so that the policy can access the MIDlet's screen.

5 Security Properties of Explicit Resource Management

This section informally summarises and motivates the security guarantees provided by the resource management API and a trusted library implementing it.

5.1 No Abuse of Resources

Property 1 *MIDlets using the resource management API cannot consume more resources than granted; any attempt to do so will result in the MIDlet being aborted before the abuse happens.*

The property holds for two reasons.

(i) Before performing any actions, the instrumented methods, e. g. the `send` method from Section 3.3, check their `ResManager` argument for the required resources and throw a `ResManagerError` (which will abort the MIDlet) if there aren't enough. If there are enough resources, the instrumentation deduces the required amount from the resource manager, even if the underlying uninstrumented method throws an exception.

(ii) The implementation of the resource management API ensures that policies cannot be bypassed. Resources may be moved back and forth between managers by the methods `split` and `join`, but there is no way to sneak new resources into the managers other than by calling `enable`, in which case a policy gets to decide which resources to grant and which to deny. Furthermore, the implementation confines the multiset held by a manager, i. e. it ensures that there are no pointers from outside a manager into its mutable state, hence a manager's multiset cannot be modified from the outside.

Of course, the above argument assumes that the MIDlet does not bypass or subvert the resource management library itself; see Section 5.4 on how to ensure this.

5.2 Erasure

Tracking the use of resources with resource managers does induce some overhead, mainly in terms of the memory required for storing the multisets. On small devices, one might want to avoid this overhead if a MIDlet is known to be *resource safe*, i. e. if it cannot ever throw a `ResManagerError`. In this case, resource managers can be "erased".

Erasure cannot be performed as a simple source code transformation removing all occurrences of resource managers from a MIDlet, for two reasons. First, MIDlets must be able to access resource managers in order to call the `enable` method, even after erasure, to let a policy decide which resources to grant. Second, resource managers may appear in conditions like `(mgr1 == mgr2)`, from where they cannot be removed unless the condition can be evaluated statically. What can be done, however, is a "soft" erasure, which keeps the managers themselves in place but erases their multisets, resulting in very lightweight *erased* resource managers.

Soft erasure can be achieved by retaining the public interface of class `ResManager` but replacing its implementation with a stateless dummy implementation. More precisely, erasure removes the private field `rs` (storing the manager's multiset), which turns all public methods into no-ops, except for `split` and `enable`. The latter still calls the policy and reports the denied resources back to the MIDlet, whereas the former creates a fresh (erased) manager.

Property 2 *If a MIDlet is resource safe then erasing the resource managers does not change its observable behaviour.*

The property holds because by design of the resource management API, the value of a resource manager can only affect the values of other resource managers; it cannot affect the values of other types.

Note that an optimiser can eliminate all of the calls on erased resource managers, except calls to `enable`, by inlining. As a result, resource managers may become unused and can be optimised away. In fact, a clever optimiser could optimise away the entire instrumentation code from the instrumented `send` method in Figure 3, leaving just the call of the uninstrumented method.

5.3 Information Flow Security

It may seem as if resource managers could infringe information flow security. Is it not possible that sensitive data (e. g. phone numbers from the address book) leaks from a manager while it is passed from method to method? We argue that at least for resource safe MIDlets, this is not the case.

Property 3 *If a MIDlet is resource safe then its resource managers do not leak information.*

This is a corollary of Property 2. If a MIDlet is resource safe, the resource managers can be erased without changing the MIDlet's observable behaviour. Yet, erased resource managers are stateless, so they cannot leak information. Hence, no leakage is observable.

5.4 Secure Deployment

As mentioned in Section 5.1, the security guarantees do not only depend on the correctness of the resource management library itself but also on the MIDlet correctly using the API (i. e. not bypassing or subverting the library).

Property 4 *Correct use of the resource management API can be checked statically by inspecting the MIDlet's JAR only.*

The property holds due to the restrictions imposed by CLDC and MIDP (see Section 2), which imply that all of the MIDlet's classes are statically known (since all classes must be loaded from a single JAR) and the signature of each method call is statically known (since reflection is not supported). Thus, the following properties of the MIDlet's class files can be statically checked.

- The MIDlet does not bypass the instrumentation. More precisely, if the MIDlet allocates a particular resource type (e. g. `MsgResource`) then it does not call uninstrumented methods for consuming resources of that type (e. g. the method `send(Message)` declared in the WMA interface `MessageConnection`).
- The MIDlet does not suppress failing assertions. More precisely, it does not catch `ResManagerError` or any of its superclasses.
- The MIDlet does not pass policies of its own to the `enable` method. More precisely, none of the MIDlet's classes extend the abstract class `Policy`.
- The MIDlet does not subvert the implementation of resource multisets by adding re-

source types of its own.[7] More precisely, none of the MIDlet's classes extend the abstract class `Resource`.

- The MIDlet does not exploit non-public methods (e. g. `decide`) of the resource management library. More precisely, none of the MIDlet's classes are declared to be part of the packages that constitute the resource management library.

The correctness of the resource management library itself cannot be checked easily, hence the library (including the instrumented methods) has to be trusted. Yet, as MIDP does not support the download of trusted libraries, MIDlets using the resource management API have to provide the library as part of their own JAR. To establish trust in the library, a trustworthy third party (e. g. the network operator) should vouch for it by signing the MIDlet. In detail, the deployment process should comprise the following steps.

(i) In the MIDlet's JAR, the signer replaces the untrusted resource management library with its own trusted implementation.

(ii) The signer checks for correct use of the resource management API by checking the above properties.

(iii) The signer signs and deploys the MIDlet (possibly after it passed other checks, too).

The signer may choose to erase resource managers by replacing the resource management library with the library for erased managers (see Section 5.2) if there is additional confidence in the MIDlet's resource safety (where this confidence may have been gained by type checking, extended static checking, interactive verification or extensive testing). Of course, Property 1 is not guaranteed by the library for erased managers.

There is a reason, why MIDlets should be signed by the network operator (or device manufacturer) rather than just by any trusted third party. For otherwise, the MIDP specification (see Section 2) demands that the uninstrumented methods which are called by the instrumented ones do still pop-up authorisation screens, despite the fact that the user (or the policy) has already approved all of the resources held by resource managers.

As an alternative deployment scenario, the resource management library could be integrated into future versions of MIDP. In this case, the MIDP class loader would have to check for correct use of the API, rendering unnecessary the requirement that MIDlets be signed by the network operator.

6 Related Work

Runtime monitoring to increase software reliability is at the heart of the Java language [4] with its mandatory runtime checking of array bounds and null pointer dereferences. Several frameworks have been proposed for enhancing Java with runtime monitoring of resource consumption, for example JRes [3], J-Seal [1] and J-RAF [10]. Real-time Java (RTSJ [5]) provides resource monitoring as part of its support for real-time applications. These frameworks monitor specific resources (CPU, memory, network bandwidth, threads), relying on instrumentation of either the JVM (for CPU time), low level system classes (for memory and network bandwidth) and the bytecode itself (for memory and instruction counting). Where our resource management API is designed to enforce security, these frameworks

[7] The hash table based implementation of multisets may fail to function correctly if resources are added that breach the contract that Java imposes on the `equals` and `hashCode` methods.

were developed to support resource aware applications, which can adapt their behaviour in response to resource fluctuation, for example by trading precision for time (by returning an imprecise result to meet a deadline), or time for memory (by caching less to reduce memory consumption).

Runtime monitoring can be used to check whether a program meets a safety property specified in a propositional temporal logic. Tools like JPaX [9] compile a specification into a finite automaton which runs in parallel with the program, observing its behaviour. This kind of temporal specification can express resource protocols like authorise-before-use but is not expressive enough to capture protocols that involve counting potentially unbounded resources.

Schneider [13] advocates a similar use of (not necessarily finite) automata for enforcing security policies at runtime. [15] extends this by allowing an application to query the policy for compliance with a planned sequence of actions. Thus, the application can react gracefully to the policy's decisions; our resource managers provide a similar policy query feature through the `enable` method.

7 Conclusion

We have designed a Java library for tracking and monitoring the use of external resources on MIDP mobile phones (e. g. sending text messages). The library improves the flexibility of runtime monitoring in MIDP (which previously was in the hands of the user), providing a clear user interface and flexible policies while maintaining the security guarantee that any attempt to abuse resources will be trapped.

Our technical contribution is an API for fine-grained accounting of external resources, where fine-grained accounting is achieved by resource managers tracking not just a fixed set of resources but an input-dependent unbounded set (e. g. phone numbers from the user's address book). The API is extensible, admitting to add new resource types and new policies by extending the class hierarchy. Moreover, we have outlined how a trusted library implementing the API can be deployed to MIDP phones as part of a potentially malicious application in such a way that the application cannot subvert the security guarantee (turning the application into a less malicious one). Finally, resource monitoring can be switched off by "erasing" resource managers, which reduces the overhead without changing the observable behaviour of resource safe applications (and we are working on a type system for certifying resource safety [2, chapter 3.3]).

Acknowledgements

This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein. Ian Stark was also supported by an Advanced Research Fellowship from the UK Engineering and Physical Sciences Research Council, EPSRC project GR/R76950/01.

References

[1] Walter Binder, Jarle Hulaas, and Alex Villazón. Portable resource control in Java: The J-SEAL2 approach. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 139–155, 2001.

[2] Mobius Consortium. Deliverable 2.1: Intermediate report on type systems. Available online from http://mobius.inria.fr, September 2006.

[3] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 21–35, 1998.

[4] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, third edition*. The Java Series. Addison-Wesley Publishing Company, 2005.

[5] JSR 1 Expert Group. JSR 1: Real-time specification for Java. Java specification request, Java Community Process, January 2002.

[6] JSR 118 Expert Group. JSR 118: Mobile information device profile 2.0. Java specification request, Java Community Process, November 2002.

[7] JSR 139 Expert Group. JSR 139: Connected limited device configuration 1.1. Java specification request, Java Community Process, March 2003.

[8] JSR 205 Expert Group. JSR 205: Wireless messaging API 2.0. Java specification request, Java Community Process, June 2004.

[9] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. *Electr.Notes Theor. Comput. Sci.*, 55(2):200–217, 2001.

[10] Jarle Hulaas and Walter Binder. Program transformations for portable CPU accounting and control in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 169–177, 2004.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, July 2007. In Progress. Available from http://www.jmlspecs.org.

[12] Kevin D. Mitnick and William L. Simon. *The Art of Deception*. John Wiley and Sons, Inc., 2002.

[13] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[14] Unknown. Redbrowser.A, February 2006. J2ME trojan. Identified as Redbrowser.A (F-Secure), J2ME/Redbrowser.a (McAfee), Trojan.Redbrowser.A (Symantec), Trojan-SMS.J2ME.Redbrowser.a (Kaspersky Lab).

[15] Dries Vanoverberghe and Frank Piessens. Supporting security monitor-aware development. In *International Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2007.

Safety Guarantees from Explicit Resource Management

David Aspinall, Patrick Maier, and Ian Stark

Laboratory for Foundations of Computer Science
School of Informatics, The University of Edinburgh, Scotland
{David.Aspinall,Patrick.Maier,Ian.Stark}@ed.ac.uk

Abstract. We present a language and a program analysis that certifies the safe use of flexible resource management idioms, in particular advance reservation or "block booking" of costly resources. This builds on previous work with *resource managers* that carry out runtime safety checks, by showing how to assist these with compile-time checks. We give a small ANF-style language with explicit resource managers, and introduce a type and effect system that captures their runtime behaviour. In this setting, we identify a notion of *dynamic safety* for running code, and show that dynamically safe code may be executed without runtime checks. We show a similar *static safety* property for type-safe code, and prove that static safety implies dynamic safety. The consequence is that typechecked code can be executed without runtime instrumentation, and is guaranteed to make only appropriate use of resources.

1 Introduction

Safe management of resources is a crucial aspect of software correctness. Bad resource management impacts reliability and security. The more expensive a resource or the more complex its usage pattern, the more important is good management. For example, a media player could crash badly, leaving the hardware in a messy state, if its memory management was governed by the overly optimistic assumption that every request for memory will succeed. Malware on a mobile phone can defraud an unaware user by maliciously sending text messages to premium rate numbers, if there is no effective management of network access [12]. On current mobile platforms such as Java MIDP 2.0, management of network access is commonly left to the user, but users can easily be deceived by social engineering attacks.

Unfortunately, current programming languages do not provide special mechanisms for resource management. Therefore, programmers can only hope that their applications are resource safe, or use necessarily imprecise analyses to try to show this. For example, there are type systems that over-approximate (hopefully tightly) the memory requirements of an application [6], and static analyses that over-approximate the number of text messages being sent by an application [7].

These approaches may fail if a dynamic set of resources must be managed, as with *bulk messaging* where the user wants to send a text message to a number of recipients selected from an address book. Because of the cost of sending text messages, the user must authorise each recipient (i. e., their phone number) explicitly. This could happen individually, just before each message is being sent, or collectively, before sending the

first message. Collective authorisation, or *block booking* of resources, is preferable but requires detailed resource management, keeping track of the (multi-)set of authorised resources – in this case the permitted phone numbers.

In this paper, we present a language-based mechanism that provides programmers with a safe way to control complex resource usage patterns using a notion of *resource manager*. Figure 1 shows the code of a bulk messaging application using resource managers in our intermediate-level functional programming language. The language and functions used will be explained in full detail in Section 2; for now, we just give an outline of operation. The function send_bulk calls send_msgs to send the message msg to the phone numbers stored in the array nums. Along with these two arguments send_msgs takes a resource manager m' which encapsulates the resources that have been authorised (during the call to enable) to send the messages. For each phone number in nums, send_msgs calls the wrapper function send_msg, passing along a resource manager. Prior to calling the primitive send function prim_send_msg, the wrapper checks (using assertAtLeast) whether its input manager m contains the resource required to send a message to num; if the resource is not present, the program will abort with a runtime error, otherwise send_msg removes the resource from the manager (using split), and returns the modified manager as m'.

The bulk messaging application is (dynamically) resource safe by construction, as the resource managers will trap attempts to abuse resources. The resource manager abstraction works in tandem with a static analysis, so that programs which can be proved resource safe statically can be treated more efficiently at runtime by removing the dynamic accounting code. In Section 3.2, we prove resource safety statically for the bulk messaging application.

Our contribution is two-fold. In Section 2, we develop a functional programming language for coding complex resource idioms, such as block booking resources in the bulk messaging application. The language is essentially a first-order functional language in administrative normal form (ANF) [10] with a novel type system serving two purposes. First, the type system names input and output parameters of functions and avoids shadowing of previously bound names, thus admitting to view functions as relations (expressed by logical formulae) between their input and output parameters. Second, the language includes a special, linear type for resource managers, where linearity serves as a means of introducing stateful objects into an otherwise pure functional language. Resource managers track what resources a program is allowed to use, and the operational semantics causes the program to go wrong (i. e., abort with a runtime error) as soon as it attempts to abuse resources. This induces a notion of *dynamic resource safety*, which holds if a program never attempts to abuse resources. In this case, accounting is not necessary. As our first result, we show that erasing resource managers does not alter the semantics of dynamically resource safe programs.

Decisions about which resources programs may use are typically guided by *resource policies*. From the point of view of a program, a policy is simply an oracle determining what resources to grant; and we abstract this as a non-deterministic operation on resource managers. This covers many concrete policy mechanisms, both static (e. g., Java-style policy files) or dynamic (e. g., user interaction); see [3] for more on the interaction of resource managers and policies.

```
send_bulk ::
λ let (r) = res_from_nums (nums) in
  let (m) = init () in
  let (m',r') = enable (m,r) in
  let (n) = size (r') in
  if n then let () = consume (m') in
            ret ()
       else let (m'') = send_msgs (msg,nums,m') in
            let (m''') = assertEmpty (m'') in
            let () = consume (m''') in
            ret () :
(msg:str, nums:str[]) → ()

res_from_nums ::
λ let (i) = length (nums) in
  let (r) = empty () in
  let (r') = res_from_nums' (nums,r,i) in
  ret (r') :
(nums:str[]) → (r':res{})

res_from_nums' ::
λ if i then let (i') = sub (i,1) in
            let (num) = read (nums,i') in
            let (c) = fromstr (num) in
            let (r_c) = single (c,1) in
            let (r'') = sum (r, r_c) in
            let (r') = res_from_nums' (nums,r'',i') in
            ret (r')
       else let (r') = id (r) in
            ret (r') :
(nums:str[], r:res{}, i:int) → (r':res{})

send_msgs ::
λ let (i) = length (nums) in
  let (m') = send_msgs' (msg,nums,m,i) in
  ret (m') :
(msg:str, nums:str[], m:mgr) → (m':mgr)

send_msgs' ::
λ if i then let (i') = sub (i,1) in
            let (num) = read (nums,i') in
            let (m'') = send_msg (msg,num,m) in
            let (m') = send_msgs' (msg,nums,m'',i') in
            ret (m')
       else let (m') = id (m) in
            ret (m') :
(msg:str, nums:str[], m:mgr, i:int) → (m':mgr)

send_msg ::
λ let (c) = fromstr (num) in
  let (r) = single (c,1) in
  let (m',m_r) = split (m,r) in
  let (m_r') = assertAtLeast (m_r,r) in
  let () = prim_send_msg (msg,num) in
  let () = consume (m_r') in
  ret (m') :
(msg:str, num:str, m:mgr) → (m':mgr)

prim_send_msg ::
λ . . . :
(msg:str, num:str) → ()
```

Fig. 1. Bulk messaging application.

In Section 3 we present our second contribution, an effect type system for deriving relational approximations of functions. These approximations are expressed as pairs of constraints in a first-order logic, specifying a pre- and postcondition (or rather, state transforming action) of a given function, similar to Hoare type theory [11]; note that the use of logical formulae as effects is the rationale behind choosing a programming language where functions have named input and output parameters. Typability of functions in the effect type system induces a notion of *static resource safety*. As our second result, we prove a soundness theorem stating that static implies dynamic resource safety. As a corollary, we show that resource managers can always be erased from statically resource safe programs. Proofs have been omitted due to lack of space.

2 A Programming Language for Resource Management

We introduce a simple programming language with built-in constructs for handling resource managers. The language is essentially a simply-typed first-order functional language in ANF [10], with the additional features that functions take and return tuples of values, function types name input and output arguments, scoping avoids shadowing, and the type of resource managers enforces a linearity restriction on its values. The first three of these features are related to giving the language a relational appeal: for the purpose of specifying and reasoning logically, functions ought to be viewed as relations

$$
\begin{array}{rcll}
\langle \text{fundecl} \rangle & ::= & \langle \text{prodtype} \rangle \rightarrow \langle \text{prodtype} \rangle & \textit{(built-in function)} \\
& | & \boldsymbol{\lambda} \langle \text{exp} \rangle : \langle \text{prodtype} \rangle \rightarrow \langle \text{prodtype} \rangle & \textit{(λ-abstraction)} \\
\langle \text{exp} \rangle & ::= & \textbf{if } \langle \text{val} \rangle \textbf{ then } \langle \text{exp} \rangle \textbf{ else } \langle \text{exp} \rangle & \textit{(conditional)} \\
& | & \textbf{let } (\langle \text{var} \rangle, \ldots, \langle \text{var} \rangle) = \langle \text{fun} \rangle \, (\langle \text{val} \rangle, \ldots, \langle \text{val} \rangle) \textbf{ in } \langle \text{exp} \rangle & \textit{(function call)} \\
& | & \textbf{ret } (\langle \text{var} \rangle, \ldots, \langle \text{var} \rangle) & \textit{(return)} \\
\langle \text{val} \rangle & ::= & \langle \text{const} \rangle \mid \langle \text{var} \rangle & \\
\langle \text{prodtype} \rangle & ::= & (\langle \text{var} \rangle : \langle \text{type} \rangle, \ldots, \langle \text{var} \rangle : \langle \text{type} \rangle) & \\
\langle \text{type} \rangle & ::= & \langle \text{datatype} \rangle \mid \textbf{mgr} & \\
\langle \text{datatype} \rangle & ::= & \textbf{unit} \mid \textbf{int} \mid \textbf{str} \mid \textbf{res} \mid \textbf{res\{\}} \mid \langle \text{datatype} \rangle [] &
\end{array}
$$

Fig. 2. BNF grammar.

between input and output parameters. The fourth feature is a means of introducing state into a functional language.

The choice for such a language has been inspired by Grail [2], another first-order functional language in ANF. Moreover, Appel [1] argues that ANF, the intermediate language used by many compilers for functional languages, and SSA, the intermediate representation used by most compilers for imperative languages, are essentially the same thing. Therefore, our language should capture the essence of first-order programming languages, whether functional or imperative.

2.1 Syntax and Static Semantics

Grammar. Figure 2 shows the grammar of the programming language. The nonterminals ⟨fun⟩, ⟨var⟩ and ⟨const⟩ represent *functions*, *variables* and *constants*, respectively. A *program* Π is a partial function from ⟨fun⟩ to ⟨fundecl⟩, i. e., Π maps functions to function declarations, which are either type declarations for built-in functions or λ-abstractions (with type annotations serving as variable binders). We use the notation $\Pi(f) = [\boldsymbol{\lambda} \ldots] \sigma \rightarrow \sigma'$ if we are only interested in the type of f, regardless whether f is built-in or a λ-abstraction. By $dom(\Pi)$, we denote the domain of Π. We denote the restriction of Π to the built-in functions by Π_0, i. e., $\Pi(f)$ is a λ-abstraction if and only if $f \in dom(\Pi) \setminus dom(\Pi_0)$. We assume that Π_0 declares exactly the functions that are shown in Figure 4.

The grammar of *expressions* $e \in$ ⟨exp⟩ and *values* $v \in$ ⟨val⟩ is quite standard for a first-order functional language in ANF. Throughout, functions operate on tuples of values, which is reflected by the syntax for function call and return. The sets of free and bound (by the let-construct) variables of an expression e, denoted by $free(e)$ and $bound(e)$ respectively, are defined in the usual way.

Datatypes $\tau \in$ ⟨datatype⟩ comprise the unit type, integers, strings, resources, multisets of resources, and arrays. A *type* $\tau \in$ ⟨type⟩ is either a datatype or the special type of resource managers, denoted $\mathbf{mgr}$. See Section 2.2 for the interpretations of types. A tuple $(x_1{:}\tau_1, \ldots, x_n{:}\tau_n) \in$ ⟨prodtype⟩ is a *product type* if the variables $x_1, \ldots, x_n$ are pairwise distinct. Product types appear to associate types to variables,

but they really associate variables *and* types to positions in tuples. A pair of product types of the form $(x_1{:}\tau_1,\ldots,x_m{:}\tau_m) \longrightarrow (x'_1{:}\tau'_1,\ldots,x'_n{:}\tau'_n)$ forms a *function type* if the variable sets $\{x_1,\ldots,x_m\}$ and $\{x'_1,\ldots,x'_n\}$ are disjoint. We call the product types to the left and right of the arrow *argument type* and *return type*, respectively. As an example consider the type of the function send_msg from Figure 1. It states that send_msg takes two strings and a resource manager and returns a resource manager, while at the same time binding the names of the formal input parameters msg, num and m and announcing that the formal output parameter will be m'.

Static typing. A *type environment* Γ is a functional association list of type declarations of the form $x{:}\tau$, where x is a variable and τ a type. Being functional implies that whenever Γ contains two type declarations $x{:}\tau$ and $x{:}\tau'$ we must have $\tau = \tau'$. Therefore, Γ can be seen as a partial function mapping variables to types. By $dom(\Gamma)$, we denote the domain of this partial function, and for $x \in dom(\Gamma)$, we may write $\Gamma(x)$ for the unique type which Γ associates to x. We write type environments as comma-separated lists, the empty list being denoted by $\emptyset$. The restriction $\Gamma|_X$ of Γ to a set of variables X, is defined in the usual way and induces a partial order $\succeq$ type environments, where $\Gamma' \succeq \Gamma$ iff $\Gamma'|_{dom(\Gamma)} = \Gamma$.

We call a type environment $\Gamma = x_1{:}\tau_1,\ldots,x_n{:}\tau_n$ *linear* if the variables $x_1,\ldots,x_n$ are pairwise distinct. Note that such a linear type environment Γ may be viewed as a product type $\sigma = (x_1{:}\tau_1,\ldots,x_n{:}\tau_n)$, and vice versa. Occasionally, we will write $\Pi(f) = [\boldsymbol{\lambda}\ldots]\Gamma \longrightarrow \Delta$ to emphasise that argument and return types of the function f are to be viewed as linear type environments.

Figure 3 shows the typing rules for the programming language. The judgement $C;\Gamma \vdash v : \tau$ expresses that the value v has type τ in type environment Γ and context C, where a *context* is a set of variables (generally the set of variables occurring in some super-expression of v). Note that (T-const) restricts program constants to the unit value, integers and strings, which are the interpretations of the types $\mathbf{unit}$, $\mathbf{int}$ and $\mathbf{str}$, respectively (see Section 2.2). All other types are abstract in the sense that their values can only be accessed through built-in functions.

The judgement $C;\Gamma \vdash_\Pi e : \sigma$ means that the expression e has product type σ in type environment Γ, context C and program Π. If the program is understood we may write $C;\Gamma \vdash e : \sigma$. There are three things worth noting about expression typing. First, although the type system is linear, weakening and contraction are available to all types but $\mathbf{mgr}$, rendering $\mathbf{mgr}$ the sole linear type of the language. Second, the side condition of (T-let) ensures that let-bound variables do not shadow any variables in the context (which is generally a superset of the set of variables occurring in the let-expression). Third, the rule (T-ret) matches the variables in the return expression to the variables in the product type, thus enforcing that an expression uniformly uses the same variables to return its results (even though these return variables may be let-bound in different branches of the expression). Note that (T-ret) is the only rule to exploit type information about variables. Finally, the judgement $\Gamma \vdash e : \sigma$ (or $\Gamma \vdash_\Pi e : \sigma$ if we want to stress the program Π) means that e has product type σ in a linear type environment Γ.

The judgement $\Pi \vdash f$ states that f is a well-typed λ-abstraction in program Π. Note that the syntax of λ-abstractions does not appear to bind variables, yet it does bind the variables hidden in the argument type. Note also that the restriction on function

Typing of values $C; \Gamma \vdash v : \tau$

(T-var) $\dfrac{}{C; x{:}\tau \vdash x : \tau}$ if $x \in C$ (T-const) $\dfrac{}{C; \emptyset \vdash d : \tau}$ if $\begin{cases} d \in \tau \wedge \\ \tau \in \{\mathbf{unit}, \mathbf{int}, \mathbf{str}\} \end{cases}$

Typing of expressions $C; \Gamma \vdash e : \sigma$

(T-weak) $\dfrac{C; \Gamma \vdash e : \sigma}{C; \Gamma, x{:}\tau \vdash e : \sigma}$ if $\begin{cases} x \in C \wedge \\ \tau \neq \mathbf{mgr} \end{cases}$ (T-contr) $\dfrac{C; \Gamma, x{:}\tau, x{:}\tau \vdash e : \sigma}{C; \Gamma, x{:}\tau \vdash e : \sigma}$ if $\tau \neq \mathbf{mgr}$

(T-if) $\dfrac{C; \Gamma \vdash v : \mathbf{int} \qquad C; \Gamma' \vdash e_1 : \sigma \qquad C; \Gamma' \vdash e_2 : \sigma}{C; \Gamma, \Gamma' \vdash \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \sigma}$ (T-xch) $\dfrac{C; \Gamma, \Gamma' \vdash e : \sigma}{C; \Gamma', \Gamma \vdash e : \sigma}$

(T-ret) $\dfrac{C; \Gamma_1 \vdash x_1 : \tau_1 \quad \ldots \quad C; \Gamma_n \vdash x_n : \tau_n}{C; \Gamma_1, \ldots, \Gamma_n \vdash \mathbf{ret}\ (x_1, \ldots, x_n) : (x_1{:}\tau_1, \ldots, x_n{:}\tau_n)}$

(T-let) $\dfrac{\begin{array}{c} \Pi(f) = [\boldsymbol{\lambda} \ldots](z_1{:}\tau_1, \ldots, z_m{:}\tau_m) \to (z'_1{:}\tau'_1, \ldots, z'_n{:}\tau'_n) \\ C; \Gamma_1 \vdash v_1 : \tau_1 \quad \ldots \quad C; \Gamma_n \vdash v_m : \tau_m \\ C \cup \{x'_1, \ldots, x'_n\}; \Gamma', x'_1{:}\tau'_1, \ldots, x'_n{:}\tau'_n \vdash e' : \sigma'' \end{array}}{C; \Gamma_1, \ldots, \Gamma_m, \Gamma' \vdash \mathbf{let}\ (x'_1, \ldots, x'_n) = f\ (v_1, \ldots, v_m)\ \mathbf{in}\ e' : \sigma''}$ if $(*)$

where $(*)$ $\begin{cases} x'_1, \ldots, x'_n \text{ pairwise distinct} \wedge \\ x'_1, \ldots, x'_n \notin C \cup dom(\Gamma') \end{cases}$

Typing of expressions $\Gamma \vdash e : \sigma$

(T-lin) $\dfrac{dom(\Gamma); \Gamma \vdash e : \sigma}{\Gamma \vdash e : \sigma}$ if Γ linear

Well-typedness of λ-abstractions $\Pi \vdash f$

(T-lam) $\dfrac{\Pi(f) = \boldsymbol{\lambda} e : (x_1{:}\tau_1, \ldots, x_m{:}\tau_m) \to \sigma' \qquad x_1{:}\tau_1, \ldots, x_m{:}\tau_m \vdash e : \sigma'}{\Pi \vdash f}$

Fig. 3. Typing rules (for a fixed program Π).

types means that the return variables of the body of a λ-abstraction must be disjoint from its argument variables. Finally, we call a program Π *well-typed* if $\Pi \vdash f$ for all $f \in dom(\Pi) \setminus dom(\Pi_0)$.

Lemma 1. *Let e be an expression (referring to an implicit program Π), Γ a type environment and σ a product type.*

1. *If $\Gamma \vdash e : \sigma$ then $free(e) \subseteq dom(\Gamma)$ and $bound(e) \cap dom(\Gamma) = \emptyset$.*
2. *If $\Gamma \vdash e : \sigma$ and $X \supseteq free(e)$ then $\Gamma|_X \vdash e : \sigma$.*

2.2 Interpretation of Types and Effects of Built-in Functions

Constraints. To provide a formal semantics for the built-in functions, we introduce a many-sorted first-order language $\mathcal{L}$ with equality. Sorts of $\mathcal{L}$ are the datatypes of the programming language (note that this excludes the type $\mathbf{mgr}$). Formulae of $\mathcal{L}$ are formed from atomic formulae using the usual Boolean connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ (in decreasing order of precedence), and the quantifiers $\forall x{:}\tau$ and $\exists x{:}\tau$, where $x \in \langle \text{var} \rangle$

is a variable and $\tau \in \langle\text{datatype}\rangle$ a sort. Atomic formulae are the Boolean constants $\top$ and $\bot$, or are constructed from terms using the binary equality predicate $\approx$ (which is available for all sorts), the binary inequality predicate $\leq$ on sort $\mathbf{int}$ or the binary inclusion predicate $\subseteq$ on sort $\mathbf{res\{\}}$. Terms are constructed from variables in $\langle\text{var}\rangle$ and the term constructors, which are introduced below, alongside associating the sorts to specific interpretations.

Sort $\mathbf{unit}$ is interpreted by the one-element set $\{\star\}$. Its only constant is $\star$. There are no function symbols.

Sort $\mathbf{int}$ is interpreted by the integers with infinity. Constants are the integers plus ∞. Function symbols are the usual $- : \mathbf{int} \to \mathbf{int}$ and $+, \cdot, /, \% : \mathbf{int} \times \mathbf{int} \to \mathbf{int}$ (where $/$ and $\%$ denote integer division and remainder, respectively).

Sort $\mathbf{str}$ is interpreted by the set of strings (over some fixed but unspecified alphabet). Constants are all strings. The only function symbol is $++ : \mathbf{str} \times \mathbf{str} \to \mathbf{str}$ (concatenation).

Sort $\mathbf{res}$ is interpreted by an arbitrary infinite set (whose elements are termed *resources*). There are no constants, and $\mathit{fromstr} : \mathbf{str} \to \mathbf{res}$, an embedding of strings into resources, is the only one function symbol.

Sort $\mathbf{res\{\}}$ is interpreted by multisets of resources. It features the constant $\emptyset$ (empty multiset) and the function symbols $\cap, \cup, \uplus : \mathbf{res\{\}} \times \mathbf{res\{\}} \to \mathbf{res\{\}}$ (intersection, union and sum of multisets, respectively), $|\_| : \mathbf{res\{\}} \to \mathbf{int}$ (size of a multiset), $\mathit{count} : \mathbf{res\{\}} \times \mathbf{res} \to \mathbf{int}$ (counting the multiplicity of a resource in a multiset) and $\{\_:\_\} : \mathbf{res} \times \mathbf{int} \to \mathbf{res\{\}}$ (constructing a "singleton" multiset containing a given resource with a given multiplicity and nothing else).

Sort $\tau[]$ is interpreted by integer-indexed arrays of elements of sort τ, where an integer-indexed array is a function from an initial segment of the natural numbers to τ. This sort features the constant null (array of length 0) and the function symbols $\mathit{len} : \tau[] \to \mathbf{int}$ (length of an array), $\_[\_] : \tau[] \times \mathbf{int} \to \tau$ (reading at a given index) and $\_[\_:=\_] : \tau[] \times \mathbf{int} \times \tau \to \tau[]$ (updating a given index with a given value). Note that the values of $a[i]$ and $a[i:=v]$ are generally unspecified if the index i is out of bounds (i. e., $i < 0$ or $i \geq \mathit{len}(a)$). As an exception, for $i = \mathit{len}(a)$, the array $a[i:=v]$ properly extends a, i. e., $\mathit{len}(a[i:=v]) = \mathit{len}(a) + 1$. This models vectors that can grow in size.

Treating the type $\mathbf{mgr}$ as an alias for the sort $\mathbf{res\{\}}$, type environments can be seen as associating sorts to variables. Given a type environment Γ and constraint $\phi \in \mathcal{L}$, we write $\Gamma \vdash \phi$ if ϕ is well-sorted w. r. t. Γ; note that this entails $\mathit{free}(\phi) \subseteq \mathit{dom}(\Gamma)$, where $\mathit{free}(\phi)$ is the set of free variables in ϕ.

Substitutions. A *substitution* μ maps variables $x \in \langle\text{var}\rangle$ to values $\mu(x) \in \langle\text{val}\rangle$ (which are variables again or constants, not arbitrary terms). We denote the domain of a substitution μ by $\mathit{dom}(\mu)$. Given a type environment Γ, we write $\Gamma\mu$ for the type environment that arises from substituting the variables in Γ according to μ. This is defined recursively: $\emptyset\mu = \emptyset$ and $(\Gamma, x{:}\tau)\mu$ equals $\Gamma\mu, x{:}\tau$ if $x \notin \mathit{dom}(\mu)$, or $\Gamma\mu, \mu(x){:}\tau$ if $\mu(x) \in \langle\text{var}\rangle$, or $\Gamma\mu$ if $\mu(x) \in \langle\text{const}\rangle$. Note that $\Gamma\mu$ need not be linear even if Γ is. Given a formula ϕ such that $\Gamma \vdash \phi$, we write $\phi\mu$ for the formula obtained by substituting the free variables of ϕ according to μ, avoiding capture. Note that $\Gamma \vdash \phi$ implies $\Gamma\mu \vdash \phi\mu$.

Valuations. Let Γ be a type environment. A *Γ-valuation* α maps variables $x \in dom(\Gamma)$ to elements $\alpha(x)$ in the interpretation of the sort $\Gamma(x)$; we call α a *valuation* if we do not care about the particular type environment Γ. We denote the domain of α by $dom(\alpha)$. Note that $dom(\alpha) \subseteq dom(\Gamma)$ but not necessarily $dom(\alpha) = dom(\Gamma)$; we call α a *maximal* Γ-valuation if $dom(\alpha) = dom(\Gamma)$. Given a Γ-valuation α and a set of variables X, we denote the restriction of α to X by $\alpha|_X$; note that $dom(\alpha|_X) = dom(\alpha) \cap X$. Restriction induces a partial order $\succeq$ on Γ-valuations, where $\alpha' \succeq \alpha$ iff $\alpha'|_{dom(\alpha)} = \alpha$. Given n pairwise distinct variables $x_i \in dom(\Gamma)$ and corresponding elements d_i in the interpretation of $\Gamma(x_i)$, we write $\alpha\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ for the Γ-valuation α' that maps the x_i to d_i and all other $x \in dom(\alpha)$ to $\alpha(x)$. In the special case $dom(\alpha) = \emptyset$, we may drop α and simply write $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$.

Entailment. Let $\phi, \psi \in \mathcal{L}$ be constraints such that $\Gamma \vdash \phi$ and $\Gamma \vdash \psi$. Given a Γ-valuation α with $free(\phi) \subseteq dom(\alpha)$, we write $\alpha \models \phi$ if α satisfies ϕ. We write $\models \phi$ if $\alpha \models \phi$ for all Γ-valuations α with $free(\phi) \subseteq dom(\alpha)$, and we write $\phi \models \psi$ if $\alpha \models \phi$ implies $\alpha \models \psi$ for all Γ-valuations α with $free(\phi) \cup free(\psi) \subseteq dom(\alpha)$. Entailment induces a theory $\mathcal{T} = \{\phi \mid free(\phi) = \emptyset \wedge \top \models \phi\}$, with respect to which entailment can be reduced to unsatisfiability. Note that unsatisfiability w. r. t. $\mathcal{T}$ is not even semi-decidable as $\mathcal{T}$ contains Peano arithmetic. Thus for reasoning purposes, we will generally approximate $\mathcal{T}$ by weaker theories.

Effects. Let f be a built-in function with $\Pi(f) = \Gamma \longrightarrow \Delta$ (viewing argument and return types of f as type environments Γ and Δ, respectively.) An *effect* for f is a pair of constraints ϕ and ψ such that $\Gamma \vdash \phi$ and $\Gamma, \Delta \vdash \psi$. (Note that $\Gamma \longrightarrow \Delta$ being a function type implies $dom(\Gamma) \cap dom(\Delta) = \emptyset$, hence Γ, Δ is a type environment.) We write $\phi \rightarrow \psi$ to denote such an effect, and we call ϕ its *precondition* and ψ its *action*.

An *effect environment* maps the built-in functions $f \in dom(\Pi_0)$ to effects for f. Figure 4 displays the effect environment Θ_0, providing an axiomatic, relational semantics for all $f \in dom(\Pi_0)$. This semantics ties most built-in functions to corresponding logical operators in a straightforward way; note the non-trivial preconditions for division, reading and writing arrays, and constructing singleton multisets. The effects of functions operating on resource managers warrant some explanation.

init returns an empty manager m'.

enable non-deterministically adds some sub-multiset of r to manager m, returning the result in manager m'; the complement of the added multiset is returned in r'. In an implementation [3] the multiset to be added to m would be chosen by some *policy*, perhaps involving security profiles or user input; we use non-determinism to abstractly model such policy mechanisms.

split splits the multiset held by manager m and distributes it to the managers m'_1 and m'_2 such that m'_2 gets the largest possible sub-multiset of r.

join adds the multisets held by managers m_1 and m_2, returning their sum in m'.

consume is an explicit destructor for manager m and all its resources; the linear type system means that calls to **consume** are necessary even if m is known to be empty.

assertEmpty acts as identity on managers, but subject to the precondition that m is empty; it will be treated specially by the programming language semantics.

f	$\Pi_0(f)$	$\Theta_0(f)$
$\mathbf{id}_\tau$	$(x{:}\tau) \longrightarrow (x'{:}\tau)$	$\top \rightarrow x' \approx x$
$\mathbf{eq}_\tau$	$(x_1{:}\tau, x_2{:}\tau) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx 1 \wedge x_1 \approx x_2 \vee i' \approx 0 \wedge x_1 \not\approx x_2$
add	$(i_1{:}\mathbf{int}, i_2{:}\mathbf{int}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx i_1 + i_2$
sub		$\top \rightarrow i' \approx i_1 + (-i_2)$
mul		$\top \rightarrow i' \approx i_1 \cdot i_2$
div		$i_2 \not\approx 0 \rightarrow i' \approx i_1 \,/\, i_2$
mod		$i_2 \not\approx 0 \rightarrow i' \approx i_1 \,\%\, i_2$
leq		$\top \rightarrow i' \approx 1 \wedge i_1 \leq i_2 \vee i' \approx 0 \wedge i_1 \not\leq i_2$
conc	$(w_1{:}\mathbf{str}, w_2{:}\mathbf{str}) \longrightarrow (w'{:}\mathbf{str})$	$\top \rightarrow w' \approx w_1 +\!\!+ w_2$
fromstr	$(w{:}\mathbf{str}) \longrightarrow (c'{:}\mathbf{res})$	$\top \rightarrow c' \approx \mathit{fromstr}(w)$
$\mathbf{null}_\tau$	$() \longrightarrow (a'{:}\tau[])$	$\top \rightarrow a' \approx \mathit{null}$
$\mathbf{length}_\tau$	$(a{:}\tau[]) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \mathit{len}(a)$
$\mathbf{read}_\tau$	$(a{:}\tau[], i{:}\mathbf{int}) \longrightarrow (x'{:}\tau)$	$0 \leq i \wedge i < \mathit{len}(a) \rightarrow x' \approx a[i]$
$\mathbf{write}_\tau$	$(a{:}\tau[], i{:}\mathbf{int}, x{:}\tau) \longrightarrow (a'{:}\tau[])$	$0 \leq i \wedge i \leq \mathit{len}(a) \rightarrow a' \approx a[i{:=}x]$
empty	$() \longrightarrow (r'{:}\mathbf{res}\{\})$	$\top \rightarrow r' \approx \emptyset$
single	$(c{:}\mathbf{res}, i{:}\mathbf{int}) \longrightarrow (r'{:}\mathbf{res}\{\})$	$i \geq 0 \rightarrow r' \approx \{c{:}i\}$
inter	$(r_1{:}\mathbf{res}\{\}, r_2{:}\mathbf{res}\{\}) \longrightarrow (r'{:}\mathbf{res}\{\})$	$\top \rightarrow r' \approx r_1 \cap r_2$
union		$\top \rightarrow r' \approx r_1 \cup r_2$
sum		$\top \rightarrow r' \approx r_1 \uplus r_2$
size	$(r{:}\mathbf{res}\{\}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \lvert r \rvert$
count	$(r{:}\mathbf{res}\{\}, c{:}\mathbf{res}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx \mathit{count}(r, c)$
include	$(r_1{:}\mathbf{res}\{\}, r_2{:}\mathbf{res}\{\}) \longrightarrow (i'{:}\mathbf{int})$	$\top \rightarrow i' \approx 1 \wedge r_1 \subseteq r_2 \vee i' \approx 0 \wedge r_1 \not\subseteq r_2$
init	$() \longrightarrow (m'{:}\mathbf{mgr})$	$\top \rightarrow m' \approx \emptyset$
enable	$(m{:}\mathbf{mgr}, r{:}\mathbf{res}\{\}) \longrightarrow (m'{:}\mathbf{mgr}, r'{:}\mathbf{res}\{\})$	$\top \rightarrow r' \subseteq r \wedge m \uplus r \approx m' \uplus r'$
split	$(m{:}\mathbf{mgr}, r{:}\mathbf{res}\{\}) \longrightarrow (m'_1{:}\mathbf{mgr}, m'_2{:}\mathbf{mgr})$	$\top \rightarrow m'_2 \approx m \cap r \wedge m \approx m'_1 \uplus m'_2$
join	$(m_1{:}\mathbf{mgr}, m_2{:}\mathbf{mgr}) \longrightarrow (m'{:}\mathbf{mgr})$	$\top \rightarrow m' \approx m_1 \uplus m_2$
consume	$(m{:}\mathbf{mgr}) \longrightarrow ()$	$\top \rightarrow \top$
assertEmpty	$(m{:}\mathbf{mgr}) \longrightarrow (m'{:}\mathbf{mgr})$	$m \approx \emptyset \rightarrow m' \approx m$
assertAtLeast	$(m{:}\mathbf{mgr}, r{:}\mathbf{res}\{\}) \longrightarrow (m'{:}\mathbf{mgr})$	$r \subseteq m \rightarrow m' \approx m$

Fig. 4. Types and effects of built-in functions. The subscripts τ indicate families of functions indexed by $\tau \in \langle\text{datatype}\rangle$, except for $\mathbf{id}_\tau$, which is indexed by $\tau \in \langle\text{type}\rangle$.

assertAtLeast acts as identity on managers, but subject to the precondition that the manager m contains the multiset r; will be treated specially by the programming language semantics.

To facilitate the presentation of programming language semantics, we capture the logical semantics of effects directly in terms of valuations. Given a built-in function f with $\Pi_0(f) = \Gamma \longrightarrow \Delta$ and $\Theta_0(f) = \phi \rightarrow \psi$, we define $\mathit{Eff}^{\Pi_0}_{\Theta_0}(f)$ to be the set of maximal (Γ, Δ)-valuations such that $\alpha \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f)$ if and only if $\alpha \models \phi \wedge \psi$.

2.3 Small-step Reduction Semantics

We present a stack-based reduction semantics (which is essentially a continuation semantics) for our programming language. We will show that reduction preserves the

resources stored in resource managers, thanks to linearity. Throughout this section, let Π be a fixed well-typed program.

Stacks. We call a tuple $\langle x_1, \ldots, x_n | \alpha, e \rangle$ a *frame* if $x_1, \ldots, x_n$ is a list of pairwise distinct variables, α is a valuation and e is an expression such that

- $dom(\alpha) \cap \{x_1, \ldots, x_n\} = \emptyset$ and
- $dom(\alpha) \subseteq free(e) \subseteq dom(\alpha) \cup \{x_1, \ldots, x_n\}$.

The roles of e (redex) and α (providing values for the free variables of e) should be clear. The x_i are only present if the frame is suspended waiting for a function to return in which case the x_i act as slots for the return values. A *pre-stack* is either $\lightning$ or ϵ or $F :: S$, where F is a frame and S is a pre-stack. (Pre-stacks essentially correspond to continuations in an abstract machine interpreting λ-terms in ANF [10].) A *stack* (or Π-stack if we want to emphasise the program Π) is a pre-stack of the form $\lightning$ or $\langle |\alpha, e\rangle :: S$. We call $\lightning$ the *error stack*. A stack of the form $\langle |\alpha, \mathbf{ret}\ (x_1,\ldots,x_n)\rangle :: \epsilon$ is called *terminal*. If $F :: S$ is a stack then F is its *top frame*.

Reduction. Figure 5 presents the rules generating the reduction relation $\rightsquigarrow_\Pi$ on stacks. We denote the reflexive-transitive closure of $\rightsquigarrow_\Pi$ by $\rightsquigarrow^*_\Pi$. As usual Π may be omitted if it is understood. Note that reduction performs an eager garbage collection in that it deallocates unused variables immediately by restricting the valuation α in the post stack to the free variables of the expression e.

Reduction is deterministic, except for calls to the built-in function $\mathbf{enable}$.

Proposition 2. *For all stacks S_0 there is at most one stack S_1 such that $S_0 \rightsquigarrow S_1$, unless S_0 is of the form $\langle |\alpha, \mathbf{let}\ (m',r') = \mathbf{enable}\ (m,r)\ \mathbf{in}\ e\rangle :: S'_0$.*

Typed stacks. Reduction is untyped since type information is not needed at runtime. However, various properties of reduction are best stated if the type of variables is known. Therefore, we annotate stacks with type environments and conservatively extend reduction to typed stacks.

Given a frame $\langle x_1, \ldots, x_n | \alpha, e\rangle$, we call $\langle x_1, \ldots, x_n | \alpha, e\rangle^\Gamma$ a *typed frame* if Γ is a linear type environment such that

- $dom(\Gamma) = dom(\alpha) \cup \{x_1, \ldots, x_n\}$,
- α is a Γ-valuation, and
- $\Gamma \vdash e : \sigma$ for some product type σ.

A *typed pre-stack* is $\lightning$, or ϵ, or $F :: \epsilon$ where F is a typed frame, or $F :: F' :: S'$ where S' is a typed pre-stack and $F = \langle x_1, \ldots, x_m | \alpha, e\rangle^\Gamma$ and $F' = \langle x'_1, \ldots, x'_n | \alpha', e'\rangle^{\Gamma'}$ are typed frames such that $\Gamma \vdash e : (z'_1 : \Gamma'(x'_1), \ldots, z'_n : \Gamma'(x'_n))$ for some variables $z'_1, \ldots, z'_n$. A *typed stack* is typed pre-stack of the form $\lightning$ or $\langle |\alpha, e\rangle^\Gamma :: S$. Given a typed frame $F = \langle x_1, \ldots, x_n | \alpha, e\rangle^\Gamma$, we denote its underlying frame $\langle x_1, \ldots, x_n | \alpha, e\rangle$ by $F^\natural$. We extend this notation to typed (pre-)stacks, writing $S^\natural$ for the (pre-)stack underlying the typed (pre-)stack S.

The following proposition shows that reduction does not break the invariants maintained by typed stacks.

$$\text{(R-ret)}\ \frac{\alpha'' = \alpha'\{x'_1 \mapsto \alpha(x_1), \ldots, x'_n \mapsto \alpha(x_n)\}}{\langle|\alpha, \mathbf{ret}\ (x_1,\ldots,x_n)\rangle :: \langle x'_1, \ldots, x'_n|\alpha', e'\rangle :: S \ \rightsquigarrow\ \langle|\alpha''|_{free(e')}, e'\rangle :: S}$$

$$\text{(R-let}_1^{\natural})\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \\ \alpha' = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\}\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in\ ret}\ (x'_1,\ldots,x'_n)\rangle :: S \rightsquigarrow \langle|\alpha'|_{free(e)}, e\rangle :: S}$$

$$\text{(R-let}_1)\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \qquad e' \neq \mathbf{ret}\ (x'_1,\ldots,x'_n) \\ \alpha' = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\}\end{array}}{\begin{array}{l}\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \\ \rightsquigarrow\ \langle|\alpha'|_{free(e)}, e\rangle :: \langle x'_1, \ldots, x'_n|\alpha|_{free(e')}, e'\rangle :: S\end{array}}$$

$$\text{(R-let}_2)\ \frac{\begin{array}{c}\Pi_0(f) = (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to (z'_1{:}\tau'_1,\ldots,z'_n{:}\tau'_n) \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad \alpha'_f \in \mathit{Eff}_{\Theta_0}^{\Pi_0}(f) \qquad \alpha'_f \succeq \alpha_f \\ \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1), \ldots, x'_n \mapsto \alpha'_f(z'_n)\}\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \ \rightsquigarrow\ \langle|\alpha'|_{free(e')}, e'\rangle :: S}$$

$$\text{(R-let}_2^{\lightning})\ \frac{\begin{array}{c}\Pi_0(f) = (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \qquad f \in \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\} \\ \alpha_f = \{z_1 \mapsto \alpha(v_1), \ldots, z_m \mapsto \alpha(v_m)\} \qquad \forall \alpha'_f \in \mathit{Eff}_{\Theta_0}^{\Pi_0}(f) : \alpha'_f \not\succeq \alpha_f\end{array}}{\langle|\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e'\rangle :: S \ \rightsquigarrow\ \lightning}$$

$$\text{(R-if}_1)\ \frac{\alpha(v) \neq 0}{\langle|\alpha, \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle :: S \ \rightsquigarrow\ \langle|\alpha|_{free(e_1)}, e_1\rangle :: S}$$

$$\text{(R-if}_2)\ \frac{\alpha(v) = 0}{\langle|\alpha, \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\rangle :: S \ \rightsquigarrow\ \langle|\alpha|_{free(e_2)}, e_2\rangle :: S}$$

Fig. 5. Small-step reduction relation $\rightsquigarrow$ (for a fixed program Π). Application of valuations α extends to values $v \in \langle \text{val} \rangle$ in the natural way, i. e., $\alpha(v) = v$ if v is a constant.

Proposition 3. *Let $\hat{S}_0$ be a typed stack and S_1 a stack. If $\hat{S}_0^{\natural} \rightsquigarrow S_1$ then there is a typed stack $\hat{S}_1$ such that $\hat{S}_1^{\natural} = S_1$.*

The proposition justifies the view of reduction on typed stacks as a conservative extension of the reduction relation defined in Figure 5, where reduction on typed stacks is defined by $\hat{S}_0 \rightsquigarrow_\Pi \hat{S}_1$ if and only if $\hat{S}_0^{\natural} \rightsquigarrow_\Pi \hat{S}_1^{\natural}$; as usual Π may be omitted if it is understood.

We call a stack S_0 *stuck* if there is no stack S_1 such that $S_0 \rightsquigarrow S_1$, and S_0 is neither terminal nor the error stack. Our next result shows that reduction on typed stacks will get stuck only at calls to built-in functions (other than $\mathbf{assertEmpty}$ and $\mathbf{assertAtLeast}$), and only if the preconditions of these calls fail. As the effects listed in Figure 4 reveal, reduction will get stuck only upon attempts to divide by 0, access arrays out of bounds or construct singleton multisets with negative multiplicity.

Proposition 4. *Let $\hat{S}$ be a typed stack. If $\hat{S}^{\natural}$ is stuck then it is of the form*

$$\langle |\alpha, \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e' \rangle :: S'\,,$$

$f \in dom(\Pi_0) \setminus \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\}$, and there is no $\alpha'_f \in Eff_{\Theta_0}^{\Pi_0}(f)$ such that $\alpha'_f \succeq \alpha_f$, where α_f is defined as in rule (R-let$_2$).

Preservation of resources. Given a typed frame $F = \langle x_1, \ldots, x_n | \alpha, e \rangle^{\Gamma}$, we define the multiset *res*(F) of *resources* in F by *res*$(F) = \biguplus\{\alpha(x) \mid x \in dom(\alpha), \Gamma(x) = \mathbf{mgr}\}$. We extend *res* to typed non-error stacks by defining *res*$(\epsilon) = \emptyset$ and *res*$(F :: S) =$ *res*$(F) \uplus$ *res*(S). Proposition 5 states *resource preservation*: The sum of all resources in the system remains unchanged by reduction, unless the built-in functions $\mathbf{enable}$ and $\mathbf{consume}$ are called. The former admits increasing (but not decreasing) the resources, whereas the latter behaves the other way round. Obviously, resource preservation depends on the linearity restriction on type $\mathbf{mgr}$, otherwise resources could be duplicated by re-using managers.

Proposition 5. *Let S_0 and S_1 be typed stacks such that $S_0 \rightsquigarrow S_1 \neq \lightning$.*

1. *If S_0 is of the form $\langle |\alpha, \mathbf{let}\ (m',r') = \mathbf{enable}\ (m,r)\ \mathbf{in}\ e \rangle^{\Gamma} :: S'_0$ then* res$(S_0) \subseteq$ *res*(S_1).
2. *If S_0 is of the form $\langle |\alpha, \mathbf{let}\ () = \mathbf{consume}\ (m)\ \mathbf{in}\ e \rangle^{\Gamma} :: S'_0$ then* res$(S_0) \supseteq$ *res*(S_1).
3. *In all other cases,* res$(S_0) =$ *res*(S_1).

2.4 Erasing Resource Managers

According to the reduction semantics, a call to $\mathbf{assertEmpty}$ or $\mathbf{assertAtLeast}$ either does nothing[1] or goes wrong, and calling one of these two tests is the only way to go wrong. Hence, if we know that a program cannot go wrong (and Section 3 will present a type system for proving just that) then we can erase all calls to these built-ins (or rather, replace them by true no-ops) and obtain an equivalent program.

In fact, we can do more than that. Once the assertion built-ins are gone, it is even possible to remove the resource managers themselves. By the design of the programming language (in particular, the choice of built-in operations on resource managers) the contents of resource managers cannot influence the values of variables of any other type. Informally, this justifies replacing the resource managers themselves by variables of type $\mathbf{unit}$ whenever we know that a program cannot go wrong. Erasing resource managers also means that the built-in functions acting on managers can be replaced by simpler ones on $\mathbf{unit}$: all of which are no-ops, except for $\mathbf{enable}$ itself.[2] The remainder of the section formalises this intuition.

Figure 6 shows the necessary program transformations to erase resource managers. Most fundamentally, erasure maps the manager type $\mathbf{mgr}$ to the unit type $\mathbf{unit}$.

[1] Due to the linearity restriction on resource managers these functions must copy the input manager to an output manager; a true no-op would violate resource preservation.

[2] We do keep the calls in place, so that erasure preserves the structure of programs; this simplifies reasoning, and does not preclude optimising away no-op calls at a later stage.

Erasure τ° of types τ

$\tau^\circ = \mathbf{unit}$ if $\tau = \mathbf{mgr}$
$\tau^\circ = \tau$ otherwise

Erasure Γ° of type environments Γ

$\emptyset^\circ = \emptyset$
$(\Gamma, x{:}\tau)^\circ = \Gamma^\circ, x{:}\tau^\circ$

Erasure σ° of product types σ

$(x_1{:}\tau_1,\ldots,x_n{:}\tau_n)^\circ = (x_1{:}\tau_1^\circ,\ldots,x_n{:}\tau_n^\circ)$

Erasure Π° of programs Π

$dom(\Pi^\circ) = dom(\Pi)$
$\Pi^\circ(f) = \boldsymbol{\lambda} e : \sigma^\circ \to \sigma'^\circ$ if $\Pi(f) = \boldsymbol{\lambda} e : \sigma \to \sigma'$
$\Pi^\circ(f) = \sigma^\circ \to \sigma'^\circ$ if $\Pi(f) = \sigma \to \sigma'$

Erasure Θ_0° of effect environment Θ_0

$dom(\Theta_0^\circ) = dom(\Theta_0)$
$\Theta_0^\circ(\mathbf{enable}) = \top \to r' \subseteq r$
$\Theta_0^\circ(f) = \top \to \top$ if $f \in \{\mathbf{init}, \mathbf{split}, \mathbf{join}, \mathbf{consume}\} \cup \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\}$
$\Theta_0^\circ(f) = \Theta_0(f)$ otherwise

Erasure α° of Γ-valuations α

$dom(\alpha^\circ) = dom(\alpha)$
$\alpha^\circ(x) = \star$ if $\Gamma(x) = \mathbf{mgr}$
$\alpha^\circ(x) = \alpha(x)$ otherwise

Erasure S° of typed stacks S

$\lightning^\circ = \lightning$ $\quad \epsilon^\circ = \epsilon$ $\quad (\langle x_1, \ldots, x_n | \alpha, e \rangle^\Gamma :: S)^\circ = \langle x_1, \ldots, x_n | \alpha^\circ, e \rangle^{\Gamma^\circ} :: S^\circ$

Fig. 6. Erasure of resource managers.

Erasure on types determines erasure on product types, type environments, programs and valuations (where erasure uniformly maps the values of $\mathbf{mgr}$-variables to $\star$, the only value of type $\mathbf{unit}$), which in turn determines erasure on typed stacks. As outlined above, erasure on effect environments trivialises the effect of resource manager built-ins, except $\mathbf{enable}$, and preserves the effects of all built-ins not operating on managers. The effect of $\mathbf{enable}$ after erasure is to non-deterministically choose a sub-multiset of r and return its complement in r'. This reflects the fact that calls to $\mathbf{enable}$ provide points of interaction for the policy (e. g., the user) to decide how many resources the system is granted. Erasing resource managers does not mean that policy decisions are fixed, it just removes the managers' book keeping about those decisions.

Lemma 6. *Let Π be a well-typed program and S a typed Π-stack. Then Π° is a well-typed program and S° a typed Π°-stack.*

Erasure makes trivial the effects of $\mathbf{assertEmpty}$ and $\mathbf{assertAtLeast}$, and in particular, replaces their precondition by $\top$. Thus a program cannot go wrong after erasure, as rule (R-let$^{\lightning}_2$) will never apply.

Proposition 7. *Let Π be a well-typed program and S a Π°-stack S. Then $S \not\rightsquigarrow^*_{\Pi^\circ} \lightning$.*

The next result states that the small-step reduction relation $\rightsquigarrow_\Pi$ of a program Π is almost bisimulation equivalent to the reduction relation $\rightsquigarrow_{\Pi^\circ}$ of its erasure. In fact, it shows that the relation $R = \{\langle S, S^\circ\rangle \mid S \text{ is a } \Pi\text{-stack}\}$ would be a bisimulation if $\rightsquigarrow_\Pi$ could not reduce stacks to the error stack $\lightning$. Put differently, if Π cannot go wrong then $\rightsquigarrow_\Pi$ and $\rightsquigarrow_{\Pi^\circ}$ are bisimulation equivalent. The proof of this theorem is by case analysis on the reduction relation $\rightsquigarrow_\Pi$ of the unerased program. As a corollary, we get that reachability in the erased program is essentially the same as reachability in the unerased one, provided that the unerased program cannot go wrong.

Theorem 8. *Let Π be a well-typed program and $\hat{S}_0$ a typed Π-stack with $\hat{S}_0 \not\rightsquigarrow_\Pi \lightning$.*

1. *For all typed Π-stacks $\hat{S}_1$, if $\hat{S}_0 \rightsquigarrow_\Pi \hat{S}_1$ then $\hat{S}_0^\circ \rightsquigarrow_{\Pi^\circ} \hat{S}_1^\circ$.*
2. *For all typed Π°-stacks S_1, if $\hat{S}_0^\circ \rightsquigarrow_{\Pi^\circ} S_1$ then there is a typed Π-stack $\hat{S}_1$ such that $\hat{S}_0 \rightsquigarrow_\Pi \hat{S}_1$ and $\hat{S}_1^\circ = S_1$.*

Corollary 9. *Let Π be a well-typed program and S_0 a typed Π-stack. If $S_0 \not\rightsquigarrow^*_\Pi \lightning$ then $\{S^\circ \mid S_0 \rightsquigarrow^*_\Pi S\} = \{S \mid S_0^\circ \rightsquigarrow^*_{\Pi^\circ} S\}$.*

What distinguishes erasure of resource managers from other erasure results (e. g., type erasure during compilation, Java generics erasure) is that here, erasure does not completely remove a language construct. Instead, it removes the book keeping but retains the semantically important bit that deals with dynamic policy decisions.

2.5 Big-step Relational Semantics

The reduction semantics presented in Section 2.3 is good for showing preservation properties, like the preservation of resources. However, it does not easily yield a relational view on functions, relating input and output parameters. This is achieved by a relational semantics, which we will prove equivalent to the reduction semantics. Contrary to the reduction semantics, which was originally untyped and had type environments added conservatively, the relational semantics will be typed from the start. (Types do not hurt here, as the relational semantics is not geared towards execution.)

Throughout this section, we assume that Π is a well-typed program. A *state* β is either the error state $\lightning$ or a normal state $\langle \Gamma; \alpha\rangle$, where Γ is a linear type environment and α a maximal Γ-valuation. Given an expression e, a normal state $\langle \Gamma; \alpha\rangle$ and a state β', we define the judgement $e, \langle \Gamma; \alpha\rangle \Downarrow_\Pi \beta'$ (or $e, \langle \Gamma; \alpha\rangle \Downarrow \beta'$ if Π is understood) by the rules in Figure 7 if $dom(\Gamma) \cap bound(e) = \emptyset$ and there are Γ_e and σ such that $\Gamma \succeq \Gamma_e$ and $\Gamma_e \vdash e : \sigma$. The intended meaning of $e, \langle \Gamma; \alpha\rangle \Downarrow \beta'$ is that evaluating expression e in state $\langle \Gamma; \alpha\rangle$ may terminate and result in state β'.

The reduction semantics deallocates variables once they become unused (an eager garbage collection, so to say), which is essential for the linear variables as otherwise resource preservation would not hold. However, the intermediate values of variables are thus lost. In contrast, the relational semantics names and records all intermediate values, even the linear ones, as $e, \langle \Gamma; \alpha\rangle \Downarrow \langle \Gamma'; \alpha'\rangle$ implies $\Gamma' \succeq \Gamma$ and $\alpha' \succeq \alpha$.

By definition, violations of resource safety manifest themselves in reductions ending in the error stack, and hence reductions which diverge or get stuck cannot

Evaluation of expressions $e, \langle \Gamma; \alpha \rangle \Downarrow \beta'$

$$\text{(E-ret)}\ \frac{}{\mathbf{ret}\ (x_1,\ldots,x_n), \langle \Gamma; \alpha \rangle \Downarrow \langle \Gamma; \alpha \rangle}$$

$$\text{(E-let}_1\text{)}\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to (z'_1{:}\tau'_1,\ldots,z'_n{:}\tau'_n) \qquad \Gamma_f = z_1{:}\tau_1,\ldots,z_m{:}\tau_m \\ \alpha_f = \{z_1 \mapsto \alpha(v_1),\ldots,z_m \mapsto \alpha(v_m)\} \qquad e, \langle \Gamma_f; \alpha_f \rangle \Downarrow \langle \Gamma'_f; \alpha'_f \rangle \\ \Gamma' = \Gamma, x'_1{:}\tau'_1,\ldots,x'_n{:}\tau'_n \qquad \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1),\ldots,x'_n \mapsto \alpha'_f(z'_n)\} \\ e', \langle \Gamma'; \alpha' \rangle \Downarrow \beta''\end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \Gamma; \alpha \rangle \Downarrow \beta''}$$

$$\text{(E-let}_1^{\lightning}\text{)}\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \qquad \Gamma_f = z_1{:}\tau_1,\ldots,z_m{:}\tau_m \\ \alpha_f = \{z_1 \mapsto \alpha(v_1),\ldots,z_m \mapsto \alpha(v_m)\} \qquad e, \langle \Gamma_f; \alpha_f \rangle \Downarrow \lightning\end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \Gamma; \alpha \rangle \Downarrow \lightning}$$

$$\text{(E-let}_2\text{)}\ \frac{\begin{array}{c}\Pi(f) = (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to (z'_1{:}\tau'_1,\ldots,z'_n{:}\tau'_n) \\ \alpha_f = \{z_1 \mapsto \alpha(v_1),\ldots,z_m \mapsto \alpha(v_m)\} \qquad \alpha'_f \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f) \qquad \alpha'_f \succeq \alpha_f \\ \Gamma' = \Gamma, x'_1{:}\tau'_1,\ldots,x'_n{:}\tau'_n \qquad \alpha' = \alpha\{x'_1 \mapsto \alpha'_f(z'_1),\ldots,x'_n \mapsto \alpha'_f(z'_n)\} \\ e', \langle \Gamma'; \alpha' \rangle \Downarrow \beta''\end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \Gamma; \alpha \rangle \Downarrow \beta''}$$

$$\text{(E-let}_2^{\lightning}\text{)}\ \frac{\begin{array}{c}\Pi(f) = \boldsymbol{\lambda} e : (z_1{:}\tau_1,\ldots,z_m{:}\tau_m) \to \sigma' \qquad f \in \{\mathbf{assertEmpty}, \mathbf{assertAtLeast}\} \\ \alpha_f = \{z_1 \mapsto \alpha(v_1),\ldots,z_m \mapsto \alpha(v_m)\} \qquad \forall \alpha'_f \in \mathit{Eff}^{\Pi_0}_{\Theta_0}(f) : \alpha'_f \not\succeq \alpha_f\end{array}}{\mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e', \langle \Gamma; \alpha \rangle \Downarrow \lightning}$$

$$\text{(E-if}_1\text{)}\ \frac{e_1, \langle \Gamma; \alpha \rangle \Downarrow \beta'}{\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \langle \Gamma; \alpha \rangle \Downarrow \beta'}\ \text{if } \alpha(v) \neq 0$$

$$\text{(E-if}_2\text{)}\ \frac{e_2, \langle \Gamma; \alpha \rangle \Downarrow \beta'}{\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2, \langle \Gamma; \alpha \rangle \Downarrow \beta'}\ \text{if } \alpha(v) = 0$$

Fig. 7. Big-step evaluation relation (for a fixed program Π).

violate resource safety. Therefore, resource safety is not affected by the fact that the relational semantics ignores such reductions. Under this proviso, Proposition 10 shows the equivalence of reduction and relational semantics.

Proposition 10. *Let $\langle \Gamma; \alpha \rangle$ and $\langle \Gamma'; \alpha' \rangle$ be states. Let e be an expression such that $dom(\Gamma) = free(e)$ and $\Gamma \vdash e : \sigma$ for some product type σ. Then*

1. *$e, \langle \Gamma; \alpha \rangle \Downarrow \lightning$ if and only if $\langle |\alpha, e\rangle^{\Gamma} :: \epsilon \rightsquigarrow^* \lightning$, and*
2. *$e, \langle \Gamma; \alpha \rangle \Downarrow \langle \Gamma'; \alpha' \rangle$ if and only if there is a typed stack $\langle |\alpha'', \mathbf{ret}\ (x_1,\ldots,x_n)\rangle^{\Gamma''} :: \epsilon$ such that $\langle |\alpha, e\rangle^{\Gamma} :: \epsilon \rightsquigarrow^* \langle |\alpha'', \mathbf{ret}\ (x_1,\ldots,x_n)\rangle^{\Gamma''} :: \epsilon$ and $\Gamma' \succeq \Gamma''$ and $\alpha' \succeq \alpha''$.*

3 Effect Type System

In this section, we will develop a type system to statically guarantee dynamic resource safety, i. e., the absence of reductions to the error stack $\lightning$. We will do so by annotating

functions with effects and then extending the notion of effect to a judgement on expressions, which we will define by a simple set of typing rules.

3.1 Effect Type System

We extend the notion of effect $\phi \rightarrow \psi$ from built-in functions to λ-abstractions. To be precise, $\phi \rightarrow \psi$ is an *effect* for f if $\Gamma \vdash \phi$ and $\Gamma, \Delta \vdash \psi$, where $\Pi(f) = [\boldsymbol{\lambda} \ldots] \Gamma \rightarrow \Delta$, regardless of whether f is built-in or a λ-abstraction. In line with this extension, an *effect environment* Θ maps all functions $f \in dom(\Pi)$ to effects $\Theta(f)$ for f.

In order to derive the effects of λ-abstractions, we generalise effects to effect types for expressions and develop a type system for inductively constructing such effect types. Effects relate input and output parameters of functions by logical formulae. Likewise, effect types shall relate input and output parameters of expressions. Here, the input parameters of an expression are its free variables; the output parameters are those variables that are not free yet but will become free during reduction, i. e., the (let-)bound variables. Formally, an *effect type* $\Gamma; \phi \rightarrow \Delta; \psi$ is a pair of constraints ϕ and ψ together with a pair of type environments Γ and Δ such that $dom(\Gamma) \cap dom(\Delta) = \emptyset$ and $\Gamma \vdash \phi$ and $\Gamma, \Delta \vdash \psi$. We call ϕ and ψ *precondition* and *action*, and Γ and Δ *input* and *output (parameters)*, respectively. Given an expression e, we say that an effect type $\Gamma; \phi \rightarrow \Delta; \psi$ is an *effect type for* e if $dom(\Gamma) \cap bound(e) = \emptyset$.

We say that an effect type $\Gamma; \phi \rightarrow \Delta; \psi$ is *stronger than* an effect type $\Gamma'; \phi' \rightarrow \Delta'; \psi'$, denoted by $\Gamma; \phi \rightarrow \Delta; \psi \supseteq \Gamma'; \phi' \rightarrow \Delta'; \psi'$, if $\phi' \models \phi$ and $(\phi' \wedge \psi) \models \psi'$, i. e., the stronger effect type $\Gamma; \phi \rightarrow \Delta; \psi$ has a weaker precondition but stronger action. The stronger-than relation $\supseteq$ is a quasi-order, i. e., reflexive and transitive, and induces an equivalence relation on effect types, the *as-strong-as* relation, which we denote by $\equiv$. Note that for every effect type $\Gamma; \phi \rightarrow \Delta; \psi$ is as strong as an effect type $\Gamma'; \phi \rightarrow \Delta'; \psi$ with linear type environments Γ' and Δ'.

Figure 8 presents the typing rules for deriving effect types. There, the judgement $\Theta \vdash_\Pi e : \Gamma; \phi \rightarrow \Delta; \psi$ states that expression e has effect type $\Gamma; \phi \rightarrow \Delta; \psi$ in the context of program Π and effect environment Θ. If Π is understood, we may omit it and write $\Theta \vdash e : \Gamma; \phi \rightarrow \Delta; \psi$ instead. The judgement $\Pi, \Theta \vdash f$ means that the effect type ascribed to a λ-abstraction f by Θ and Π is consistent with the effect type derived for the body of f. We say that Θ is an *admissible* effect environment for a program Π if $\Pi, \Theta \vdash f$ for all λ-abstractions $f \in dom(\Pi) \setminus dom(\Pi_0)$.

Lemma 11. *Let e be an expression, Θ an effect environment (referring to an implicit program Π) and $\Gamma; \phi \rightarrow \Delta; \psi$ an effect type. If $\Theta \vdash e : \Gamma; \phi \rightarrow \Delta; \psi$ then $\Gamma; \phi \rightarrow \Delta; \psi$ is an effect type for e.*

Theorem 12 states soundness of effect typing w. r. t. the big-step relational semantics. The proof is by double induction on the derivation of relational semantics judgements over the derivation of effect type judgements. As a corollary, we get that reduction starting from a state that satisfies the precondition can't go wrong, hence resource managers can be erased. In fact, the untyped reductions in the erased program match exactly the typed reductions in the original program.

Typing of expression effects $\Theta \vdash e : \Gamma; \phi \to \Delta; \psi$

$$\text{(ET-weak)}\ \frac{\Theta \vdash e : \Gamma; \phi \to \Delta; \psi}{\Theta \vdash e : \Gamma'; \phi' \to \Delta'; \psi'}\ \text{if}\ \begin{cases} dom(\Gamma') \cap bound(e) = \emptyset \wedge \\ \Gamma; \phi \to \Delta; \psi \supseteq \Gamma'; \phi' \to \Delta'; \psi' \end{cases}$$

$$\text{(ET-ret)}\ \frac{}{\Theta \vdash \mathbf{ret}\ (x_1,\ldots,x_n) : \emptyset; \top \to \emptyset; \top}$$

$$\text{(ET-if)}\ \frac{\Theta \vdash e_1 : \Gamma; v \not\approx 0 \wedge \phi \to \Delta; \psi \qquad \Theta \vdash e_2 : \Gamma; v \approx 0 \wedge \phi \to \Delta; \psi}{\Theta \vdash \mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \Gamma; \phi \to \Delta; \psi}$$

$$\text{(ET-let)}\ \frac{\begin{array}{c} \Pi(f) = [\boldsymbol{\lambda}\ldots]\Gamma \to \Delta \qquad \Gamma = z_1{:}\tau_1, \ldots, z_m{:}\tau_m \qquad \Delta = z'_1{:}\tau'_1, \ldots, z'_n{:}\tau'_n \\ \Theta(f) = \phi \to \psi \qquad \mu = \{z_1 \mapsto v_1, \ldots, z_m \mapsto v_m, z'_1 \mapsto x'_1, \ldots, z'_n \mapsto x'_n\} \\ \Theta \vdash e' : \Gamma', \Delta'; \phi' \wedge \psi' \to \Delta''; \psi'' \end{array}}{\Theta \vdash \mathbf{let}\ (x'_1,\ldots,x'_n) = f\ (v_1,\ldots,v_m)\ \mathbf{in}\ e' : \Gamma'; \phi' \to \Delta', \Delta''; \psi' \wedge \psi''}\ \text{if}\ (*)$$

$$\text{where}\ (*)\ \begin{cases} dom(\Gamma') \cap \{x'_1, \ldots, x'_n\} = \emptyset \wedge \\ \Gamma\mu; \phi\mu \to \Delta\mu; \psi\mu \supseteq \Gamma'; \phi' \to \Delta'; \psi' \end{cases}$$

Well-typedness of λ**-abstraction effects** $\Pi, \Theta \vdash f$

$$\text{(ET-lam)}\ \frac{\Pi(f) = \boldsymbol{\lambda} e : \Gamma \to \Delta \qquad \Theta(f) = \phi \to \psi \qquad \Theta \vdash e : \Gamma; \phi \to \Delta; \psi}{\Pi, \Theta \vdash f}$$

Fig. 8. Typing rules for effect types (for a fixed program Π).

Theorem 12. *Let Θ be an admissible effect environment for a well-typed program Π. Let e be an expression and $\Gamma; \phi \to \Delta; \psi$ an effect type such that $\Theta \vdash e : \Gamma; \phi \to \Delta; \psi$. Let $\langle \Gamma; \alpha \rangle$ and β' be states such that $e, \langle \Gamma; \alpha \rangle \Downarrow \beta'$ (which implies $\Gamma_e \vdash e : \sigma$ for some Γ_e, σ). If $\alpha \models \phi$ then $\beta' = \langle \Gamma'; \alpha' \rangle$ for some Γ' and α' such that $\alpha' \models \phi \wedge \psi$. (In particular, if $\alpha \models \phi$ then $\beta' \neq \lightning$.)*

Corollary 13. *Let Θ be an admissible effect environment for a well-typed program Π. Let e be an expression and $\Gamma; \phi \to \Delta; \psi$ an effect type such that $\Theta \vdash_\Pi e : \Gamma; \phi \to \Delta; \psi$. Let α be a maximal Γ-valuation, and let $\hat{S}_0 = \langle |\alpha|_{free(e)}, e \rangle^{\Gamma|_{free(e)}} :: \epsilon$ be a typed Π-stack (which implies $\Gamma|_{free(e)} \vdash_\Pi e : \sigma$ for some σ). If $\alpha \models \phi$ then*

1. *$\hat{S}_0 \not\rightsquigarrow^*_\Pi \lightning$ and*
2. *for all (untyped) Π°-stacks S, $\hat{S}_0^{\circ\natural} \rightsquigarrow^*_{\Pi^\circ} S$ if and only if there is a typed Π-stack $\hat{S}$ such that $\hat{S}_0 \rightsquigarrow^*_\Pi \hat{S}$ and $\hat{S}^{\circ\natural} = S$. (In particular, $\hat{S}_0^{\circ\natural} \not\rightsquigarrow^*_{\Pi^\circ} \lightning$.)*

3.2 Example: Bulk Messaging Application

To illustrate the use of the effect type system, we revisit the example from Figure 1. The interesting bits of code are in the functions send_bulk and send_msg.

The function send_bulk first builds up a multiset of resources r by converting the strings representing phone numbers in nums into resources. Next it attempts to authorise the use of all resources by having enable add r to an empty resource manager m. If this

f	$\Theta(f)$
send_bulk	$\top \to \top$
res_from_nums	$\top \to \mathsf{r} \approx bagof(map_{fromstr}(\mathsf{nums}))$
res_from_nums'	$0 \le \mathsf{i} \le len(\mathsf{nums}) \wedge \mathsf{r'} \approx bagof(map_{fromstr}(subarray(\mathsf{nums}, \mathsf{i}, len(\mathsf{nums}))))$ $\to \mathsf{r} \approx bagof(map_{fromstr}(\mathsf{nums}))$
send_msgs	$bagof(map_{fromstr}(\mathsf{nums})) \subseteq \mathsf{m} \to \mathsf{m} \approx \mathsf{m'} \uplus bagof(map_{fromstr}(\mathsf{nums}))$
send_msgs'	$0 \le \mathsf{i} \le len(\mathsf{nums}) \wedge bagof(map_{fromstr}(subarray(\mathsf{nums}, 0, \mathsf{i}))) \subseteq \mathsf{m}$ $\to \mathsf{m} \approx \mathsf{m'} \uplus bagof(map_{fromstr}(subarray(\mathsf{nums}, 0, \mathsf{i})))$
send_msg	$count(\mathsf{m}, fromstr(\mathsf{num})) \ge 1 \to \mathsf{m} \approx \mathsf{m'} \uplus \{\!\{fromstr(\mathsf{num}){:}1\}\!\}$
prim_send_msg	$\top \to \top$

$\forall a : len(map_{fromstr}(a)) \approx len(a)$
$\forall a \forall i : 0 \le i < len(a) \Rightarrow map_{fromstr}(a)[i] \approx fromstr(a[i])$

$\forall a \forall j \forall k : 0 \le j \le k \le len(a) \Rightarrow len(subarray(a, j, k)) = k + (-j)$
$\forall a \forall j \forall k \forall i : 0 \le j \le k \le len(a) \wedge 0 \le i < len(subarray(a, j, k)) \Rightarrow subarray(a, j, k)[i] = a[j + i]$

$\forall a : |bagof(a)| \approx len(a)$
$\forall a : len(a) \approx 1 \Rightarrow bagof(a) \approx \{a[0]{:}1\}$
$\forall a \forall k : 0 \le k \le len(a) \Rightarrow bagof(a) \approx bagof(subarray(a, 0, k)) \uplus bagof(subarray(a, k, len(a)))$

Fig. 9. Bulk messaging application: admissible effect environment Θ and axiomatisation of theory extension; for the sake of readability sort information is suppressed in the axioms.

fails, i. e., the multiset r' returned by enable is of non-zero size, send_bulk terminates (after destroying m' and whatever resources it holds).[3] If authorising all resources succeeds, send_bulk calls send_msgs to actually send the messages while checking that the manager m' contains the required resources. After that, send_bulk checks that send_msgs has used up all resources by asserting that the returned manager m'' is empty; failing this assertion will trigger a runtime error. Finally, send_bulk explicitly destroys the empty manager m''' and terminates.

The function send_msg sends one message, checking whether the resource manager m holds the resource required. It does so by converting the string num into a singleton multiset of resources r. Then it splits the manager m into m' and m_r, so that m_r contains at most the resources in r. Next, send_msg asserts that m_r contains at least r; failing this assertion will trigger a runtime error. Succeeding the assertion, send_msg calls the primitive send function, destroys the now used resource by consuming m_r', and returns the remaining resources in the manager m'.

The bulk messaging example is statically resource safe, as witnessed by the admissible effect environment displayed in Figure 9. Of particular interest is the effect $\top \to \top$ ascribed to the main function send_bulk. This least informative effect expresses nothing about the function itself but implies the absence of runtime errors via Corollary 13.

The effects require an extension of the theory $\mathcal{T}$ (see Section 2.2) by three new functions, axiomatised in Figure 9. The function map maps an array of strings to an

[3] A more sophisticated version of the application could deal more gracefully with enable granting only part of the requested resources. This would require more complex code to inspect the multisets r and r' (but not the resource manager m').

array of resources, $subarray$ takes an array and cuts out the sub-array between two given indices, and $bagof$ converts an array of resources to a multiset (containing the same elements with the same multiplicity). Note that the axiomatisation of $bagof$ is not complete[4] but sufficient for our purposes.

Effect type checking, e. g., for checking admissibility of the effect environment Θ from Figure 9, requires checking the side condition of the weakening rule (ET-weak), which involves checking logical entailment w. r. t. to an extension of the theory $\mathcal{T}$. Due to the high undecidability of $\mathcal{T}$, we actually check entailment w. r. t. (an extension of) an approximation of $\mathcal{T}$; in particular, we approximate multiplication and division by uninterpreted functions. For the bulk messaging example, we used an SMT solver [4] that can handle linear integer arithmetic and arrays. We added axioms for multisets and the axioms in Figure 9. Due to an incomplete quantifier instantiation heuristic, we had to instantiate a number of these axioms by hand, yet eventually, the solver was able to prove all the entailments required by the weakening rules.

Even though arising from a single example, we believe that the extension of the theories of multisets and arrays with the functions $subarray$ and $bagof$ is quite generic and could prove useful in many cases.

4 Conclusion

We have presented a programming language with support for complex resource management, close to the standard SSA/ANF forms of compiler intermediate languages [1]. By construction, programs are *dynamically resource safe* in that any attempts to abuse resources are trapped. We have extended the language with an effect type system which guarantees the for well-typed programs no such attempts occur: we have *static resource safety*. In addition, for such programs the bookkeeping required by dynamic resource management can be erased.

Related Work. Many tools and methods have been proposed to assist with resource management at runtime, e.g., in Java, the JRes [9] and J-Seal [8] frameworks. Generally, these aim to enable programs to react to fluctuations of resources caused by an unpredictable environment. Our aim, however is to track the flow of resources through the program, where the environment can influence the availability of resources only at well-understood points of interaction with the program and with clear availability policies. This offers the chance for more precise resource control whose behaviour can be predicted statically.

This paper builds on previous work [3] with a Java library implementing resource managers and focusing on the dynamic aspects of resource management policies. This Java library supports essentially the same operations on resource managers as our functional language, except that state is realised by destructive updates instead of linear types. While [3] does not provide a static analysis to prove static resource safety, it does outline how dynamic accounting could be erased if static resource safety were provable. Our work here shows one way to do just that.

[4] A complete axiomatisation of $bagof$ is possible in the full first-order theory of multisets and arrays but it is much more complicated and unusable in practise.

Our approach is in line with a general trend of providing the programmer with language-based mechanisms for security and additional static analyses (often using type systems) which use these mechanisms. This combination provides a desirable graceful degradation: if static analysis succeeds in proving certain properties, then the program may be optimised without affecting security. Yet, even if the analyses fail the language based mechanisms will enforce the security properties at runtime.

The context of our work is the MOBIUS project [5] on proof-carrying code (PCC) for mobile devices. Our effect type system is very simple and in principle well-suited for a PCC setting where checkers themselves are resource bounded. However, the weakening rule relies on checking logical entailment in a first-order theory, which is undecidable in general. Therefore, a certificate for PCC need not only provide a type derivation tree but also proofs (in some proof system) for the entailment checks in the weakening rule. The development of a suitable such proof system is a topic for further research, as is the investigation of decidable fragments of relevant first-order theories.

Acknowledgements. This work was funded in part by the Sixth Framework programme of the European Community under the MOBIUS project FP6-015905. This paper reflects only the authors' views and the European Community is not liable for any use that may be made of the information contained therein. Ian Stark was also supported by an Advanced Research Fellowship from the UK Engineering and Physical Sciences Research Council, EPSRC project GR/R76950/01.

References

[1] A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.
[2] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theoret. Comput. Sci.*, 389(3):411–445, 2007.
[3] D. Aspinall, P. Maier, and I. Stark. Monitoring external resources in Java MIDP. *Electron. Notes Theor. Comput. Sci.*, 197:17–30, 2008.
[4] C. Barrett, L. de Moura, and A. Stump. Design and results of the 2nd annual satisfiability modulo theories competition. *Form. Meth. Syst. Des.*, 31(3):221–239, 2007.
[5] G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. MOBIUS: Mobility, ubiquity, security. Objectives and progress report. In *Proc. TGC 2006*, LNCS 4661, pp.10–29. Springer, 2007.
[6] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In *Proc. LPAR 2004*, LNCS 3452, pp.347–362. Springer, 2005.
[7] F. Besson, G. Dufay, and T. P. Jensen. A formal model of access control for mobile interactive devices. In *Proc. ESORICS 2006*, LNCS 4189, pp.110–126. Springer, 2006.
[8] W. Binder, J. Hulaas, and A. Villazón. Portable resource control in Java. In *Proc. OOPSLA 2001*, pp.139–155. ACM, 2001.
[9] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proc. OOPSLA '98*, pp.21–35. ACM, 1998.
[10] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. PLDI '93*, pp.237–247. ACM, 1993.
[11] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *Proc. ESOP 2007*, LNCS 4421, pp.189–204. Springer, 2007.
[12] Unknown. Redbrowser.A, Feb. 2006. J2ME trojan, variously identified in the wild as *Redbrowser.A* (F-Secure), *J2ME/Redbrowser.a* (McAfee), *Trojan.Redbrowser.A* (Symantec), *Trojan-SMS.J2ME.Redbrowser.a* (Kaspersky Lab).

Termination Analysis of Java Bytecode

Elvira Albert[1], Puri Arenas[1], Michael Codish[2],
Samir Genaim[3], Germán Puebla[3], and Damiano Zanardini[3]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] CS, Ben-Gurion University of the Negev, Israel
[3] CLIP, Technical University of Madrid (UPM), Spain

Abstract. Termination analysis has received considerable attention, traditionally in the context of declarative programming, and recently also for imperative languages. In existing approaches, termination is performed on source programs. However, there are many situations, including mobile code, where only the compiled code is available. In this work we present an automatic termination analysis for sequential Java Bytecode programs. Such analysis presents all of the challenges of analyzing a low-level language as well as those introduced by object-oriented languages. Interestingly, given a bytecode program, we produce a *constraint logic* program, CLP, whose termination entails termination of the bytecode program. This allows applying the large body of work in termination of CLP programs to termination of Java bytecode. A prototype analyzer is described and initial experimentation is reported.

1 Introduction

It has been known since the pre-computer era that it is not possible to write a program which correctly decides, in all cases, if another program will *terminate*. However, termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Automated techniques are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. This information is used for specifying a *ranking function* which strictly decreases on a well-founded domain on each computation step, thus guaranteeing termination.

In the last two decades, a variety of sophisticated termination analysis tools have been developed, primarily for less-widely used programming languages. These include analyzers for term rewrite systems [15], and logic and functional languages [18,10,17]. Termination-proving techniques are also emerging in the imperative paradigm [6,11,15], even for dealing with large industrial code [11].

Static analysis of *Java ByteCode* (JBC for short) has received considerable attention lately [25,23,24,22,1]. The present paper presents a static analysis for sequential JBC which is, to the best of our knowledge, the first approach to proving termination. Bytecode is a low-level representation of a program, designed to be executed by a virtual machine rather than by dedicated hardware. As such, it is usually higher level than actual machine code, and independent of

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 2–18, 2008.

the specific hardware. This, together with its security features, makes JBC [19] the chosen language for many *mobile code* applications. In this context, analysis of JBC programs may enable performing a certain degree of static (i.e., before execution) verification on program components obtained from untrusted providers. *Proof-Carrying Code* [20] is a promising approach in this area: mobile code is equipped with certain *verifiable evidence* which allows deployers to independently verify properties of interest about the code. Termination analysis is also important since the verification of functional program properties is often split into separately proving partial correctness and termination.

Object-oriented languages in general, and their low-level (bytecode) counterparts in particular, present new challenges to termination analyzers: (1) loops originate from different sources, such as conditional and unconditional jumps, method calls, or even exceptions; (2) *size measures* must consider primitive types, user defined objects, and arrays; and (3) *tracking* data is more difficult, as data can be stored in variables, operand stack elements or heap locations.

Analyzing JBC is a necessity in many situations, including mobile code, where the user only has access to compiled code. Furthermore, it can be argued that analyzing low-level programs can have several advantages over analyzing their high-level (Java) counterparts. One advantage is that low-level languages typically remain stable, as their high-level counterparts continue to evolve — analyzers for bytecode programs need not be enhanced each time a new language construct is introduced. Another advantage is that analyzing low-level code narrows the gap between what is verified and what is actually executed. This is relevant, for example, in safety critical applications.

In this paper we take a semantic-based approach to termination analysis, based on two steps. The first step transforms the bytecode into a *rule-based* program where all loops and all variables are represented uniformly, and which is semantically equivalent to the bytecode. This rule-based representation is based on previous work [1] in cost analysis, and is presented in Sec. 2. In the second step (Sec. 3), we adapt directly to the rule-based program standard techniques which usually prove termination of high-level languages. Sec. 4 reports on our prototype implementation and validates it by proving termination of a series of object-oriented benchmarks, containing recursion, nested loops and data structures such as trees and arrays. Conclusions and related work are presented in Sec. 5.

2 Java Bytecode and Its Rule-Based Representation

We consider a subset of the Java Virtual Machine (JVM) language which handles integers and object creation and manipulation (by accessing fields and calling methods). For simplicity, exceptions, arrays, interfaces, and primitive types besides integers are omitted. Yet, these features can be easily handled within our setting: all of them are implemented in our prototype and included in benchmarks in Table 1. A full description of the JVM [19] is out of the scope of this paper.

A sequential JBC program consists of a set of *class files*, one for each class, partially ordered with respect to the subclass relation $\preceq$. A class file contains

information about its name, the class it extends, and the fields and methods it defines. Each method has a unique signature m from which we can obtain the class, denoted $class(m)$, where the method is defined, the name of the method, and its signature. When it is clear from the context, we ignore the class and the types parts of the signature. The bytecode associated with m is a sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n \rangle$, where each b_i is a *bytecode instruction*, and pc_i is its address. The local variables of a method are denoted by $\langle l_0, \ldots, l_{n-1} \rangle$, of which the first $k \leq n$ are the formal parameters, and l_0 corresponds to the *this* reference (unlike Java, in JBC, the *this* reference is explicit). Similarly, each field f has a unique signature, from which we can obtain its name and the name of the class it belongs to. The bytecode instructions we consider include:

$$
\begin{array}{rl}
bcInst ::= & \mathsf{istore}\ v \mid \mathsf{astore}\ v \mid \mathsf{iload}\ v \mid \mathsf{aload}\ v \mid \mathsf{iconst}\ i \mid \mathsf{aconst\_null} \\
\mid & \mathsf{iadd} \mid \mathsf{isub} \mid \mathsf{iinc}\ v\ n \mid \mathsf{imul} \mid \mathsf{idiv} \\
\mid & \mathsf{if\_}\phi\ pc \mid \mathsf{goto}\ pc \mid \mathsf{ireturn} \mid \mathsf{areturn} \\
\mid & \mathsf{new}\ c \mid \mathsf{invokevirtual}\ m \mid \mathsf{invokespecial}\ m \mid \mathsf{getfield}\ f \mid \mathsf{putfield}\ f
\end{array}
$$

where c is a class, ϕ is a comparison condition on numbers (ne, le, icmpgt) or references (null, nonnull), v is a local variable, i is an integer, and pc is an instruction address. Briefly, instructions are: (row 1) stack operations referring to constants and local variables; (row 2) arithmetic operations; (row 3) jumps and method return; and (row 4) object-oriented instructions. All instructions in row 3, together with invokevirtual, are *branching* (the others are *sequential*). For simplicity, we will assume all methods to return a value. Fig. 1 depicts the bytecode for the iterative method *fact*, where indexes $0, \ldots, 3$ stands for local variables *this*, n, *ft* and i respectively. $next(pc)$ is the address immediately after the program counter pc. As instructions have different sizes, addresses do not always increase by one (e.g., $next(6){=}9$).

We assume an operational semantics which is a subset of the JVM specification [19]. The execution environment of a bytecode program consists of a *heap* h and a stack A of *activation records*. Each activation record contains a program counter, a local operand stack, and local variables. The heap contains all objects (and arrays) allocated in the memory. Each method invocation generates a new activation record according to its signature. Different activation records do not share information, but may contain references to the same object in the heap.

2.1 From Bytecode to Control Flow Graphs

The JVM language is unstructured. It allows conditional and unconditional jumps as well as other implicit sources of branching, such as virtual method invocation and exception throwing. The notion of a *Control Flow Graph* (CFG for short) is a well-known instrument which facilitates reasoning about programs in unstructured languages. A CFG is similar to the older notion of a flow chart, but CFGs include a concept of "call to" and "return from". Methods in the bytecode program are represented as CFGs, and calls from one method to another correspond to calls between these graphs. In order to build CFGs, the first step

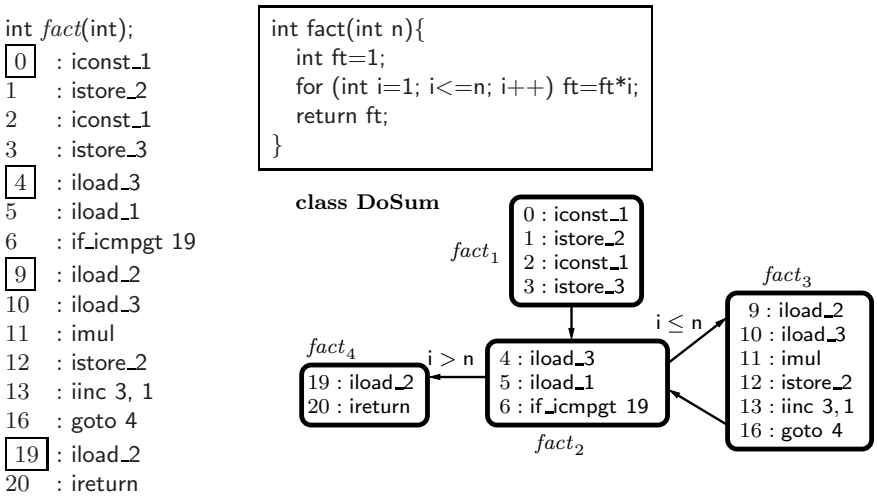


Fig. 1. A JBC method (left) with its corresponding source (center) and its CFG (right)

is to partition a sequence of bytecode instructions into a set of maximal subsequences, or basic blocks, of instructions which execute sequentially, i.e., without branching nor jumping. Given a bytecode instruction $pc{:}b$, we say that $pc'{:}b'$ is a *predecessor* of $pc{:}bc$ if one of the following conditions holds: (1) b'=goto pc, (2) b'=if_ϕ pc, (3) $next(pc')$=pc.

Definition 1 (partition to basic blocks). *Given a method m and its sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n\rangle$, a partition into basic blocks $m_1, \ldots, m_k$ takes the form*

$$\underbrace{pc_{i_1}{:}b_{i_1}, \ldots, pc_{f_1}{:}b_{f_1}}_{m_1}, \underbrace{pc_{i_2}{:}b_{i_2} \ldots, pc_{f_2}{:}b_{f_2}}_{m_2}, \ldots \underbrace{pc_{i_k}{:}b_{i_k} \ldots, pc_{f_k}{:}b_{f_k}}_{m_k}$$

where i_1=1, f_k=n and

1. *the number of basic blocks (i.e. k) is minimal;*
2. *in each basic block m_j, only the instruction b_{f_j} can be branching; and*
3. *in each basic block m_j, only the instruction b_{i_j} can have more than one predecessor.*

A partition to basic blocks can be obtained as follows: the first sequence m_1 starts at pc_1 and ends at pc_{f_1}=$min(pc_{e_1}, pc_{s_1})$, where pc_{e_1} is the address of the first branching instruction after pc_1, and pc_{s_1} is the first address after pc_1 s.t. the instruction at address $next(pc_{s_1})$ has more than one predecessor. The sequence m_2 is computed similarly starting at pc_{i_2}=$next(pc_{f_1})$, etc. Note that this partition can be computed in two passes: the first computes the predecessors, and the second defines the beginning and end of each sub-sequence.

Example 1. The JBC *fact* method on the left of Fig. 1 is partitioned into four basic blocks. The initial addresses (pc_{i_x}) of these blocks are shown within boxes. Each block is labeled by $fact_{id}$ where id is a unique block identifier. A directed edge indicates a control flow from the last instruction in the source node to the first instruction in the destination node. Edges may be labeled by a guard which states conditions under which the edge may be traversed during execution. □

In *invokevirtual*, due to dynamic dispatching, the actual method to be called may not be known at compile time. To facilitate termination analysis, we capture this information and introduce it explicitly in the CFG. This is done by adding new blocks, the *implicit basic blocks*, containing calls to *actual methods* which might be called at runtime. Moreover, access to these blocks is guarded by mutually exclusive conditions on the runtime class of the calling object.

Definition 2 (implicit basic block). *Let m be a method which contains an instruction of the form pc:b, where b=*invokevirtual *m'. Let M be a superset of the methods (signatures) that might actually be called at runtime when executing pc:b. The implicit basic block for $m''{\in}M$ is $m_{pc:c}$, where $c{=}class(m'')$ if $class(m''){\preceq}class(m')$, otherwise $c{=}class(m')$. The block includes the single special instruction* invoke(m'). *The guard of $m_{pc:c}$ is $m^g_{pc:c}{=}$*instanceof(n, c, D), *where $D{=}\{class(m'') \mid m''{\in}M,\ class(m''){\prec}c\}$, and n is the arity of m'.*

It can be seen that m is used to denote both methods and blocks in order to make them globally unique. The above condition instanceof(n, c, D) states that the $(n{+}1)$th stack element (from the top) is an instance of class c and not an instance of any class in D. Computing the set M in the above definition can be statically done by considering the class hierarchy and the method signature, which is clearly a safe approximation of the set of the actual methods that might be called at runtime when executing b. However, in some cases this might result in a much larger set than the actual one, which in turn affects the precision and performance of the corresponding static analysis. In such cases, class analysis [25] is usually applied to reduce this set as it gives information about the possible runtime classes of the object whose method is being called. Note that the instruction invoke(m') does not appear in the original bytecode, but it is instrumental to define our rule-based representation in Sec. 2.2. For example, consider the CFG in Fig. 2, which corresponds to the recursive method *doSum* and calls *fact*. This CFG contains two implicit blocks labeled $doSum_{11:DoSum}$ and $doSum_{19:DoSum}$.

The following definition formalizes the notion of a CFG for a method. Although the invokespecial bytecode instruction always corresponds to only one possible method call which can be identified from the symbolic method reference, in order to simplify the presentation, we treat it as invokevirtual, and associate it to a single implicit basic block with the *true* guard. Note that every bytecode instruction belongs to exactly one basic block. By $BlockId(pc, m){=}i$ we denote the fact that the instruction pc in m belongs to block m_i. In addition, for a given *invokevirtual* instruction pc:b in a method m, we use M^m_{pc} and G^m_{pc} to denote the set of its *implicit basic blocks* and their corresponding guards respectively.

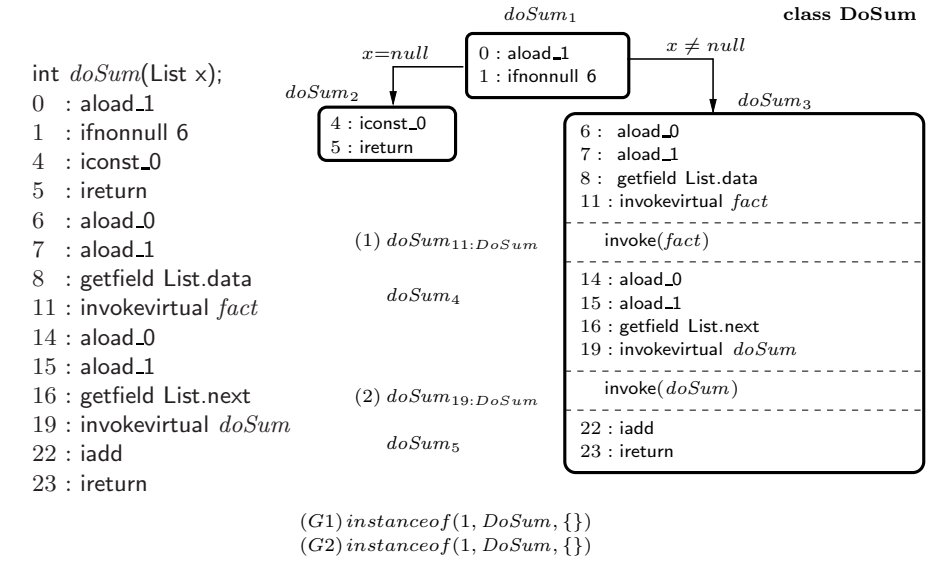


Fig. 2. The Control Flow Graph of the *doSum* example

Definition 3 (CFG). *The* control flow graph *for a method m is a graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$. Nodes $\mathcal{N}$ consist of:*

(a) basic blocks $m_1, \ldots, m_k$ of m; and
(b) implicit basic blocks corresponding to calls to methods.

Edges in $\mathcal{E}$ take the form $\langle m_i \rightarrow m_j, condition_{ij} \rangle$ where m_i and m_j are, resp., the source *and* destination *node, and $condition_{ij}$ is the Boolean condition labeling this transition. The set of edges is constructed, by considering each node $m_i \in \mathcal{N}$ which corresponds to a (non-implicit) basic block, whose last instruction is denoted as $pc{:}b$, as follows:*

1. *if b=*goto *pc' and j=$BlockId(pc', m)$ then we add $\langle m_i \rightarrow m_j, true \rangle$ to $\mathcal{E}$;*
2. *if b=*if_ϕ *pc', j=$BlockId(pc', m)$ and i'=$BlockId(next(pc), m)$ then we add both $\langle m_i \rightarrow m_j, \phi \rangle$ and $\langle m_i \rightarrow m_{i'}, \neg\phi \rangle$ to $\mathcal{E}$;*
3. *if $b \in$ {*invokevirtual *m',* invokespecial *m'}, and i'=$BlockId(next(pc), m)$ then, for all $d \in M^m_{pc}$ and its corresponding $g^m_{pc:d} \in G^m_{pc}$, we add $\langle m_i \rightarrow m_{pc:d}, g^m_{pc:d} \rangle$ and $\langle m_{pc:d} \rightarrow m_{i'}, true \rangle$ to $\mathcal{E}$;*
4. *otherwise, if j=$BlockId(next(pc), m)$ then we add $\langle m_i \rightarrow m_j, true \rangle$ to $\mathcal{E}$.*

For conciseness, when a branching instruction b involving implicit blocks leads to a single successor block, we include the corresponding invoke instruction within the basic block b belongs to. For instance, consider that the classes *DoSum* and *List* are not extended by any other class. In this case, the branching instructions 11 and 19 have a single continuation. Their associated implicit blocks marked with (1) and (2) in Fig. 2 are, thus, just included within the basic block $doSum_3$. $G1$ and $G2$ at the bottom indicate the guards which should label the edge.

2.2 Rule-Based Representation

The CFG, while having advantages, is not optimal for our purposes. Therefore, we introduce a *Rule-Based Representation* (RBR) on which we demonstrate our approach to termination analysis. This RBR is based on a recursive representation presented in previous work [1], where it has been used for cost analysis.

The main advantages of the RBR are that: (1) all iterative constructs (loops) fit in the same setting, independently of whether they originate from recursive calls or iterative loops (conditional and unconditional jumps); and (2) all variables in the local scope of the method a block corresponds to (formal parameters, local variables, and stack values) are represented uniformly as explicit arguments. This is possible as in JBC the height of the *operand stack* at each program point is statically known. We prefer to use this rule-based representation, rather than other existing ones (e.g., BoogiePL [13] or those in Soot [26]), as in a simple post-processing phase we can eliminate almost all stack variables, which results, as we will see in Sec. 3.1, in a more efficient analysis.

A *Rule-Based Program* (RBP for short) defines a set of *procedures*, each of them defined by one or more rules. As we will see later, each block in the CFG generates one or two procedures. Each rule has the form $head(\bar{x}, \bar{y}) := guard, instr, cont$ where *head* is the name of the procedure the rule belongs to, $\bar{x}$ and $\bar{y}$ indicate sequences $\langle x_1, \ldots, x_n \rangle$, $n>0$ (resp. $\langle y_1, \ldots, y_k \rangle$, $k>0$) of input (resp. output) arguments, *guard* is of the form guard(ϕ), where ϕ is a Boolean condition on the variables in $\bar{x}$, *instr* is a sequence of (decorated) bytecode instructions, and *cont* indicates a possible call to another procedure representing the continuation of this procedure. In principle, $\bar{x}$ includes the method's local variables and the stack elements at the beginning of the block. In most cases, $\bar{y}$ only needs to store the return value of the method, which we denote by r. For simplicity, guards of the form guard($true$) are omitted. When a procedure p is defined by means of several rules, the corresponding guards must cover all cases and be pairwise exclusive.

Decorating Bytecode Instructions. In order to make all arguments explicit, each bytecode instruction in *instr* is *decorated* explicitly with the (local and stack) variables it operates on. We denote by $t{=}stack\_height(pc, m)$ the height of the stack immediately before the program point pc in a method m. Function dec in the following table shows how to *decorate* some selected instructions, where n is the number of arguments of m.

pc:b	dec(b)
iconst i	$iconst(i, s_{t+1})$
istore v	$istore(s_t, \ell_v)$
iload v	$iload(l_v, s_{t+1})$
new c	$new(c, s_{t+1})$
ireturn	$ireturn(s_t, r)$

pc:b	dec(b)
iadd	$iadd(s_{t-1}, s_t, s_{t-1})$
invoke(m)	$m(\langle s_{t-n}, \ldots, s_t\rangle, \langle s_{t-n}\rangle)$
getfield f	$getfield(f, s_t, s_t)$
putfield f	$putfield(f, s_{t-1}, s_t, s_{t-1})$
guard($icmpgt$)	guard$(icmpgt(s_{t-1}, s_t))$

Guards are translated according to the bytecode instruction they come from. Note that branching instructions do not need to appear in the RBR, since their effect is already captured by the branching at the RBR level and since invoke instructions are replaced by calls to the entry rule of the corresponding method.

Definition 4 (RBR). *Let m be a method with $l_0, \ldots, l_{n-1}$ local variables, of which $l_0, \ldots, l_{k-1}$ are the formal parameters together with the* this *reference l_0 ($k \leq n$), and let $\langle \mathcal{N}, \mathcal{E}\rangle$ be its CFG. The* rule-based representation *of $\langle \mathcal{N}, \mathcal{E}\rangle$ is* $\mathsf{rules}(\langle \mathcal{N}, \mathcal{E}\rangle) = \mathsf{entry}(\langle \mathcal{N}, \mathcal{E}\rangle) \bigcup_{m_p \in \mathcal{N}} \mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle)$, *with:*

$$\mathsf{entry}(\langle \mathcal{N}, \mathcal{E}\rangle) = \{m(\langle \ell_0, \ldots, \ell_{k-1}\rangle, \langle r\rangle) := \mathsf{init\_local\_vars}(\langle l_k, \ldots, l_{n-1}\rangle), m_1(\langle \ell_0, \ldots, \ell_{n-1}\rangle, \langle r\rangle)\}$$

where the call init_local_vars *initializes the local variables of the method, and*

$$\mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle) = \begin{cases} \{m_p(\langle \bar{l}, s_0, \ldots, s_{p_i-1}\rangle, \langle r\rangle) := TBC^p_m.\} & \nexists \langle m_p \mapsto \_, \_\rangle \in \mathcal{E} \\ \{m_p(\langle \bar{l}, s_0, \ldots, s_{p_i-1}\rangle, \langle r\rangle) := TBC^p_m, m^c_p(\langle \bar{l}, s_0, \ldots, s_{p_o-1}\rangle, \langle r\rangle).\} \bigcup & \\ \quad \{m^c_p(\langle \bar{l}, s_0, \ldots, s_{p_o-1}\rangle, \langle r\rangle) := g, m_q(\langle \bar{l}, s_0, \ldots, s_{q_i-1}\rangle, \langle r\rangle). & \textit{otherwise} \\ \quad \mid \langle m_p \to m_q, \phi_q\rangle \in \mathcal{E} \wedge g = \mathsf{dec}(\phi_q)\} & \end{cases}$$

In the above formula, p_i (resp., p_o) denotes the height of the operand stack of m at the entry (resp., exit) of m_p. Also, q_i is the height of the stack at the entry of m_q, and TBC^p_m is the decorated bytecode for m_p. We use "_" to indicate that the value at the corresponding position is not relevant.

The function $\mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle)$ is defined by cases. The first case is applied when m_p is a *sink* node with no out-edges. Otherwise, the second rule introduces an additional procedure m^c_p (c is for *continuation*), which is defined by as many rules as there are out-edges for m_p. These rules capture the different alternatives which execution can follow from m_p. We will unfold calls to m^c_p whenever it is deterministic (m_p has a single out-edge). This results in m_p calling m_q directly.

Example 2. The RBR of the CFG in Fig. 1 consists of the following rules where local variables have the same name as in the source code and o is the *this* object:

$$
\begin{array}{ll}
fact(\langle o, n\rangle, \langle r\rangle) & := \mathsf{init\_local\_vars}(\langle ft, i\rangle),\ fact_1(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_1(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iconst}(1, s_0),\ \mathsf{istore}(s_0, ft),\ \mathsf{iconst}(1, s_0),\\
 & \quad \mathsf{istore}(s_0, i),\ fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_2(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(i, s_0),\ \mathsf{iload}(n, s_1),\\
 & \quad fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle).\\
fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{icmpgt}(s_0, s_1)),\ fact_4(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{icmple}(s_0, s_1)),\ fact_3(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_3(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(ft, s_0),\ \mathsf{iload}(i, s_1),\ \mathsf{imul}(s_0, s_1, s_0),\\
 & \quad \mathsf{istore}(s_0, ft),\ \mathsf{iinc}(i, 1),\ fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_4(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(ft, s_0),\ \mathsf{ireturn}(s_0, r).
\end{array}
$$

The first rule corresponds to the entry. Block $fact_4$ is a sink block. Blocks $fact_1$ and $fact_3$ have a single out-edge and we have unfolded the continuation. Finally, block $fact_2$ has two out-edges and needs the procedure $fact_2^c$. The RBR from the CFG of *doSum* in Fig. 2 is ($doSum_3$ merges several blocks with one out-edge):

$$
\begin{array}{ll}
doSum(\langle o, x\rangle, \langle r\rangle) & := \mathsf{init\_local\_vars}(\langle\rangle),\ doSum_1(\langle o, x\rangle, \langle r\rangle).\\
doSum_1(\langle o, x\rangle, \langle r\rangle) & := \mathsf{aload}(x, s_0),\ doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle).\\
doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{nonnull}(s_0)),\ doSum_3(\langle o, x\rangle, \langle r\rangle).\\
doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{null}(s_0)),\ doSum_2(\langle o, x\rangle, \langle r\rangle).\\
doSum_2(\langle o, x\rangle, \langle r\rangle) & := \mathsf{iconst}(0, s_0), \mathsf{ireturn}(s_0, r).\\
doSum_3(\langle o, x\rangle, \langle r\rangle) & := \mathsf{aload}(o, s_0), \mathsf{aload}(x, s_1), \mathsf{getfield}(List.data, s_1, s_1),\\
 & \quad fact(\langle s_0, s_1\rangle, \langle s_0\rangle), \mathsf{aload}(o, s_1), \mathsf{aload}(x, s_2),\\
 & \quad \mathsf{getfield}(List.next, s_2, s_2), doSum(\langle s_1, s_2\rangle, \langle s_1\rangle),\\
 & \quad \mathsf{iadd}(s_0, s_1, s_0), \mathsf{ireturn}(s_0, r).
\end{array}
$$

We can see that a call to a different method, *fact*, occurs in $doSum_3$. This shows that our RBR allows simultaneously handling the two CFGs in our example. □

Rule-based Programs vs JBC Programs. Given a JBC program P, P_r denotes the RBP obtained from P. Note that, it is trivial to define an *interpreter* (or *abstract machine*) which can execute any P_r and obtain the same return value and termination behaviour as a JVM does for P. RBPs, in spite of their declarative appearance, are in fact imperative programs. As in the JVM, an interpreter for RBPs needs, in addition to a stack for activation records, a global heap. These activation records differ from those in the JVM in that the operand stack is no longer needed (as stack elements are explicit) and in that the scope of variables is no longer associated to methods but rather to rules. In RBPs all rules are treated uniformly, regardless of the method they originate from, so that method borders are somewhat blurred. As in the JVM, call-by-value is used for passing arguments in calls.

3 Proving Termination

This section describes how to prove termination of a JBC program given its RBR. The approach consists of two steps. In the first, we *abstract* the RBR rules by replacing all program data by their corresponding *size*, and replacing calls

corresponding to bytecode instructions by *size constraints* on the values their variables can take. This step results in a Constraint Logic Program (CLP) [16] over integers, where, for any bytecode trace t, there exists a CLP trace t' whose states are abstractions of t states. In particular, every infinite (non terminating) bytecode trace has a corresponding infinite CLP trace, so that termination of the CLP program implies termination of the bytecode program. Note that, unlike in bytecode traces which are always deterministic, the execution of a CLP program can be non-deterministic, due to the precision loss inherent to the abstraction.

In the second step, we apply techniques for proving termination of CLP programs [9], which consist of: (1) analyzing the rules for each method to infer input-output *size relations* between the method input and output variables; (2) using the input-output size relations for the methods in the program, we infer a set of abstract *direct calls-to pairs* which describe, in terms of size-change, all possible calls from one procedure to another; and (3) given this set of abstract direct calls-to pairs, we compute a set of all possible calls-to pairs (direct and indirect), describing all transitions from one procedure to another. Then we focus on the pairs which describe loops, and try to identify *ranking functions* which guarantee the termination of each loop and thus of the original program.

3.1 Abstracting the Rules

As mentioned above, rule abstraction replaces data by the *size* of data, and focuses on relations between data size. For integers, their size is just their integer value [12]. For references, we take their size to be their *path-length* [24], i.e., the length of the maximal path reachable from the reference. Then, bytecode instructions are replaced by constraints on sizes taking into account a Static Single Assignment (SSA) transformation. SSA is needed because variables in CLP programs cannot be assigned more than one value. For example, an instruction $\mathsf{iadd}(s_0, s_1, s_0)$ will be abstracted to $s'_0{=}s_1{+}s_0$ where s'_0 refers to the value of s_0 after executing the instruction. Also, the bytecode $\mathsf{getfield}(f, s_0, s_0)$ is abstracted to $s_0{>}s'_0$ if it can be determined that s_0 (before executing the instruction) does not reference a cyclic data-structure, since the length of the longest-path reachable from s_0 is larger than the length of the longest path reachable from s'_0.

bytecode b	*abstract bytecode* b^α	ρ_{i+1}
$\mathsf{iload}(l_v, s_j)$	$s'_j{=}\rho_i(l_v)$	$\rho_i[s_j{\mapsto}s'_j]$
$\mathsf{iadd}(s_j, s_{j+1}, s_j)$	$s'_j{=}\rho_i(s_j){+}\rho_i(s_{j+1})$	$\rho_i[s_j{\mapsto}s'_j]$
$\mathsf{guard}(icmpgt(s_j, s_{j+1}))$	$\rho_i(s_j){>}\rho_i(s_{j+1})$	ρ_i
$\mathsf{getfield}(f, s_j, s_j)$	if f is of ref. type: $\rho_i(s_j){>}s'_j$ if s_j is not cyclic otherwise $\rho_i(s_j) \geq s'_j$. If f is not of ref. type: *true*	$\rho_i[s_j \mapsto s'_j]$
$\mathsf{putfield}(f, s_j, s_{j+1})$ s.t f is of ref. type	if s_j and s_{j+1} do not share, $s'_k{\leq}\rho_i(s_k){+}\rho_i(s_{j+1})$ for any s_k that shares with s_j, otherwise *true*.	$\rho_i[s_k \mapsto s'_k]$

To implement the SSA transformation we maintain a mapping ρ of variable names (as they appear in the rule) to new variable names (constraint variables). Such a mapping is referred to as a *renaming*. We let $\rho[x \mapsto y]$ denote the

modification of the renaming ρ such that it maps x to the new variable y. We denote by $\rho[\bar{x} \mapsto \bar{y}]$ the mapping of each element in $\bar{x}$ to a corresponding one in $\bar{y}$.

Definition 5 (abstract compilation). *Let* $\mathrm{R} \equiv p(\overline{x}, \overline{y}) := b_1, \ldots, b_n$ *be a rule. Let* ρ_i *be a renaming associated with the point before each* b_i *and let* ρ_1 *be the identity renaming (on the variables in the rule). The abstraction of* R *is denoted* R^α *and takes the form* $p(\overline{x}, \overline{y}') := b_1^\alpha, \ldots, b_n^\alpha$ *where* b_i^α *are computed iteratively from left to right as follows:*

1. *if* b_i *is a bytecode instruction or a guard, then* b_i^α *and* ρ_{i+1} *are obtained from a predefined lookup table similar to the one above.*
2. *if* b_i *is a call to a procedure* $q(\overline{w}, \overline{z})$*, then the abstraction* b_i^α *is* $q(\overline{w}', \overline{z}')$ *where each* $w'_k \in \overline{w}'$ *is* $\rho_i(w_k)$*, variables* $\overline{z}'$ *are fresh, and* $\rho_{i+1}{=}\rho_i[\overline{z} \mapsto \overline{z}', \overline{u} \mapsto \overline{u}']$ *where* $\overline{u}'$ *are also fresh variables and* $\overline{u}$ *is the set of all variables in* $\overline{w}$ *which reference data-structures that can be modified when executing* q *and those that share (i.e., might have common regions in the heap) with them.*
3. *at the end we define each* $y'_i \in \overline{y}'$ *to be the constrained variable* $\rho_{n+1}(y_i)$*.*

In addition, all reference *variables are (implicitly) assumed to be non-negative.*

Note that in point 2 above, the set of variables such that the data-structures they point to may be modified during the execution of q can be approximated by applying constancy analysis [14], which aims at detecting the method arguments that remain constant during execution, and sharing analysis [23] which aims at detecting reference variables that might have common regions on the heap. Also, the non-cyclicity condition required for the abstraction of getfield can be verified by non-cyclicity analysis [22]. In what follows, for simplicity, we assume that abstract rules are normalized to the form $p(\overline{x}, \overline{y}) := \varphi, p_1(\overline{x}_1, \overline{y}_1), \ldots, p_j(\overline{x}_j, \overline{y}_j)$ where φ is the conjunction of the (linear) size constraints introduced in the abstraction and each $p_i(\overline{x}_i, \overline{y}_i)$ is a call to a procedure (i.e., block or method).

Example 3. Recall the following rule from Ex. 2 (on the left) and its abstraction (on the right) where the renamings are indicated as comments.

$fact_3(\langle o, n, ft, i\rangle, \langle r\rangle) :=$	$fact_3(\langle o, n, ft, i\rangle, \langle r'\rangle) :=$	$\%\ \rho_1 = id$
$\quad \mathsf{iload}(ft, s_0),$	$\quad s'_0 = ft,$	$\%\ \rho_2 = \rho_1[s_0 \mapsto s'_0]$
$\quad \mathsf{iload}(i, s_1),$	$\quad s'_1 = i,$	$\%\ \rho_3 = \rho_2[s_1 \mapsto s'_1]$
$\quad \mathsf{imul}(s_0, s_1, s_0),$	$\quad true,$	$\%\ \rho_4 = \rho_3[s_0 \mapsto s''_0]$
$\quad \mathsf{istore}(s_0, ft),$	$\quad ft' = s''_0,$	$\%\ \rho_5 = \rho_4[ft \mapsto ft']$
$\quad \mathsf{iinc}(i, 1),$	$\quad i' = i + 1,$	$\%\ \rho_6 = \rho_5[i \mapsto i']$
$\quad fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).$	$\quad fact_2(\langle o, n, ft', i'\rangle, \langle r'\rangle).$	$\%\ \rho_7 = \rho_6[r \mapsto r']$

Note that imul is abstracted to *true*, since it imposes a non-linear constraint. □

3.2 Input Output Size-Relations

We consider the abstract rules obtained in the previous step to infer an abstraction (w.r.t. size) of the input-output relation of the program blocks. Concretely,

we infer *input-output size relations* of the form $p(\overline{x},\overline{y}) \leftarrow \varphi$, where φ is a constraint describing the relation between the sizes of the input $\overline{x}$ and the output $\overline{y}$ upon exit from p. This information is needed since output of one call may be input to another call. E.g., consider the following contrived abstract rule $p(\langle x\rangle,\langle r\rangle)$:=$\{x{>}0,x{>}z\}, q(\langle z\rangle,\langle y\rangle), p(\langle y\rangle,\langle r\rangle)$. To prove termination, it is crucial to know the relation between x in the head and y in the recursive call to p. This requires knowledge about the input-output size relations for $q(\langle z\rangle,\langle y\rangle)$. Assuming this to be $q(\langle z\rangle,\langle y\rangle) \leftarrow z{>}y$, we can infer $x{>}y$. Since abstract programs are CLP programs, inferring relations can rely on standard techniques [4].

Computing an approximation of input-output size relation requires a global fixpoint. In practice, we can often take a trivial over-approximation where for all rules there is no information, namely, $p(\overline{x},\overline{y}) \leftarrow \mathit{true}$. This can prove termination of many programs, and results in a more efficient implementation. It is not enough in cases as the above abstract rule, which however, in our experience, often does not occur in imperative programs.

3.3 Call-to Pairs

Consider again the abstract rule from Ex. 3 which (ignoring the output variable) is of the form $\mathit{fact}_3(\bar{x}) := \varphi, \mathit{fact}_2(\bar{z})$. It means that whenever execution reaches a call to $\mathit{fact}_3(\bar{x})$ there will be a subsequent call to $\mathit{fact}_2(\bar{z})$ and the constraint φ holds. In general, subsequent calls may arise also from rules which are not binary. Given an abstract rule of the form $p_0 := \varphi, p_1, \ldots, p_n$, a call to p_0 may lead to a call to p_i, $1{\le}i{\le}n$. Given the input-output size relations for the individual calls $p_1,\ldots,p_{i-1}$, we can characterize the constraint for a transition between the subsequent calls p_0 and p_i by adding these relations to φ. We denote a pair of such subsequent calls by $\langle p_0(\overline{x}) \rightsquigarrow p_i(\overline{y}), \varphi_i\rangle$ and call it a *calls-to pair*.

Definition 6 (direct calls-to pairs). *Given a set of abstract rules $\mathcal{A}$ and its input-output size relations $I_{\mathcal{A}}$, the* direct calls-to pairs *induced by $\mathcal{A}$ and $I_{\mathcal{A}}$ are:*

$$C_{\mathcal{A}} = \left\{ \langle p(\overline{x}) \rightsquigarrow p_i(\overline{x}_i), \psi\rangle \;\middle|\; \begin{array}{l} p(\overline{x},\overline{y}) := \varphi, p_1(\overline{x}_1,\overline{y}_1),\ldots,p_j(\overline{x}_j,\overline{y}_j){\in}\mathcal{A}, \\ i{\in}\{1,\ldots,j\},\ \forall 0{<}k{<}i.\ p_k(\overline{x}_k,\overline{y}_k) \leftarrow \varphi_k{\in}I_{\mathcal{A}} \\ \psi = \bar{\exists}\overline{x} \cup \overline{x}_i.\varphi \wedge \varphi_1 \wedge \ldots \wedge \varphi_{i-1} \end{array} \right\}$$

where $\bar{\exists}v$ means eliminating all variables but v from the corresponding constraint.

Example 4. Consider the rule for *doSum* in Ex. 2: note that input-output relations for *fact* and *doSum* are *true*. Direct calls-to pairs for those rules are:

$$\begin{array}{l} \langle \mathit{doSum}(o,x) \rightsquigarrow \mathit{doSum}_1(o',x'), \{x'{=}x, o'{=}o\}\rangle \\ \langle \mathit{doSum}_1(o,x) \rightsquigarrow \mathit{doSum}_1^c(o',x',s_0), \{x'{=}x, o'{=}o, s_0{=}x\}\rangle \\ \langle \mathit{doSum}_1^c(o,x,s_0) \rightsquigarrow \mathit{doSum}_3(o',x'), \{x'{=}x, o'{=}o, s_0{>}0\}\rangle \\ \langle \mathit{doSum}_1^c(o,x.s_0) \rightsquigarrow \mathit{doSum}_2(o',x'), \{x'{=}x, o'{=}o, s_0{=}0\}\rangle \\ \langle \mathit{doSum}_3(o,x) \rightsquigarrow \mathit{fact}(s_0',s_1''), \{s_0'{=}o\}\rangle \\ \langle \mathit{doSum}_3(o,x) \rightsquigarrow \mathit{doSum}(s_1''',s_2''), \{s_1'''{=}o, x{>}s_2''\}\rangle \end{array}$$

In the last rule, s_2'' corresponds to $x.\mathit{next}$, so that we have the constraint $x{>}s_2''$. It can be seen that since the list is not cyclic and does not share with other

variables, size analysis finds the above decreasing of its size $x{>}s_2''$. Note also that all variables corresponding to references are assumed to be non-negative. Similarly, we can obtain direct calls-to pairs for the rule of *fact*. □

It should be clear that the set of direct calls-to pairs relations $C_{\mathcal{A}}$ is also a binary CLP program that we can execute from a given goal. A key feature of this binary program is that if an infinite trace can be generated using the abstract program described in Sec. 3.1, then an infinite trace can be generated using this binary CLP program [10]. Therefore, absence of such infinite traces (i.e., termination) in the binary program $C_{\mathcal{A}}$ implies absence of infinite traces in the abstract bytecode program, as well as in the original bytecode program.

Theorem 1 (Soundness). *Let P be a JBC program and $C_{\mathcal{A}}$ the set of direct calls-to pairs computed from P. If there exists a non-terminating trace in P then there exists a non-terminating derivation in $C_{\mathcal{A}}$.*

Intuitively, the result follows from the following points. By construction, the RBP captures all possibly non-terminating traces in the original program. By the correctness of size analysis, we have that, given a trace in the RBP, there exists an equivalent one in $C_{\mathcal{A}}$, among possibly others. Therefore, termination in $C_{\mathcal{A}}$ entails termination in the JBC program.

3.4 Proving Termination of the Binary Program $C_{\mathcal{A}}$

Several automatic termination tools and methods for proving termination of such binary constraint programs exists [9,10,17]. They are based on the idea of first computing all possible calls-to pair from the direct ones, and then finding a ranking function for each recursive calls-to pairs, which is sufficient for proving termination. Computing all possible calls-to pairs, usually called the *transitive closure* $C_{\mathcal{A}}^*$, can be done by starting from the set of direct calls-to pairs $C_{\mathcal{A}}$, and iteratively adding to it all possible compositions of its elements until a fixed-point is reached. Composing two calls-to pairs $\langle p(\overline{x}) \rightsquigarrow q(\overline{y}), \varphi_1\rangle$ and $\langle q(\overline{w}) \rightsquigarrow r(\overline{z}), \varphi_2\rangle$ returns the new calls-to pair $\langle p(\overline{x}) \rightsquigarrow r(\overline{z}), \bar{\exists}\bar{x} \cup \bar{z}.\varphi_1 \wedge \varphi_2 \wedge (\bar{y} = \bar{w})\rangle$.

Example 5. Applying the transitive closure on the direct calls-to pairs of Ex. 4, we obtain, among many others, the following calls-to pairs. Note that x (resp. i) strictly decreases (resp. increases) at each iteration of its corresponding loop:

$$\langle doSum(o,x) \rightsquigarrow doSum(o',x'), \{o'{=}o, x{>}x', x{\geq}0\}\rangle$$
$$\langle fact_2(o,n,ft,i) \rightsquigarrow fact_2(o',n',ft',i'), \{o'{=}o, n'{=}n, i'{>}i, i{\geq}1, n{\geq}i'{-}1\}\rangle \quad \square$$

As already mentioned, in order to prove termination, we focus on *loops* in $C_{\mathcal{A}}^*$. Loops are the *recursive* entities of the form $\langle p(\overline{x}) \rightsquigarrow p(\overline{y}), \varphi\rangle$ which indicate that a call to a program point p with values $\bar{x}$ eventually leads to a call to the same program point with values $\bar{y}$ and that φ holds between $\overline{x}$ and $\overline{y}$. For each loop, we seek a *ranking function* F over a well-founded domain such that $\varphi \models \mathrm{F}(\overline{x}){>}\mathrm{F}(\overline{y})$. As shown in [9,10], finding a ranking function for every recursive calls-to pair implies termination. Computing such functions can be done, for instance, as described in [21]. As an example, for the loops in Ex. 5 we get the following ranking functions: $\mathrm{F}_1(o,x){=}x$ and $\mathrm{F}_2(o,n,ft,i){=}n{-}i{+}1$.

3.5 Improving Termination Analysis by Extracting Nested Loops

In proving termination of JBC programs, one important question is whether we can prove termination at the JBC level for a class of programs which is comparable to the class of Java *source* programs for which termination can be proved using similar technology. As can be seen in Sec. 4, directly obtaining the RBR of a bytecode program is non-optimal, in the sense that proving termination on it may be more complicated than on the source program. This happens because, while in source code it is easy to reason about a nested loop independently of the outer loop, loops are not directly visible when control flow is unstructured. Loop extraction is useful for our purposes since nested loops can be dealt with one at a time. As a result, finding a ranking function is easier, and computing the closure can be done locally in the strongly connected components. This can be crucial in proving the termination of programs with nested loops.

To improve the accuracy of our analysis, we include a component which can detect and extract loops from CFGs. Due to space limitations, we do not describe how to perform this step here (more details in work about decompilation [2], where loop extraction has received considerable attention). Very briefly, when a loop is extracted, a new CFG is created. As a result, a method can be converted into several CFGs. These ideas fit very nicely within our RBR, since calls to loops are handled much in the same way as calls to other methods.

4 Experimental Results

Our prototype implementation is based on the size analysis component of [1] and extends it with the additional components needed to prove termination. The analyzer can also output the set of direct call-pairs, which allows using existing termination analyzers based on similar ideas [10,17]. The system is implemented in Ciao Prolog, and uses the Parma Polyhedra Library (PPL) [3].

Table 1 shows the execution times of the different steps involved in proving the termination of JBC programs, computed as the arithmetic mean of five runs. Experiments have been performed on an Intel 1.86 GHz Pentium M with 1 GB of RAM, running Linux on a 2.6.17 kernel. The table shows a range of benchmarks for which our system can prove termination, and which are meant to illustrate different features. We show classical recursive programs such as *Hanoi*, *Fibonacci*, *MergeList* and *Power*. Iterative programs *DivByTwo* and *Concat* contain a single loop, while *Sum*, *MatMult* and *BubbleSort* are implemented with nested loops. We also include programs written in object-oriented style, like *Polynomial*, *Incr*, *Scoreboard*, and *Delete*. The remaining benchmarks use data structures: arrays (*ArrayReverse*, *MatMultVector*, and *Search*); linked lists (*Delete* and *ListReverse*); and binary trees (*BST*).

Columns **CFG**, **RBR**, **Size**, **TC**, **RF**, **Total**$_1$ contain the running times (in ms) required for the CFG (including loop extraction), the RBR, the size analysis (including input-output relations), the transitive closure, the ranking functions and the total time, respectively. Times are high, as the implementation has been developed to check if our approach is feasible, but is still preliminary. The most

Table 1. Measured time (in ms) of the different phases of proving termination

Benchmark	**CFG**	**RBR**	**Size**	**TC**	**RF**	**Total$_1$**	**Termin**	**Total$_2$**	**Ratio**
Polynomial	138	12	260	1453	26	1890	yes	2111	1.12
DivByTwo	52	4	168	234	4	462	yes	538	1.17
EvenDigits	59	7	383	1565	17	2030	yes	2210	1.09
Factorial	43	3	46	268	3	363	yes	353	0.97
ArrayReverse	58	5	208	339	24	635	yes	834	1.32
Concat	65	8	660	943	38	1715	yes	3815	2.23
Incr	35	12	854	4723	28	5652	yes	6590	1.17
ListReverse	21	5	141	310	5	481	yes	515	1.07
MergeList	107	23	130	5184	21	5464	yes	5505	1.01
Power	14	3	72	357	9	454	yes	459	1.01
Cons	25	7	65	1318	10	1424	yes	1494	1.05
ListInter	136	22	585	9769	49	10560	yes	27968	2.65
SelectOrd	154	16	1298	4076	48	5592	no	25721	4.60
DoSum	57	10	64	923	6	1060	yes	1069	1.01
Delete	121	14	54	2418	1	2608	yes	33662	12.91
MatMult	240	11	2411	4646	294	7602	no	32212	4.24
MatMultVector	254	15	2563	8744	242	11817	no	34688	2.94
Hanoi	39	5	172	979	3	1198	no	1198	1.00
Fibonacci	23	3	90	290	5	411	yes	401	0.98
BST	68	12	97	4643	18	4838	yes	4901	1.01
BubbleSort	152	12	1125	4366	83	5738	no	14526	2.53
Search	65	11	307	756	11	1150	yes	1430	1.24
Sum	64	7	480	1758	35	2343	no	5610	2.39
FactSumList	65	12	80	961	5	1123	yes	1306	1.16
Scoreboard	268	23	1597	4393	81	6362	no	32999	5.19

expensive steps are the size analysis and the transitive closure, since they require global analysis. Last three columns show the benefits of loop extraction. **Termin** tells if termination can be proven (using polyhedra) without extraction. In seven cases, termination is only proven if loop extraction is performed. **Total$_2$** shows the total time required to check termination without loop extraction. **Ratio** compares **Total$_2$** with **Total$_1$** (**Total$_2$**/**Total$_1$**), showing that, in addition to improving precision, loop extraction is beneficial for efficiency, since **Ratio** ≥ 1 in most cases, and can be as high as 12.91 in *Delete*. Note that termination of these programs may be proved without loop extraction by using other domains such as *monotonicity constraints* [7]. However, we argue that loop extraction is beneficial as it facilitates reasoning on the loops separately. Also, if it fails to prove termination, it reports the possibly non-terminating loops.

5 Conclusions and Related Work

We have presented a termination analysis for (sequential) JBC which is, to the best of our knowledge, the first approach in this direction. This analysis

successfully deals with the challenges related to the low-level nature of JBC, and adapts standard techniques used, in other settings, in decompilation and termination analysis. Also, we believe that many of the ideas presented in this paper are also applicable to termination analysis of low-level languages in general, and not only JBC. We have used the notion of path-length to measure the size of data structures on the heap. However, our approach is parametric on the abstract domain used to measure the size. As future work, we plan to implement non-cyclicity analysis [22], constancy analysis [14], and sharing analysis [23], and to enrich the transitive closure components with monotonicity constraints [7]. Unlike polyhedra, monotonicity constraints can handle disjunctive information which is often crucial for proving termination. In [5], a termination analysis for C programs, based on binary relations similar to ours, is proposed. It uses separation logic to approximate the heap structure, which in turn allows handling termination of programs manipulating cyclic data structures. We believe that, for programs whose termination does not depend on cyclic data-structures, both approaches deal with the same class of programs. However, ours might be more efficient, as it is based on a simpler abstract domains (a detailed comparison is planned for future work). Recently, a novel termination approach has been suggested [8]. It is based on cyclic proofs and separation logic, and can even handle complicated examples as the reversal of panhandle data-structures. It is not clear to us how practical this approach is.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. Samir Genaim was supported by a *Juan de la Cierva* Fellowship awarded by the Spanish Ministry of Science and Education. Part of this work was performed during a research stay of Michael Codish at UPM supported by a grant from the Secretaría de Estado de Educación y Universidades, Spanish Ministry of Science and Education. The authors would like to thank the anonymous referees for their useful comments.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
2. Allen, F.: Control flow analysis. In: Symp. on Compiler optimization (1970)
3. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
4. Benoy, F., King, A.: Inferring Argument Size Relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)

5. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
6. Bradley, A., Manna, Z., Sipma, H.: Termination of polynomial programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
7. Brodsky, A., Sagiv, Y.: Inference of Inequality Constraints in Logic Programs. In: Proceedings of PODS 1991, pp. 95–112. ACM Press, New York (1991)
8. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Proceedings of POPL-35 (January 2008)
9. Bruynooghe, M., Codish, M., Gallagher, J., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms. ACM TOPLAS 29(2) (2007)
10. Codish, M., Taboch, C.: A semantic basis for the termination analysis of logic programs. J. Log. Program. 41(1), 103–123 (1999)
11. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. POPL. ACM Press, New York (1978)
13. DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft (2005)
14. Genaim, S., Spoto, F.: Technical report, Personal Communication (2007)
15. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130. Springer, Heidelberg (2006)
16. Jaffar, J., Maher, M.: Constraint Logic Programming: A Survey. Journal of Logic Programming 19(20), 503–581 (1994)
17. Lee, C., Jones, N., Ben-Amram, A.: The size-change principle for program termination. In: Proc. POPL. ACM Press, New York (2001)
18. Lindenstrauss, N., Sagiv, Y.: Automatic termination analysis of logic programs. In: ICLP (1997)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. A-W (1996)
20. Necula, G.: Proof-Carrying Code. In: POPL 1997. ACM Press, New York (1997)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937. Springer, Heidelberg (2004)
22. Rossignoli, S., Spoto, F.: Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855. Springer, Heidelberg (2005)
23. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)
24. Spoto, F., Hill, P.M., Payet, E.: Path-length analysis for object-oriented programs. In: Proc. EAAI (2006)
25. Spoto, F., Jensen, T.: Class analyses as abstract interpretations of trace semantics. ACM Trans. Program. Lang. Syst. 25(5), 578–630 (2003)
26. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proc. of CASCON 1999, pp. 125–135 (1999)

Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis

Elvira Albert[1], Puri Arenas[1], Samir Genaim[2], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. The classical approach to automatic cost analysis consists of two phases. Given a program and some measure of cost, we first produce *recurrence relations* (RRs) which capture the cost of our program in terms of the size of its input data. Second, we convert such RRs into *closed form* (i.e., without recurrences). Whereas the first phase has received considerable attention, with a number of cost analyses available for a variety of programming languages, the second phase has received comparatively little attention. In this paper we first study the features of RRs generated by automatic cost analysis and discuss why existing computer algebra systems are not appropriate for automatically obtaining closed form solutions nor upper bounds of them. Then we present, to our knowledge, the first practical framework for the fully automatic generation of reasonably accurate upper bounds of RRs originating from cost analysis of a wide range of programs. It is based on the inference of *ranking functions* and *loop invariants* and on *partial evaluation*.

1 Introduction

The aim of *cost analysis* is to obtain *static* information about the execution cost of programs w.r.t. some cost *measure*. Cost analysis has a large application field, which includes resource certification [11,4,16,9], whereby code consumers can reject code which is not guaranteed to run within the resources available. The resources considered include processor cycles, memory usage, or *billable events*, e.g., the number of text messages or bytes sent on a mobile network.

A well-known approach to automatic cost analysis, which dates back to the seminal work of [25], consists of two phases. In the first phase, given a program and some cost measure, we produce a set of equations which captures the cost of our program in terms of the size of its input data. Such equations are generated by converting the iteration constructs of the program (loops and recursion) into recurrences and by inferring *size relations* which approximate how the size of arguments varies. This set of equations can be regarded as *recurrence relations* (RRs for short). Equivalently, it can be regarded as *time bound programs* [22]. The aim of the second phase is to obtain a non-recursive representation of the equations, known as *closed form*. In most cases, it is not possible to find an exact solution and the closed form corresponds to an upper bound.

M. Alpuente and G. Vidal (Eds.): SAS 2008, LNCS 5079, pp. 221–237, 2008.

There are a number of cost analyses available which are based on this approach and which can handle a range of programming languages, including functional [7,18,22,23,24,25], logic [12,20], and imperative [1,3]. While in all such analyses the first phase is studied in detail, the second phase has received comparatively less attention. Basically, there are three different approaches for the second phase. One approach, which is conceptually linked viewing equations as time bound programs, was proposed in [18] and advocated in [22]. It is based on existing source-to-source transformations which convert recursive programs into non-recursive ones. The second approach consists in building restricted recurrence solvers using standard mathematical techniques, as in [12,25]. The third approach consists in relying on existing *computer algebra systems* (CASs for short) such as Mathematica®, MAXIMA, MAPLE, etc., as in [3,7,23,24].

The problem with the three approaches above is that they assume a rather limited form of equations which does not cover the essential features of equations actually generated by automatic cost analysis. In the rest of the paper, we will concentrate on viewing equations as recurrence relations and will use the term *Cost Relation* (CR for short) to refer to the relations produced by automatic cost analysis. In our own experience with [3], we have detected that existing CASs are, in most cases, not capable of handling CRs. We argue that automatically converting CRs into the format accepted by CASs is unfeasible. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes. An altogether different approach to cost analysis is based on type systems with resource annotations which does not use equations. Thus, it does not need to obtain closed forms, but it is typically restricted to linear bounds [16]. The need for improved mechanisms for obtaining upper bounds was already pointed out in Hickey and Cohen [14]. A relevant work in this direction is PURRS [5], which has been the first system to provide, in a fully automatic way, non-asymptotic upper and lower bounds for a wide class of recurrences. Unfortunately, and unlike our proposal, it also requires CRs to be deterministic. Marion et. al. [19,8] use a kind of polynomial ranking functions, but the approach is limited to polynomial bounds and can only handle a rather restricted form of CRs.

We believe that the lack of automatic tools for the above second phase is a major reason for the diminished use of automatic cost analysis. In this paper we study the features of CRs and discuss why existing CASs are not appropriate for automatically bounding them. Furthermore, we present, to our knowledge, the first practical framework for the fully automatic inference of reasonably accurate upper bounds for CRs originating from a wide range of programs. To do this, we apply semantic-based transformation and analysis techniques, including inference of *ranking functions*, *loop invariants* and the use of *partial evaluation*.

1.1 Motivating Example

Example 1. Consider the Java code in Fig. 1. It uses a class List for (non sorted) linked lists of integers. Method del receives an input list without repetitions l,

```
void del(List l, int p, int a[], int la, int b[], int lb){
  while (l!=null) {
    if (l.data<p) {
      la=rm_vec(l.data, a, la);
    } else {
      lb=rm_vec(l.data, b, lb);
    }
    l=l.next;
  }
}

int rm_vec(int e, int a[], int la){
  int i=0;
  while (i<la && a[i]<e)  i++;
  for (int j=i; j<la−1; j++) a[j]=a[j+1];
  return la−1;
}
```

(1) $Del(l,\, a,\, la,\, b,\, lb)=1+C(l,\, a,\, la,\, b,\, lb)$
$\{b\geq lb,\, lb\geq 0,\, a\geq la,\, la\geq 0,\, l\geq 0\}$
(2) $C(l,\, a,\, la,\, b,\, lb)=2\ \{a\geq la,\, b\geq lb,\, b\geq 0,\, a\geq 0,\, l=0\}$
(3) $C(l,\, a,\, la,\, b,\, lb)=$
$25+D(a,\, la,\, 0)+E(la, j)+C(l',\, a,\, la-1,\, b,\, lb)$
$\{a\geq 0,\, a\geq la,\, b\geq lb, j\geq 0,\, b\geq 0,\, l>l',\, l>0\}$
(4) $C(l,\, a,\, la,\, b,\, lb)=$
$24+D(b,\, lb,\, 0)+E(lb, j)+C(l',\, a,\, la,\, b,\, lb-1)$
$\{b\geq 0,\, b\geq lb,\, a\geq la, j\geq 0,\, a\geq 0,\, l>l',\, l>0\}$
(5) $D(a,\, la,\, i)=3\quad \{i\geq la,\, a\geq la,\, i\geq 0\}$
(6) $D(a,\, la,\, i)=8\quad \{i<la,\, a\geq la,\, i\geq 0\}$
(7) $D(a,\, la,\, i)=10+D(a,\, la,\, i+1)\ \{i<la,\, a\geq la,\, i\geq 0\}$
(8) $E(la, j)=5\quad \{j\geq la-1, j\geq 0\}$
(9) $E(la, j)=15+E(la, j+1)\quad \{j<la-1, j\geq 0\}$

Fig. 1. Java Code and the Result of Cost Analysis

an integer value p (the *pivot*), two sorted arrays of integers a and b, and two integers la and lb which indicate, respectively, the number of positions occupied in a and b. The array a (resp. b) is expected to contain values which are smaller than the pivot p (resp. greater or equal). Under the assumption that all values in l are contained in either a or b, the method del removes all values in l from the corresponding arrays. The auxiliary method rm_vec removes a given value e from an array a of length la and returns its new length, la−1.

We have applied the cost analysis in [3] on this program in order to approximate the cost of executing the method del in terms of the number of executed bytecode instructions. For this, we first compile the program to bytecode and then analyze the resulting bytecode. Fig. 1 (right) presents the results of analysis, after performing *partial evaluation*, as we will explain in Sec. 6, and inlining equality constraints (e.g., inlining equality $lb'=lb-1$ is done by replacing the occurrences of lb' by $lb-1$). In the analysis results, the data structures in the program are abstracted to their sizes: l represents the maximal *path-length* [15] of the corresponding dynamic structure, which in this case corresponds to the length of the list, a and b are the lengths of the corresponding arrays, and la and lb are the integer values of the corresponding variables. There are nine equations which define the relation Del, which corresponds to the cost of the method del, and three auxiliary recursive relations, C, D, and E. Each of them corresponds to a loop (C: while loop in del; D: while loop in rm_vec; and E: for loop in rm_vec). Each equation is annotated with a set of constraints which capture *size relations* between the values of variables in the left hand side (lhs) and those in the right hand side (rhs). In addition, size relations may contain applicability conditions (i.e., *guards*) by providing constraints which only affect variables in the lhs. Let us explain the equations for D. Eqs. (5) and (6) are base cases which correspond to the exits from the loop when i≥la and a[i]≥e, respectively. Note that the condition a[i]≥e does not appear in the size relation of Eq. (6) nor (7). This is because the array a has been abstracted to its length. Thus, the value in a[i] is no longer observable. For our cost measure , we count 3 bytecode instructions in Eq. (5) and 8 in Eq. (6). The cost of executing an iteration of the loop is

captured by Eq. (7), where the condition $i<la$ must be satisfied and variable i is increased by one at each recursive call. □

1.2 Cost Relations vs. Recurrence Relations

CRs differ from standard RRs in the following ways:

(a) *Non-determinism.* In contrast to RRs, CRs are possibly non-deterministic: equations for the same relation are not required to be mutually exclusive. Even if the programming language is deterministic, *size abstractions* introduce a loss of precision: some guards which make the original program deterministic may not be observable when using the size of arguments instead of their actual values. In Ex. 1, this happens between Eqs. (3) and (4) and also between (6) and (7).

(b) *Inexact size relations.* CRs may have size relations which contain constraints (not equalities). When dealing with realistic programming languages which contain non-linear data structures, such as trees, it is often the case that size analysis does not produce exact results. E.g., analysis may infer that the size of a data structure strictly decreases from one iteration to another, but it may be unable to provide the precise reduction. This happens in Ex. 1 in Eqs. (3) and (4).

(c) *Multiple arguments.* CRs usually depend on several arguments that may increase (variable i in Eq. (7)) or decrease (variable l in Eq. (2)) at each iteration. In fact, the number of times that a relation is executed can be a combination of several of its arguments. E.g., relation E is executed $la-j-1$ times.

Point (a) was detected already in [25], where an explicit `when` operator is added to the RR language to introduce non-determinism, but no complete method for handling it is provided. Point (b) is another source of non-determinism. As a result, CRs do not define functions, but rather relations. Given a relation C and input values $\overline{v}$, there may exist multiple results for $C(\overline{v})$. Sometimes it is possible to automatically convert relations with several arguments into relations with only one. However, in contrast to our approach, it is restricted to very simple cases such as when the CR only count constant cost expressions.

Existing methods for solving RRs are insufficient to bound CRs since they do not cover points (a), (b), and (c) above. On the other hand, CASs can solve complex recurrences (e.g., coefficients to function calls can be polynomials) which our framework cannot handle. However, this additional power is not needed in cost analysis, since such recurrences do not occur as the result of cost analysis.

An obvious way of obtaining upper bounds in non-deterministic CRs would be to introduce a maximization operator. Unfortunately, such operator is not supported by existing CAS. Adding it is far from trivial, since computing the maximum when the equations are not mutually exclusive requires taking into account multiple possibilities, which results in a highly combinatorial problem. Another possibility is to convert CRs into RRs. For this, we need to remove equations from CRs as well as sometimes to replace inexact size relations by exact ones while preserving the worst-case solution. However, this is not possible in general. E.g., in Fig. 1, the maximum cost is obtained when the execution interleaves Eqs. (3) and (4), and therefore we cannot remove either of them.

2 Cost Relations: Evaluation and Upper Bounds

Let us introduce some notation. We use x, y, z, possibly subscripted, to denote variables which range over integers ($\mathbb{Z}$), v, w denote integer values, a, b natural numbers ($\mathbb{N}$) and q rational numbers ($\mathbb{Q}$). We denote by $\mathbb{Q}^+$ (resp. $\mathbb{R}^+$) the set of non-negative rational (resp. real) numbers. We use $\overline{t}$ to denote a sequence of entities $t_1, \ldots, t_n$, for some $n{>}0$. We sometimes apply set operations on sequences. Given $\overline{x}$, an *assignment* for $\overline{x}$ is a sequence $\overline{v}$ (denoted by $[\overline{x}/\overline{v}]$). Given any entity t, $t[\overline{x}/\overline{v}]$ stands for the result of replacing in t each occurrence of x_i by v_i. We use $vars(t)$ to refer to the set of variables occurring in t. A *linear expression* has the form $q_0{+}q_1x_1{+}\cdots{+}q_nx_n$. A *linear constraint* has the form l_1 *op* l_2 where l_1 and l_2 are linear expressions and $op \in \{=, \leq, <, >, \geq\}$. A *size relation* φ is a set of linear constraints (interpreted as a conjunction). The operator $\bar{\exists}\overline{x}.\varphi$ eliminates from φ all variables except for $\overline{x}$. We write $\varphi_1 \models \varphi_2$ to indicate that φ_1 implies φ_2. The following definition presents our notion of *basic cost expression*.

Definition 1 (basic cost expression). Basic cost expressions *are of the form:* $\mathtt{exp}{::=}a|\mathsf{nat}(l)|\mathtt{exp}{+}\mathtt{exp}|\mathtt{exp}{*}\mathtt{exp}|\mathtt{exp}^a|\log_{\mathtt{a}}(\mathtt{exp})|a^{\mathtt{exp}}|\max(S)|\frac{\mathtt{exp}}{a}|\mathtt{exp}{-}a$*, where* $a{\geq}1$*,* l *is a linear expression,* S *is a non empty set of cost expressions,* $\mathsf{nat}{:}\mathbb{Z}{\to}\mathbb{Q}^+$ *is defined as* $\mathsf{nat}(v){=}\max(\{v, 0\})$*, and* $\mathtt{exp}$ *satisfies that for any assignment* $\overline{v}$ *for* $vars(\mathtt{exp})$ *we have that* $\mathtt{exp}[vars(\mathtt{exp})/\overline{v}] \in \mathbb{R}^+$*.*

Basic cost expressions are symbolic expressions which indicate the resources we accumulate and are the non-recursive building blocks for defining cost relations. They enjoy two crucial properties: (1) by definition, they are always evaluated to non negative values; (2) replacing a sub-expression $\mathsf{nat}(l)$ by $\mathsf{nat}(l')$ such that $l'{\geq}l$, results in an upper bound of the original expression.

A *cost relation* C of arity n is a subset of $\mathbb{Z}^n * \mathbb{R}^+$. This means that for a single tuple $\overline{v}$ of integers there can be multiple solutions in $C(\overline{v})$. We use C and D to refer to cost relations. Cost analysis of a program usually produces multiple, interconnected, cost relations. We refer to such sets of cost relations as *cost relation systems* (CRSs for short), which we formally define below.

Definition 2 (Cost Relation System). *A* cost relation system $\mathcal{S}$ *is a set of equations of the form* $\langle C(\overline{x}){=}\mathtt{exp}{+}\sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi\rangle$ *with* $k{\geq}0$*, where* C *and all* D_i *are cost relations, all variables* $\overline{x}$ *and* $\overline{y}_i$ *are distinct variables;* $\mathtt{exp}$ *is a basic cost expression; and* φ *is a size relation between* $\bar{x}$ *and* $\bar{x}{\cup}vars(\mathtt{exp}){\cup}\bar{y_i}$*.*

In contrast to standard definitions of RRs, the variables which occur in the rhs of the equations in CRSs do not need to be related to those in the lhs by equality constraints. Other constraints such as $\leq$ and $<$ can also be used. We denote by $rel(\mathcal{S})$ the set of cost relations which are defined in $\mathcal{S}$. Also, $def(\mathcal{S}, C)$ denotes the subset of the equations in $\mathcal{S}$ whose lhs is of the form $C(\overline{x})$. W.l.o.g. we assume that all equations in $def(\mathcal{S}, C)$ have the same variable names in the lhs. We assume that any CRS $\mathcal{S}$ is self-contained in the sense that all cost relations which appear in the rhs of an equation in $\mathcal{S}$ must be in $rel(\mathcal{S})$.

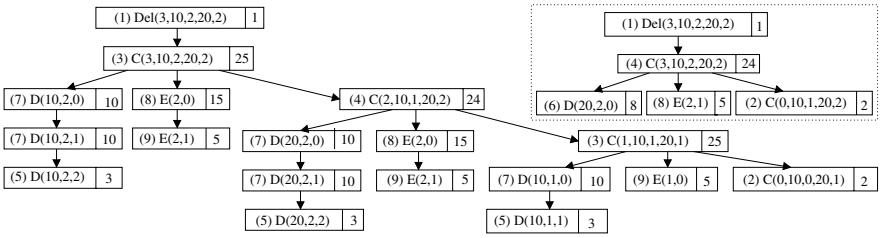


Fig. 2. Two Evaluation Trees for $Del(3, 10, 2, 20, 2)$

We now provide a semantics for CRSs. Given a CRS $\mathcal{S}$, a *call* is of the form $C(\overline{v})$, where $C{\in}rel(\mathcal{S})$ and $\overline{v}$ are integer values. Calls are evaluated in two phases. In the first phase, we build an *evaluation tree* for the call. In the second phase we obtain a *value* in $\mathbb{R}^+$ by adding up the constants which appear in the nodes of the evaluation tree. We make evaluation trees explicit since, as discussed below, our approximation techniques are based on reasoning about the number of nodes and the values in the nodes in such evaluation trees. Evaluation trees are obtained by repeatedly expanding nodes which contain calls to relations. Each expansion is performed w.r.t an appropriate instantiation of a rhs of an applicable equation. If all leaves in the tree contain basic cost expressions then there is no node left to expand and the process terminates. We will represent evaluation trees using nested terms of the form $node(Call,Local\_Cost,Children)$, where $Local\_Cost$ is a constant in $\mathbb{R}^+$ and $Children$ is a sequence of evaluation trees.

Definition 3 (evaluation tree). *Given a CRS $\mathcal{S}$ and a call $C(\overline{v})$, a tree node $(C(\overline{v}), e, \langle T_1, \ldots, T_k\rangle)$ is an* evaluation tree *for $C(\overline{v})$ in $\mathcal{S}$, denoted* $\mathsf{Tree}(C(\overline{v}), \mathcal{S})$ *if: 1) there is a renamed apart equation $\langle C(\overline{x}){=}\mathtt{exp}{+}\sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi\rangle \in \mathcal{S}$ s.t. φ' is satisfiable in $\mathbb{Z}$, with $\varphi'{=}\varphi[\overline{x}/\overline{v}]$, and 2) there exist assignments $\overline{w}, \overline{v}_i$ for $vars(\mathtt{exp}), \overline{y}_i$ respectively s.t. $\varphi'[vars(\mathtt{exp})/\overline{w},\ \overline{y}_i/\overline{v}_i]$ is satisfiable in $\mathbb{Z}$, and 3) $e{=}\mathtt{exp}[vars(\mathtt{exp})/\overline{w}]$, T_i is an evaluation tree $\mathsf{Tree}(D_i(\overline{v}_i), \mathcal{S})$ with $i = 0, \ldots, k$.*

In step 1 we look for an equation $\mathcal{E}$ which is applicable for solving $C(\overline{v})$. Note that there may be several equations which are applicable. In step 2 we look for assignments for the variables in the rhs of $\mathcal{E}$ which satisfy the size relations associated to $\mathcal{E}$. This a non-deterministic step as there may be (infinitely many) different assignments which satisfy all size relations. Finally, in step 3 we apply the assignment to exp and continue recursively evaluating the calls. We use $Trees(C(\overline{v}), \mathcal{S})$ to denote the set of all evaluation trees for $C(\overline{v})$. We define $Answers(C(\overline{v}), \mathcal{S}){=}\{\mathsf{Sum}(T) \mid T{\in}Trees(C(\overline{v}), \mathcal{S})\}$, where $\mathsf{Sum}(T)$ traverses all nodes in T and computes the sum of the cost expressions in them.

Example 2. Fig. 2 shows two possible evaluation trees for $Del(3, 10, 2, 20, 2)$. The tree on the left has maximal cost, whereas the one on the right has minimal cost. A node in either tree contains a call (left box) and its local cost (right box) and it is linked by arrows to its children. We annotate calls with a number in

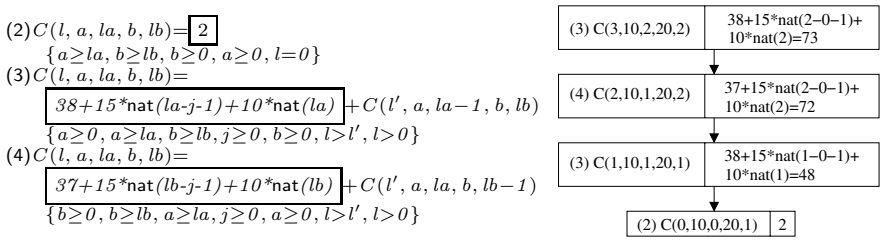


Fig. 3. Self-Contained CR for relation C and a corresponding evaluation tree

parenthesis to indicate the equation which was selected for evaluating such call. Note that, in the recursive call to C in Eqs. (3) and (4), we are allowed to pick any value l' s.t. $l'<l$. In the tree on the left we always assign $l'=l-1$. This is what happens in actual executions of the program. In the tree on the right we assign $l'=l-3$ in the recursive call to C. The latter results in a minimal approximation, however, it does not correspond to any actual execution. This is a side effect of using safe approximations in static analysis: information is correct in the sense that at least one of the evaluation trees must correspond to the actual cost, but there may be other trees with different cost. In fact, there are an infinite number of evaluation trees for our example call, as step 2 can provide an infinite number of assignments to variable j which are compatible with the constraint $j \geq 0$ in Eqs. (3) and (4). This shows that approaches like [13] based on evaluation of CRSs are not of general applicability. Nevertheless, it is possible to find an upper bound for this call since though the number of trees is infinite, infinitely many of them produce equivalent results. □

2.1 Closed Form Upper Bounds for Cost Relations

Let C be a relation over $\mathbb{Z}^n * \mathbb{R}^+$. A function $U{:}\mathbb{Z}^n \to \mathbb{R}^+$ is an *upper bound* of C iff $\forall \overline{v} \in \mathbb{Z}^n, \forall a \in Answers(C(\overline{v}), \mathcal{S}), U(\overline{v}) \geq a$. We use C^+ to refer to an upper bound of C. A function $f{:}\mathbb{Z}^n \to \mathbb{R}^+$ is in *closed form* if it is defined as $f(\overline{x})=\mathtt{exp}$, with $\mathtt{exp}$ a basic cost expression s.t. $vars(\mathtt{exp}) \subseteq \overline{x}$. An important feature of CRSs, inherited from RRs, is their *compositionality*, which allows computing upper bounds of CRSs by concentrating on one relation at a time. I.e., given a cost equation for $C(\overline{x})$ which calls $D(\overline{y})$, we can replace the call to $D(\overline{y})$ by $D^+(\overline{y})$. The resulting relation is trivially an upper bound of the original one. E.g., suppose that we have the following upper bounds: $E^+(la, j)=5+15*\mathsf{nat}(la-j-1)$ and $D^+(a, la, i)=8+10*\mathsf{nat}(la-i)$. Replacing the calls to D and E in equations (3) and (4) by D^+ and E^+ results in the CRS shown in Fig. 3.

The compositionality principle only results in an effective mechanism if all recursions are *direct* (i.e., all cycles are of length one). In that case we can start by computing upper bounds for cost relations which do not depend on any other relations, which we refer to as *standalone cost relations* and continue by replacing

the computed upper bounds on the equations which call such relations. In the following, we formalize our method by assuming standalone cost relations and in Sec. 6 we provide a mechanism for obtaining direct recursion automatically.

Existing approaches to compute upper bounds and asymptotic complexity of RRs, usually applied by hand, are based on reasoning about evaluation trees in terms of their size, depth, number of nodes, etc. They typically consider two categories of nodes: (1) *internal* nodes, which correspond to applying recursive equations, and (2) *leaves* of the tree(s), which correspond to the application of a base (non-recursive) case. The central idea then is to count (or obtain an upper bound on) the number of leaves and the number of internal nodes in the tree separately and then multiply each of these by an upper bound on the cost of the base case and of a recursive step, respectively. For instance, in the evaluation tree in Fig. 3 for the standalone cost relation C, there are three internal nodes and one leaf. The values in the internal nodes, once performed the evaluation of the expressions are 73, 72, and 48, therefore 73 is the worst case. In the case of leaves, the only value is 2. Therefore, the tightest upper bound we can find using this approximation is $3{\times}73{+}1{*}2{=}221 \geq 73{+}72{+}48{+}2{=}193$.

We now extend the approximation scheme mentioned above in order to consider all possible evaluation trees which may exist for a call. In the following, we use $|S|$ to denote the cardinality of a set S. Also, given an evaluation tree T, $\mathit{leaf}(T)$ denotes the set of leaves of T (i.e., those without children) and $\mathit{internal}(T)$ denotes the set of internal nodes (all nodes but the leaves) of T.

Proposition 1 (node-count upper bound). *Let C be a cost relation and let $C^+(\overline{x}) = \mathit{internal}^+(\overline{x}) * \mathit{costr}^+(\overline{x}) + \mathit{leaf}^+(\overline{x}) * \mathit{costnr}^+(\overline{x})$, where $\mathit{internal}^+(\overline{x})$, $\mathit{costr}^+(\overline{x})$, $\mathit{leaf}^+(\overline{x})$ and $\mathit{costnr}^+(\overline{x})$ are closed form functions defined on $\mathbb{Z}^n \to \mathbb{R}^+$. Then, C^+ is an upper bound of C if for all $\overline{v} \in \mathbb{Z}^n$ and for all $T \in \mathit{Trees}(C(\overline{v}), \mathcal{S})$, it holds: (1) $\mathit{internal}^+(\overline{v}) \geq |\mathit{internal}(T)|$ and $\mathit{leaf}^+(\overline{v}) \geq |\mathit{leaf}(T)|$; (2) $\mathit{costr}^+(\overline{v})$ is an upper bound of $\{e \mid \mathit{node}(\_, e, \_) \in \mathit{internal}(T)\}$ and (3) $\mathit{costnr}^+(\overline{v})$ is an upper bound of $\{e \mid \mathit{node}(\_, e, \_) \in \mathit{leaf}(T)\}$.*

3 Upper Bounds on the Number of Nodes

In this section we present an automatic mechanism to obtain safe $\mathit{internal}^+(\overline{x})$ and $\mathit{leaf}^+(\overline{x})$ functions which are valid for any assignment for $\overline{x}$. The basic idea is to first obtain upper bounds b and $h^+(\overline{x})$ on, respectively, the *branching factor* and *height* (the distance from the root to the deepest leaf) of all corresponding evaluation trees, and then use the number of internal nodes and leaves of a *complete* tree with such branching factor and height as an upper bound. Then,

$$\mathit{leaf}^+(\overline{x}) = b^{h^+(\overline{x})} \qquad \mathit{internal}^+(\overline{x}) = \begin{cases} h^+(\overline{x}) & b{=}1 \\ \frac{b^{h^+(\overline{x})}-1}{b-1} & b{\geq}2 \end{cases}$$

For a cost relation C, the branching factor b in any evaluation tree for a call $C(\overline{v})$ is limited by the maximum number of recursive calls which occur in a single equation for C. We now propose a way to compute an upper bound for

the height, h^+. Given an evaluation tree $T \in Trees(C(\overline{v}), \mathcal{S})$ for a cost relation C, consecutive nodes in any branch of T represent consecutive recursive calls which occur during the evaluation of $C(\overline{v})$. Therefore, bounding the height of a tree may be reduced to bounding consecutive recursive calls. The notion of *loop* in a cost relation, which we introduce below, is used to model consecutive calls.

Definition 4. *Let* $\mathcal{E}=\langle C(\overline{x})=\mathsf{exp}+\sum_{i=1}^{k} C(\overline{y_i}), \varphi\rangle$ *be an equation for a cost relation* C*. Then,* $Loops(\mathcal{E})=\{\langle C(\overline{x})\rightarrow C(\bar{y}_i), \varphi'\rangle \mid \varphi'=\bar{\exists}\bar{x}\cup\bar{y}_i.\varphi, i=1\cdots k\}$ *is the set of* loops *induced by* $\mathcal{E}$*. Similarly,* $Loops(C) = \cup_{\mathcal{E}\in def(\mathcal{S},C)} Loops(\mathcal{E})$*.*

Example 3. Eqs. (3) and (4) in Fig. 3 induce the following two loops:

(3)$\langle C(l, a, la, b, lb)\rightarrow C(l', a, la', b, lb), \varphi'_1=\{a\geq 0, a\geq la, b\geq lb, b\geq 0, l>l', l>0, la'=la-1\}\rangle$
(4)$\langle C(l, a, la, b, lb)\rightarrow C(l', a, la, b, lb'), \varphi'_2=\{b\geq 0, b\geq lb, a\geq la, a\geq 0, l>l', l>0, lb'=lb-1\}\rangle$

Bounding the number of consecutive recursive calls is extensively used in the context of termination analysis. It is usually done by proving that there is a function f from the loop's arguments to a *well-founded* partial order which decreases in any two consecutive calls and which guarantees the absence of infinite traces, and thus termination. These functions are usually called *ranking functions*. We propose to use the ranking function to generate a h^+ function. In practice, we use [21] to generate functions which are defined as follows: a function $f:\mathbb{Z}^n\mapsto\mathbb{Z}$ is a *ranking function* for a loop $\langle C(\bar{x})\rightarrow C(\bar{y}), \varphi\rangle$ if $\varphi\models f(\bar{x})>f(\bar{y})$ and $\varphi\models f(\bar{x})\geq 0$.

Example 4. The function $f_C(l, a, la, b, lb)=l$ is a ranking function for C in the cost relation in Fig. 3. Note that φ'_1 and φ'_2 in the above loops of C contain the constraints $\{l>l', l>0\}$ which is enough to guarantee that f_C is decreasing and well-founded. The height of the evaluation tree for $C(3, 10, 2, 20, 2)$ is precisely predicted by $f_C(3, 10, 2, 20, 2)=3$. Ranking functions may involve several arguments, e.g., $f_D(a, la, i)=la-i$ is a ranking function for $\langle D(a, la, i) \rightarrow D(a, la, i'), \{i'=i+1, i<la, a\geq la, i\geq 0\}\rangle$ which comes from Eq. (7). □

Observe that the use of global ranking functions allows bounding the number of iterations of possibly non-deterministic CRSs with multiple arguments (see Sec. 1.2). In order to be able to define h^+ in terms of the ranking function, one thing to fix is that the ranking function might return a negative value when is applied to values which correspond to base cases (leaves of the tree). Therefore, we define $h^+(\overline{x})=\mathsf{nat}(f_C(\overline{x}))$. Function nat guarantees that negative values are lifted to 0 and, therefore, they provide a correct approximation for the height of evaluation trees with a single node. Even though the ranking function provides an upper bound for the height of the corresponding trees, in some cases we can further refine it and obtain a tighter upper bound. For example, if the difference between the value of the ranking function in each two consecutive calls is larger than a constant $\delta>1$, then $\lceil \mathsf{nat}(\frac{f_C(\bar{x})}{\delta})\rceil$ is a tighter upper bound. A more interesting case, if each loop $\langle C(\overline{x})\rightarrow C(\overline{y}), \varphi\rangle \in Loops(C)$ satisfies $\varphi\models f_C(\overline{x})\geq k * f_C(\overline{y})$ where $k>1$, then the height of the tree is bounded by $\lceil \log_k(\mathsf{nat}(f_C(\overline{v})+1))\rceil$.

4 Estimating the Cost Per Node

Consider the evaluation tree in Fig. 3. Note that all expressions in the nodes are instances of the expressions which appear in the corresponding equations. Thus, computing $costr^+(\overline{x})$ and $costnr^+(\overline{x})$ can be done by first finding an upper bound of such expressions and then combining them through a *max* operator. We first compute *invariants* for the values that the expression variables can take w.r.t. the initial values, and use them to derive upper bounds for such expressions.

4.1 Invariants

Computing an *invariant* (in terms of linear constraints) that holds in all *calling contexts* (*contexts* for short) to a relation C between the arguments at the initial call and at each call during the evaluation can be done by using $Loops(C)$. Intuitively, if we know that a linear constraint ψ holds between the arguments of the initial call $C(\overline{x}_0)$ and those of a recursive call $C(\overline{x})$, denoted $\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}), \psi\rangle$, and we have a loop $\langle C(\overline{x}){\rightarrow}C(\overline{y}), \varphi\rangle{\in}Loops(C)$, then we can apply the loop one more step and get the new *calling context* $\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{y}), \overline{\exists\overline{x_0}\cup\overline{y}.\psi\wedge\varphi}\rangle$.

Definition 5 (loop invariants). *For a relation C, let $\mathcal{T}$ be an operator defined:*

$$\mathcal{T}(X) = \left\{ \langle C(\overline{x}_0){\rightsquigarrow}C(\overline{y}), \psi'\rangle \;\middle|\; \begin{array}{l}\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}), \psi\rangle{\in}X, \langle C(\overline{x}){\rightarrow}C(\overline{y}), \varphi\rangle{\in}Loops(C),\\ \psi'=\overline{\exists\overline{x_0}\cup\overline{y}.\psi\wedge\varphi}\end{array} \right\}$$

which derives a set of contexts, from a given context X, by applying all loops, then the loop invariants I is $lfp{\cup_{i\geq 0}}\mathcal{T}^i(I_0)$ where $I_0 = \{\langle C(\overline{x}_0){\rightsquigarrow}C(\overline{x}), \{\overline{x}_0{=}\overline{x}\}\rangle\}$.

Example 5. Let us compute I for the loops in Sec. 3. The initial context is $I_1{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{=}l_0, a{=}a_0, la{=}la_0, b{=}b_0, lb{=}lb_0\}\rangle$ where $\bar{x}_0{=}\langle l_0, a_0, la_0, b_0, lb_0\rangle$ and $\bar{x}{=}\langle l, a, la, b, lb\rangle$. In the first iteration we compute $\mathcal{T}^0(\{I_1\})$ which by definition is $\{I_1\}$. In the second iteration we compute $\mathcal{T}^1(\{I_1\})$ which results in

$$I_2{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{\underline{l{<}l_0}, a{=}a_0, \underline{la{=}la_0{-}1}, b{=}b_0, lb{=}lb_0, \underline{l_0{>}0}\}\rangle$$
$$I_3{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{\underline{l{<}l_0}, a{=}a_0, la{=}la_0, b{=}b_0, \underline{lb{=}lb_0{-}1}, \underline{l_0{>}0}\}\rangle$$

where I_2 and I_3 correspond to applying respectively the first loop and second loops on I_1. The underlined constraints are the modifications due to the application of the loop. Note that in I_2 the variable la_0 decreases by one, and in I_3 lb_0 decreases by one. The third iteration $\mathcal{T}^2(\{I_1\})$, i.e. $\mathcal{T}(\{I_2, I_3\})$, results in

$$I_4{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{<}l_0, a{=}a_0, \underline{la{=}la_0{-}2}, b{=}b_0, lb{=}lb_0, l_0{>}0\}\rangle$$
$$I_5{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{<}l_0, a{=}a_0, \underline{la{=}la_0{-}1}, b{=}b_0, \underline{lb{=}lb_0{-}1}, l_0{>}0\}\rangle$$
$$I_6{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{<}l_0, a{=}a_0, la{=}la_0, b{=}b_0, \underline{lb{=}lb_0{-}2}, l_0{>}0\}\rangle$$
$$I_7{=}\langle C(\bar{x}_0){\rightsquigarrow}C(\bar{x}), \{l{<}l_0, a{=}a_0, \underline{la{=}la_0{-}1}, b{=}b_0, lb{=}lb_0{-}1, l_0{>}0\}\rangle$$

where I_4 and I_5 originate from applying the loops to I_2, and I_6 and I_7 from applying the loops to I_3. The modifications on the constraints reflect that, when applying a loop, either we decrease *la* or *lb*. After three iterations, the invariant I includes $I_1 \cdots I_7$. More iterations will add more contexts that further modify the value of *la* or *lb*. Therefore, the invariant I grows indefinitely in this case. □

In practice, we approximate I using abstract interpretation over, for instance, the domain of convex polyhedra [10], whereby we obtain the invariant $\Psi = \langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \{l \leq l_0, a = a_0, la \leq la_0, b = b_0, lb \leq lb_0\}\rangle$.

4.2 Upper Bounds on Cost Expressions

Once invariants are available, finding upper bounds of cost expressions can be done by maximizing their nat parts independently. This is possible due to the monotonicity property of cost expressions. Consider, for example, the expression $\mathsf{nat}(la-j-1)$ which appears in equation (3) of Fig. 3. We want to infer an upper bound of the values that it can be evaluated to in terms of the input values $\langle l_0, a_0, la_0, b_0, lb_0\rangle$. We have inferred, in Sec. 4.1, that whenever we call C the invariant Ψ holds, from which we can see that the maximum value that la can take is la_0. In addition, from the local size relations φ of equation (3) we know that $j \geq 0$. Since $la-j-1$ takes its maximal value when la is maximal and j is minimal, the expression la_0-1 is an upper bound for $la-j-1$. This can be done automatically using linear constraints tools [6]. Given a cost equation $\langle C(\overline{x}) = \mathtt{exp} + \sum_{i=0}^{k} C(\overline{y}_i),\ \varphi\rangle$ and an invariant $\langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \Psi\rangle$, the function below computes an upper bound for exp by maximizing its nat components.

```
1: function ub_exp(exp,x̄0,φ,Ψ)
2:     mexp=exp
3:     for all nat(f)∈exp do
4:         Ψ'=∃x̄0, r.(φ∧Ψ∧(r=f))        // r is a fresh variable
5:         if ∃f' s.t. vars(f')⊆x̄0 and Ψ'⊨r≤f' then mexp=mexp[nat(f)/nat(f')]
6:         else return ∞
7:     return mexp
```

This function computes an upper bound f' for each expression f which occurs inside a nat operator and then replaces in exp all such f expressions with their corresponding upper bounds (line 5). If it cannot find an upper bound, the method returns ∞ (line 6). The *ub_exp* function is complete in the sense that if Ψ and φ imply that there is an upper bound for a given $\mathsf{nat}(f)$, then we can find one by *syntactically* looking on Ψ' (line 4).

Example 6. Applying *ub_exp* to $\mathtt{exp}_3$ and $\mathtt{exp}_4$ of Eqs. (3) and (4) in Fig. 3 w.r.t. the invariant we have computed in Sec. 4.1 results in $\mathtt{mexp}_3 = 38 + 15 * \mathsf{nat}(la_0 - 1) + 10 * \mathsf{nat}(la_0)$ and $\mathtt{mexp}_4 = 37 + 15 * \mathsf{nat}(lb_0 - 1) + 10 * \mathsf{nat}(lb_0)$. □

Theorem 1. *Let $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ be a cost relation where $\mathcal{S}_1$ and $\mathcal{S}_2$ are respectively the sets of non-recursive and recursive equations for C, and let $\mathcal{I} = \langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \Psi\rangle$ be a loop invariant for C; $E_i = \{ub\_exp(\mathtt{exp}, \overline{x}_0, \varphi, \Psi) \mid \langle C(\overline{x}) = \mathtt{exp} + \sum_{j=0}^{k} C(\overline{y}_j), \varphi\rangle \in \mathcal{S}_i\}$; $costnr^+(\overline{x}_0) = \max(E_1)$ and $costr^+(\overline{x}_0) = \max(E_2)$. Then for any call $C(\overline{v})$ and for all $T \in Trees(C(\overline{v}), \mathcal{S})$: (1) $\forall node(\_, e, \_) \in internal(T)$ we have $costr^+(\overline{v}) \geq e$; and (2) $\forall node(\_, e, \_) \in leaf(T)$ we have $costnr^+(\overline{v}) \geq e$.*

Example 7. At this point we have all the pieces in order to compute an upper bound for the CRS depicted in Fig. 1 as described in Prop. 1. We start by computing upper bounds for E and D as they are cost relations:

	Ranking Function	$costnr^+$	$costr^+$	*Upper Bound*
$E(la_0, j_0)$	$\mathsf{nat}(la_0{-}j_0{-}1)$	5	15	$5{+}15{*}\mathsf{nat}(la_0{-}j_0{-}1)$
$D(a_0, la_0, i_0)$	$\mathsf{nat}(la_0{-}i_0)$	8	10	$8{+}10{*}\mathsf{nat}(la_0{-}i_0)$

These upper bounds can then be substituted in the equations (3) and (4) which results in the cost relation for C depicted in Fig. 3. We have already computed a ranking function for C in Ex. 4 and $costnr^+$ and $costr^+$ in Ex. 6, which are then combined into $C^+(l_0, a_0, la_0, b_0, lb_0){=}2{+}\mathsf{nat}(l_0){*}max(\{\mathtt{mexp}_3, \mathtt{mexp}_4\})$. Reasoning similarly, for *Del* we get the upper bound shown in Table 1. □

5 Improving Accuracy in Divide and Conquer Programs

For some CRSs, we can obtain a more accurate upper bound by approximating the cost of *levels* instead of approximating the cost of nodes, as indicated by Prop. 1. Given an evaluation tree T, we denote by $\mathsf{Sum\_Level}(T, i)$ the sum of the values of all nodes in T which are at depth i, i.e., at distance i from the root.

Proposition 2 (level-count upper bound). *Let C be a cost relation and let C^+ be a function defined as: $C^+(\overline{x}){=}l^+(\overline{x}) * costl^+(\overline{x})$, where $l^+(\overline{x})$ and $costl^+(\overline{x})$ are closed form functions defined on $\mathbb{Z}^n{\to}\mathbb{R}^+$. Then, C^+ is an upper bound of C if for all $\overline{v}{\in}\mathbb{Z}^n$ and $T{\in}Trees(C(\overline{v}), \mathcal{S})$, it holds: (1) $l^+(\overline{v}) \geq depth(T) + 1$; and (2) $\forall i{\in}\{0, \ldots, depth(T)\}$ we have that $costl^+(\overline{v}) \geq \mathsf{Sum\_Level}(T, i)$.*

The function l^+ can simply be defined as $l^+(\overline{x}){=}\mathsf{nat}(f_C(\overline{x})){+}1$ (see Sec. 3). Finding an accurate $costl^+$ function is not easy in general, which makes Prop. 2 not as widely applicable as Prop. 1. However, evaluation trees for *divide and conquer* programs satisfy that $\mathsf{Sum\_Level}(T, k){\geq}\mathsf{Sum\_Level}(T, k{+}1)$, i.e., the cost per level does not increase from one level to another. In that case, we can take the cost of the root node as an upper bound of $costl^+(\overline{x})$. A sufficient condition for a cost relation falling into the divide and conquer class is that each cost expression that is contributed by an equation is greater than or equal to the sum of the cost expressions contributed by the corresponding immediate recursive calls. This check is implemented in our prototype using [6].

Consider a CRS with the two equations $\langle C(n){=}0, \{n{\leq}\,0\}\rangle$ and $\langle C(n) = \mathsf{nat}(n){+}C(n_1){+}C(n_2), \varphi\rangle$ where $\varphi{=}\{n{>}0, n_1{+}n_2{+}1{\leq}n, n{\geq}2{*}n_1, n \geq 2{*}n_2, n_1{\geq}0, n_2{\geq}0\}$. It corresponds to a divide and conquer problem such as merge-sort. In order to prove that $\mathsf{Sum\_Level}$ does not increase, it is enough to check that, in the second equation, n is greater than or equal to the sum of the expressions that immediately result from the calls $C(n_1)$ and $C(n_2)$, which are n_1 and n_2 respectively. This can be done by simply checking that $\varphi{\models}n{\geq}n_1{+}n_2$. Then, $costl^+(\overline{x}){=}max\{0, \mathsf{nat}(x)\}{=}\mathsf{nat}(x)$. Thus, given that $l^+(x){=}\lceil\log_2(\mathsf{nat}(x){+}1)\rceil{+}1$, we obtain the upper bound $\mathsf{nat}(x){*}(\lceil\log_2(\mathsf{nat}(x){+}1)\rceil{+}1)$. Note that by using the node-count approach we would obtain $\mathsf{nat}(x){*}(2^{\mathsf{nat}(x)}{-}1)$ as upper bound.

6 Direct Recursion Using Partial Evaluation

Automatically generated CRSs often contain recursions which are not direct, i.e., cycles involve more than one function. E.g., the actual CRS obtained for

the program in Fig. 1 by the analysis in [3] differs from that shown in the right hand side of Fig. 1 in that, instead of Eqs. (8) and (9), the "for" loop results in:

(8') $E(la,j)=5+F(la,j,j',la')$ $\{j'=j, la'=la-1, j'\geq 0\}$
(9') $F(la,j,j',la')=H(j',la')$ $\{j'\geq la'\}$
(10) $F(la,j,j',la')=G(la,j,j',la')$ $\{j'<la'\}$
(11) $H(j',la')=0$ $\{\}$
(12) $G(la,j,j',la')=10+E(la,j+1)$ $\{j<la-1, j\geq 0, la-la'=1, j'=j\}$

Now, E captures the cost of the loop condition "$j<la-1$" (5 cost units) plus the cost of its continuation, captured by F. Eq. (9') corresponds to the exit of the loop (it calls H, Eq. (11), which has 0 cost). Eq. (10) captures the cost of one iteration by calling G, Eq. (12), which accumulates 10 units and returns to E.

In this section we present an automatic transformation of CRSs into *directly recursive* form. The transformation is based on *partial evaluation* (PE) [17] and it is performed by replacing calls to intermediate relations by their definitions using *unfolding*. The first step in the transformation is to find a *binding time classification* (or BTC for short) which declares which relations are *residual*, i.e., they have to remain in the CRS. The remaining relations are considered *unfoldable*, i.e., they are eliminated. For computing BTCs, we associate to each CRS $\mathcal{S}$ a *call graph*, denoted $\mathcal{G}(\mathcal{S})$, which is the directed graph obtained from $\mathcal{S}$ by taking $rel(\mathcal{S})$ as the set of nodes and by including an arc (C, D) iff D appears in the rhs of an equation for C. The following definition provides sufficient conditions on a BTC which guarantee that we obtain a directly recursive CRS.

Definition 6. *Let $\mathcal{G}(\mathcal{S})$ be the call graph of $\mathcal{S}$ and let SCC be its strongly connected components. A BTC* btc *for $\mathcal{S}$ is* directly recursive *if for all $S\in SCC$ the following conditions hold: (1) if $s_1, s_2\in S$ and $s_1, s_2\in$ btc, then $s_1=s_2$; and (2) if S has a cycle, then there exists $s\in S$ such that $s\in$ btc.*

Condition 1 ensures that all recursions in the transformed CRS are direct, as there is only one residual relation per SCC. Condition 2 guarantees that the unfolding process terminates, as there is a residual relation per cycle. A directly recursive BTC for the above example is btc$=\{E\}$. In our implementation we only include in the BTC the *covering point* (i.e., a node which is part of all cycles) of SCCs which contain cycles, but no node is included for SCCs without cycles. This way of computing BTCs, in addition to ensuring direct recursion, also eliminates all relations which are not part of cycles (such as H in our example).

We now define unfolding in the context of CRSs. Such unfolding is guided by a BTC and at each step it combines both cost expressions and size relations.

Definition 7 (unfolding). *Given a CRS $\mathcal{S}$, a call $C(\overline{x}_0)$ s.t. $C\in rel(\mathcal{S})$, a size relation $\varphi_{\overline{x}_0}$ over $\overline{x}_0$, and a BTC* btc *for $\mathcal{S}$, a pair $\langle E, \varphi\rangle$ is an* unfolding *for $C(\overline{x}_0)$ and $\varphi_{\overline{x}_0}$ in $\mathcal{S}$ w.r.t.* btc*, denoted* $\mathsf{Unfold}(\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$*, if either of the following conditions hold:*

(res) $C\in\mathsf{btc}\wedge\varphi\neq\mathsf{true}\wedge\langle E, \varphi\rangle=\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle$;
(unf) $(C\notin\mathsf{btc}\vee\varphi=\mathsf{true})\wedge\langle E, \varphi\rangle=\langle(\mathsf{exp}+e_1+\ldots+e_k), \varphi' \bigwedge_{i=1..k} \varphi_i\rangle$

where $\langle C(\overline{x})=\mathsf{exp}+\sum_{i=1}^{k} D_i(\overline{y}_i), \varphi_C\rangle$ is a renamed apart equation in $\mathcal{S}$ s.t. $\varphi' = \varphi_{\overline{x}_0}\wedge\varphi_C[\overline{x}/\overline{x}_0]$ is satisfiable in $\mathbb{Z}$ and $\forall 1\leq i\leq k$ $\mathsf{Unfold}(\langle D_i(\overline{y_i}), \varphi'\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle e_i, \varphi_i\rangle$.

The first case, (**res**), is required for termination. When we call a relation C which is marked as residual, we simply return the initial call $C(\overline{x}_0)$ and size relation $\varphi_{\overline{x}_0}$, as long as the current size relation $\varphi_{\overline{x}_0}$ is not the initial one (true). The latter condition is added in order to force the initial unfolding step for relations marked as residual. In all subsequent calls to Unfold different from the initial one, the size relation is different from true. The second case (**unf**) corresponds to continuing the unfolding process. Note that step 1 is non-deterministic, since often cost relations contain several equations. Since expressions are transitively unfolded, step 2 may also provide multiple solutions. Also, if the final size relation φ is unsatisfiable, we simply do not regard $\langle E, \varphi\rangle$ as a valid unfolding.

Example 8. Given the initial call $\langle E(la, j), true\rangle$, we obtain an unfolding by performing the following steps, denoted by $\overset{e}{\rightsquigarrow}$ where e is the selected equation:
$\langle E(la,j), true\rangle \overset{(8')}{\rightsquigarrow} \langle 5{+}F(la,j,j',la'), \{j'{=}j, la'{=}la{-}1, j'{\geq}0\}\rangle \overset{(10)}{\rightsquigarrow}$
$\langle 5{+}G(la,j,j',la'), \{j'{=}j, la'{=}la{-}1, j'{\geq}0, j'{<}la'\}\rangle \overset{(12)}{\rightsquigarrow} \langle 15{+}E(la,j''), \{j{<}la{-}1, j{\geq}0\}\rangle$
The call $E(la, j'')$ is not further unfolded as E belongs to btc and $\varphi{\neq}true$. □

From each result of unfolding we can build a *residual equation*. Given the unfolding $\mathsf{Unfold}(\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) \rightsquigarrow \langle E, \varphi\rangle$ its corresponding residual equation is $\langle C(\overline{x}_0){=}E, \varphi\rangle$. As customary in PE, a *partial evaluation* of C is obtained by collecting all residual equations for the call $\langle C(\overline{x}_0), \mathsf{true}\rangle$. The PE of $\langle E(la, j), true\rangle$ results in Eqs. (8) and (9) of Fig. 1. Eq. (9) is obtained from the unfolding steps depicted in Ex. 8 and Eq. (8) from unfolding w.r.t. Eqs. (8'), (9'), and (11).

Correctness of PE ensures that the solutions of CRSs are preserved. Regarding completeness, we can obtain direct recursion if all SCCs in the call graph have covering point(s). Importantly, structured loops (for, while, etc.) and recursive patterns found in most programs result in CRSs that satisfy this property. In addition, before applying PE, we check that the CRS terminates [2] with respect to the initial query, otherwise we might compromise non-termination and thus lead to incorrect upper bounds. We believe this check is not required when CRSs are generated from imperative programs.

7 Experiments in Cost Analysis of Java Bytecode

A prototype implementation in Ciao Prolog, which uses PPL [6] for manipulating linear constraints, is available at `http://www.cliplab.org/Systems/PUBS`. We have performed a series of experiments which are shown in Table 1. We have used CRSs automatically generated by the cost analyzer of Java bytecode described in [3] using two cost measures: heap consumption for those marked with "*", and the number of executed bytecode instructions for the rest. The benchmarks are presented in increasing complexity order and grouped by asymptotic class. Those marked with M were solved using Mathematica® by [3] but after significant human intervention. The marks a, b and c after the name indicate, respectively, if the CRS is non-deterministic, has inexact size relations and multiple arguments (Sec. 1.2). Column $\#_{eq}$ shows the number of equations before PE (in brackets

Table 1. Experiments on Cost Analysis of Java Bytecode

Benchmark		$\#_{eq}$	T	$\#^c_{eq}$	$\mathbf{T}_{pe}$	$\mathbf{T}_{ub}$	Rat.	Upper Bound
Polynomial*	abc	23 (3)	13	346 (70)	174	649	2.4	216
DivByTwo	ab	9 (3)	3	323 (68)	166	596	2.4	$8\log_2(\mathsf{nat}(2x{-}1){+}1){+}14$
Factorial$^{\mathsf{M}}$		8 (2)	4	314 (66)	165	590	2.4	$9\mathsf{nat}(x){+}4$
ArrayRev$^{\mathsf{M}}$	a	9 (3)	4	305 (64)	165	579	2.4	$14\mathsf{nat}(x){+}12$
Concat$^{\mathsf{M}}$	ac	14 (5)	13	296 (62)	158	538	2.4	$11\mathsf{nat}(x){+}11\mathsf{nat}(y){+}25$
Incr$^{\mathsf{M}}$	ac	28 (5)	29	282 (58)	155	490	2.3	$19\mathsf{nat}(x{+}1){+}9$
ListRev$^{\mathsf{M}}$	abc	9 (3)	4	254 (54)	144	415	2.2	$13\mathsf{nat}(x){+}8$
MergeList	abc	21 (4)	18	245 (52)	138	406	2.2	$29\mathsf{nat}(x{+}y){+}26$
Power		8 (2)	3	223 (48)	125	371	2.2	$10\mathsf{nat}(x){+}4$
Cons*	ab	22 (2)	6	214 (46)	123	359	2.3	$22\mathsf{nat}(x{-}1){+}24$
EvenDigits	abc	18 (5)	9	191 (44)	115	322	2.3	$\mathsf{nat}(x)(8\log_2(\mathsf{nat}(2x{-}3){+}1){+}24){+}9\mathsf{nat}(x){+}9$
ListInter	abc	37 (9)	59	173 (40)	110	298	2.4	$\mathsf{nat}(x)(10\mathsf{nat}(y){+}43){+}21$
SelectOrd	ac	19 (6)	27	136 (32)	86	198	2.1	$\mathsf{nat}(x{-}2)(17\mathsf{nat}(x{-}2){+}34){+}9$
FactSum	a	17 (5)	8	117 (27)	76	173	2.1	$\mathsf{nat}(x{+}1)(9\mathsf{nat}(x){+}16){+}6$
Delete	abc	33 (9)	125	100 (23)	71	165	2.4	$3{+}\mathsf{nat}(l)\max(38{+}15\mathsf{nat}(la{-}1){+}10\mathsf{nat}(la),$ $37{+}15\mathsf{nat}(lb{-}1){+}10\mathsf{nat}(lb))$
MatMult$^{\mathsf{M}}$	ac	19 (7)	23	67 (15)	27	40	1.0	$\mathsf{nat}(y)(\mathsf{nat}(x)(27\mathsf{nat}(x)){+}10){+}17$
Hanoi		9 (2)	4	48 (8)	23	17	0.8	$20(2^{\mathsf{nat}(x)})$-17
Fibonacci$^{\mathsf{M}}$		8 (2)	5	39 (6)	20	13	0.8	$18(2^{\mathsf{nat}(x-1)})$-13
BST*	ab	31 (4)	26	31 (4)	19	7	0.9	$96(2^{\mathsf{nat}(x)})$-49

after PE). Note that PE greatly reduces $\#_{eq}$ in all benchmarks. Column **T** shows the total runtime in milliseconds. The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM, running Linux.

The next four columns aim at demonstrating the scalability of our approach. To do so, we connect the CRSs for the different benchmarks by introducing a call from each CRS to the one appearing immediately below it in the table. Such call is always introduced in a recursive equation. Column $\#^c_{eq}$ shows the number of equations we want to solve in each case (in brackets after PE). Reading this column bottom-up, we can see that BST has the same number of equations as the original one and that, progressively, each benchmark adds its own number of equations to $\#^c_{eq}$. Thus, in the first row we have a CRS with all the equations connected, i.e., we compute an upper bound of CRS with at least 19 nested loops and 346 equations. The total runtime is split into $\mathbf{T}_{pe}$ and $\mathbf{T}_{ub}$, where $\mathbf{T}_{pe}$ is the time of PE and it shows that even though PE is a *global* transformation, its time efficiency is linear with the number of equations. Our system solves 346 equations in $823ms$. Column **Rat**. shows the total time per equation. The ratio is small for benchmarks with few equations, and for reasonably large CRSs (from Delete upwards) it almost has no variation (2.1–2.4 ms/eq). The small increase is due to the fact that the equations count more complex expressions as we connect more benchmarks. This demonstrates that our approach is totally scalable, even if the implementation is preliminary. The upper bound expressions get considerably large when the benchmarks are composed together. We are currently implementing standard techniques for simplification of arithmetic expressions.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

References

1. Adachi, A., Kasai, T., Moriya, E.: A theoretical study of the time analysis of programs. In: Becvar, J. (ed.) MFCS 1979. LNCS, vol. 74, pp. 201–207. Springer, Heidelberg (1979)
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Proc. of FMOODS. LNCS, Springer, Heidelberg (to appear, 2008)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
4. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile Resource Guarantees for Smart Devices. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, Springer, Heidelberg (2005)
5. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Technical report, arXiv:cs/0512056 (2005), `http://arxiv.org/`
6. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
7. Benzinger, R.: Automated higher-order complexity analysis. In: TCS, vol. 318(1-2) (2004)
8. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations and small space bounds. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, Springer, Heidelberg (2005)
9. Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.: Enforcing resource bounds via static verification of dynamic checks. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)
11. Crary, K., Weirich, S.: Resource bound certification. In: POPL (2000)
12. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. TOPLAS 15(5) (1993)
13. Gómez, G., Liu, Y.A.: Automatic time-bound analysis for a higher-order language. In: PEPM, ACM Press, New York (2002)
14. Hickey, T., Cohen, J.: Automating program analysis. J. ACM 35(1) (1988)
15. Hill, P.M., Payet, E., Spoto, F.: Path-length analysis of object-oriented programs. In: Proc. EAAI, Elsevier, Amsterdam (2006)
16. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL (2003)
17. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
18. Le Metayer, D.: ACE: An Automatic Complexity Evaluator. TOPLAS 10(2) (1988)

19. Marion, J.-Y., Péchoux, R.: Resource analysis by sup-interpretation. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, Springer, Heidelberg (2006)
20. Navas, J., Mera, E., López-García, P., Hermenegildo, M.: User-definable resource bounds analysis for logic programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, Springer, Heidelberg (2007)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, Springer, Heidelberg (2004)
22. Rosendahl, M.: Automatic Complexity Analysis. In: FPCA, ACM Press, New York (1989)
23. Sands, D.: A naïve time analysis and its theory of cost equivalence. Journal of Logic and Computation 5(4) (1995)
24. Wadler, P.: Strictness analysis aids time analysis. In: POPL (1988)
25. Wegbreit, B.: Mechanical Program Analysis. Comm. of the ACM 18(9) (1975)

Heap Space Analysis for Java Bytecode

Elvira Albert
DSIC, Complutense University of Madrid, Spain
elvira@sip.ucm.es

Samir Genaim
CLIP, Technical University of Madrid, Spain
samir@clip.dia.fi.upm.es

Miguel Gómez-Zamalloa
DSIC, Complutense University of Madrid, Spain
mzamalloa@fdi.ucm.es

Abstract

This article presents a heap space analysis for (sequential) Java bytecode. The analysis generates *heap space cost relations* which define at compile-time the heap consumption of a program as a function of its data size. These relations can be used to obtain upper bounds on the heap space allocated during the execution of the different methods. In addition, we describe how to refine the cost relations, by relying on *escape analysis*, in order to take into account the heap space that can be safely deallocated by the garbage collector upon exit from a corresponding method. These refined cost relations are then used to infer upper bounds on the *active heap space* upon methods return. Example applications for the analysis consider inference of constant heap usage and heap usage proportional to the data size (including polynomial and exponential heap consumption). Our prototype implementation is reported and demonstrated by means of a series of examples which illustrate how the analysis naturally encompasses standard data-structures like lists, trees and arrays with several dimensions written in object-oriented programming style.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D3.2 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

Keywords Heap Space Analysis, Heap Consumption, Low-level Languages, Java Bytecode

1. Introduction

Heap space analysis aims at inferring *bounds* on the heap space consumption of programs. Heap analysis is more typically formulated at the source level (see, e.g., [24, 17, 25, 19] in the context of functional programming and [18, 13] for high-level imperative programming languages). However, there are situations where one has only access to compiled code and not to the source code. An example of this is *mobile code*, where the code consumer receives code to be executed. In this context, Java bytecode [20] is widely used, mainly due to its security features and the fact that it is platform-independent. Automatic heap space analysis has interesting applications in this context. For instance, *resource bound certification* [14, 4, 5, 16, 12] proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on the resource consumption. Also, heap bounds are useful on embedded systems, e.g., smart cards in which memory is limited and cannot easily be recovered. A general framework for the cost analysis of sequential Java bytecode has been proposed in [2]. Such analysis statically generates *cost relations* which define the cost of a program as a function of its input data size. The cost relations are expressed by means of *recursive equations* generated by abstracting the recursive structure of the program and by inferring size relations between arguments. Cost relations are parametric w.r.t. a *cost model*, i.e., the cost unit associated to the bytecode b appears as an abstract value T_b within the equations.

This article develops a novel application of the cost analysis framework of [2] to infer bounds on the heap space consumption of sequential Java bytecode programs. In a first step, we develop a cost model that defines the cost of memory allocation instructions (e.g., `new` and `newarray`) in terms of the number of heap (memory) units it consumes. E.g., the cost of creating a new object is the number of heap units allocated to that object. The remaining bytecode instructions do not add any cost. With this cost model, we generate heap space cost relations which are then used to infer upper bounds on the heap space usage of the different methods. These upper bounds provide information on the maximal heap space required for executing each method in the program. In a second step, we refine this cost model to consider the effect of garbage collection. This is done by relying on escape analysis [15, 8] to identify those memory allocation instructions which create objects that will be

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.

garbage collected upon exit from the corresponding method. With this information available, we can generate heap space cost relations which contain annotations for the heap space that will be garbage collected. The annotated cost relations in turn are used to infer upper bounds on the *active heap space* upon exit from methods, i.e., the heap space consumed and that might not be garbage collected upon exit.

A distinguishing feature of the approach presented in this article w.r.t. previous type-based approaches (e.g., [5, 17]) is that it is not restricted to linear bounds since the generated cost relations can in principle capture any complexity class. Moreover, in many cases, the relations can be simplified to a *closed form* solution from which one can glean immediate information about the expected consumption of the code to be run. The approach has been assessed by means of a prototype implementation, which originates from the one of [3]. It should be noted that the examples in [3] are simple imperative algorithms which did not make use of the heap, since they were aimed at demonstrating that traditional complexity schemata can be handled by the cost analysis of [2]. In contrast, we demonstrate our heap analysis by means of a series of example applications written in an object-oriented style which make intensive use of the heap and which present novel features like heap consumption that depends on the class fields, multiple inheritance, virtual invocation, etc. These examples allow us to illustrate the most salient features of our analysis: inference of constant heap usage, heap usage proportional to input size, support of standard data-structures like lists, trees, arrays, etc. To the best of our knowledge, this is the first analysis able to infer arbitrary heap usage bounds for Java bytecode.

The rest of the paper is structured as follows: Sec. 2 presents an example that illustrates the ideas behind the analysis. Sec. 3 briefly describes the Java bytecode language. Sec. 4 defines a cost model for heap consumption and describes the analysis framework. Sec. 5 demonstrates the different features of the analysis by means of examples. In Sec.6, we extend our cost model to consider the effect of garbage collection. Sec. 7 reports on a prototype implementation and some experimental results. Finally, Sec. 8 concludes and discusses the related work.

2. Worked Example

Consider the Java classes and their corresponding (structured) Java bytecode depicted in Fig. 1 which define a linked-list data structure in an object-oriented style, as it appears in [18]. The class *Cons* is used for data nodes and the class *Nil* plays the role of *null* to indicate the end of a list. Both classes define a copy function which is used to clone the corresponding object. In the case of *Nil* the copy method just returns *this* since it is the last element of the list, and in the case of *Cons* it clones the current object and its successors recursively (by calling the *copy* method of *next*). The rest of this section describes the different steps applied by the analyzer to approximate the heap consumption of the program depicted in Fig. 1. Note that the Java program is provided here just for clarity, the analyzer works directly on the bytecode which is obtained, for example, by compiling the Java program.

Step I: In the first step, the analyzer recovers the structure of the Java bytecode program by building a control flow graph (CFG) for its methods. The CFG consists of basic blocks which contain a sequence of non-branching bytecode instructions, these blocks are connected by edges that describe the possible flows that originate from the branching instructions like conditional jumps, exceptions, virtual method invocation, etc. In Fig. 1, the CFG of the method *Nil.copy* consists of the single block $Block_0^{Nil}$ and the CFG of the method *Cons.copy* consists of the rest of the blocks. $Block_0^{Cons}$ corresponds to the bytecode of *Cons.copy* up to the recursive method call *this.next.copy()*. Then, depending on the type of the object stored in *this.next* the execution is transferred to either *Nil.copy* or *Cons.copy*. This is expressed by the (guarded) branching to $Block_1^{Cons}$ and $Block_2^{Cons}$. In both cases, the control returns to $Block_3^{Cons}$ which corresponds to the rest of the statements.

Step II: In the second step, the analyzer builds an intermediate representation for the CFG and uses it to infer information about the changes in the sizes of the different data-structures (or in the values of integer variables) when the control passes from one part of the program (e.g., a block or a method) to another part. For example, this step infers that when *Nil.copy* or *Cons.copy* are called recursively, the length of the list decreases by one. This information is essential for defining the heap consumption of one part of the program in terms of the heap consumption of other parts.

Step III: In the third step, the intermediate representation and the size information are used together with the cost model for heap consumption to generate a set of cost relations which describe the heap consumption behaviour of the program. The following equations are the ones we get for the example in Fig. 1:

Heap Space Cost Equations			Size relations
$C_{copy}^{Nil}(a)$	=	0	$\{a{=}1\}$
$C_{copy}^{Cons}(a)$	=	$C_0(a)$	
$C_0(a)$	=	$size(Cons) + CC_0(a,b)$	$\{a{\geq}1, b{\geq}0, a{=}b{+}1\}$
$CC_0(a,b)$	=	$\begin{cases} C_1(a,b) & \hat{b} \in Nil \\ C_2(a,b) & \hat{b} \in Cons \end{cases}$	
$C_1(a,b)$	=	$C_{copy}^{Nil}(b) + C_3(a)$	$\{a{=}1\}$
$C_2(a,b)$	=	$C_{copy}^{Cons}(b) + C_3(a)$	$\{a{\geq}2\}$
$C_3(a)$	=	0	

Each of these equations corresponds to a method entry, block or branching in the CFG. An equation is composed by the left hand side which indicates the block or the method it represents, and the right hand side which defines its heap consumption behaviour. In addition, size relations might be attached to describe how the data size changes when using another equation.

```
abstract class List {
    abstract List copy();
}
class Nil extends List {
    List copy() {
        return this;
    }
}
class Cons extends List {
    int elem;
    List next;
    List copy(){
        Cons aux = new Cons();
        aux.elem = this.elem;
        aux.next = this.next.copy();
        return aux;
    }
}
```

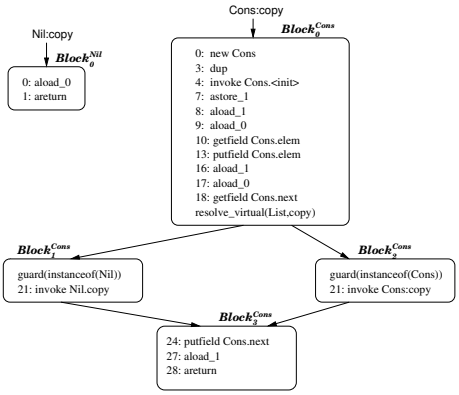


Figure 1. Java source code and CFG bytecode of example

The equation $C^{Nil}_{copy}(a)$ defines the heap consumption of *Nil.copy* in terms of (the size of) its first argument a which corresponds to its *this* reference variable (in Java bytecode the *this* reference variable is the first argument of the method). In this case the heap consumption is zero since the method does not allocate any heap space. The equation $C^{Cons}_{copy}(a)$ defines the heap consumption of *Cons.copy* as the heap consumption of $Block^{Cons}_0$ using the corresponding equation C_0, which in turn defines the heap consumption as the amount of heap units allocated by the *new* bytecode instructions, namely $size(Cons)$, plus the heap consumption of its successors which is defined by the equation CC_0. All other instructions in $Block^{Cons}_0$ contribute zero to the heap consumption. Note that in C_0, the variable b corresponds to *this.next* of *Cons.copy* and that the size analysis is able to infer the relation $a{=}b{+}1$ (i.e., the list a is longer than b by one). The equation CC_0 corresponds to the heap consumption of the branches at the end of $Block^{Cons}_0$, depending on the type of b (denoted as $\hat{b}$) it is equal to the heap consumption of $Block^{Cons}_1$ or $Block^{Cons}_2$ which are respectively defined by the equations C_1 and C_2. The equation C_1 defines the heap consumption of $Block^{Cons}_1$ as the heap consumption of *Nil.copy* (since it is called in $Block^{Cons}_1$) plus the heap consumption of $Block^{Cons}_3$ (using the equation C_3). Similarly C_2 defines the heap consumption of $Block^{Cons}_2$ in terms of the heap consumption of *Cons.copy*. The equation C_3 defines the heap consumption of $Block^{Cons}_3$ to be zero since it does not allocate any heap space.

Step IV: In the fourth step, we can simplify the equations and try to obtain an upper bound in closed form for the cost relation by applying the method described in [1]. In particular, assuming that $size(Cons)$ equals 8 (4 bytes for the integer field *data* and 4 bytes for the reference field *next*), we obtain the following simplified equations:

Equation			Size relations
$C^{Nil}_{copy}(a)$	=	0	$\{a{=}1\}$
$C^{Cons}_{copy}(a)$	=	8	$\{a{=}2\}$
$C^{Cons}_{copy}(a)$	=	$8 + C^{Cons}_{copy}(b)$	$\{a{\geq}3, b{\geq}1, a{=}b{+}1\}$

and then obtain an upper bound in closed form $C^{Cons}_{copy}(a) = 8 * (a-1)$.

The main focus of this paper is on the generation of heap space cost relations, as illustrated in Step III. Steps I and II are done as it is proposed in [2] and Step IV as it is described in [1] and hence we will not give many details on how they are performed in this paper.

3. The Java Bytecode Language

Java bytecode [20] is a low-level object-oriented programming language with unstructured control and an *operand stack* to hold intermediate computational results. Moreover, objects are stored in dynamic memory: the *heap*. A Java bytecode program consists of a set of *class files*, one for each class or interface. A class file contains information about its *name* $c \in Class\_Name$, the class it extends, the interfaces it implements, and the fields and methods it defines. In particular, for each method, the class file contains: a method signature which consists of its name and its type; its bytecode $bc_m = \langle pc_0{:}b_0, \ldots, pc_{n_m}{:}b_{n_m}\rangle$, where each b_i is a *bytecode instruction* and pc_i is its address; and the method's exceptions table. In this work we consider a subset of the JVM [20] language which is able to handle operations on integers and references, object creation and manipulation (by accessing fields and calling methods), arrays of primitive and reference types, and exceptions (either generated by abnormal execution or explicitly thrown by the program). For simplicity, we omit static fields and initializers and primitive types different from integers. Such features could be handled by making the underlying abstract interpretation support them by assuming the worst case approximation for them. Thus, our bytecode instruction set ($bcInst$) is:

$bcInst$ ::=
push x | istore v | astore v | iload v | aload v | iconst a
| iadd | isub | imul | idiv | if$\diamond$ pc | goto pc | ireturn | areturn
| return | new $Class\_Name$ |
| newarray int | anewarray $Class\_Name$ | iaload | aaload
| iastore | aastore | athrow | dup
| invokevirtual/invokespecial $Class\_Name.Meth\_Sig$
| getfield/putfield $Class\_Name.Field\_Sig$

where $\diamond$ is a comparison operator (ne,le,_icmpgt, etc.), v a local variable, a an integer, pc an instruction address, and x an integer or the special value `null`.

4. The Heap Space Analysis Framework

Cost analysis of a low-level object-oriented language such as Java bytecode is complicated mainly due to its unstructured control flow (e.g., the use of goto statements rather than recursive structures), its object-oriented features (e.g., virtual method invocation) and its stack-based model. The recent work of [2] develops a generic framework for the automatic cost analysis of Java bytecode programs. Essentially, the complications of dealing with a low-level language are handled in this framework by abstracting the *recursive structure* of the program and by inferring *size relations* between arguments. As we have seen in Sect. 2, this analysis framework is based on transforming the Java bytecode program to an intermediate representation which fits inside the same setting all possible forms of loops. Then, using this intermediate representation, the analysis infers information about the change in the sizes of the relevant data structures as the program goes through its loops (Steps I and II). Finally, this information is used to set up a cost relation which defines the cost of the program in terms of the sizes of the corresponding data structures.

In this section, we present a novel application of this generic cost analysis framework to infer bounds on the heap space consumption of sequential Java bytecode programs. So far, this framework has been only used in [3] to infer the complexity of some classical algorithms while in this paper our purpose is completely different: we aim at computing bounds on the heap usage for programs written in object-oriented programming style which make intensive use of the heap. In Sect. 4.1 and Sect. 4.2, we briefly present the notions of recursive representation and calls-to size-relation in a rather informal style. Then, we introduce our cost model for heap consumption and our notion of heap space cost relation in Sect. 4.3.

4.1 Recursive Representation

Cost relations can be elegantly expressed as systems of *recursive* equations. In order to automatically generate them, we need to capture the iterative behaviour of the program by means of recursion. One way of achieving this is by computing the CFG of the program. Also, advanced features like virtual invocation and exceptions are simply dealt as additional nodes in the graph. To analyze the bytecode, its CFG can be represented by using some auxiliary recursive representation (see, e.g., [2]). In this approach, a *bytecode* is transformed into a set of *guarded rules* of the form $\langle head \leftarrow guard, body\rangle$ where the *guard* states the applicability conditions for the rule. Rules are obtained from blocks in the CFG and *guards* indicate the conditions under which each block is executed. As it is customary in deterministic imperative languages, guards provide mutually exclusive conditions because paths from a block are always exclusive (i.e., alternative) choices.

DEFINITION 4.1 (rec. representation). *Consider a block* $\mathtt{p}$ *in a CFG, which contains a sequence of bytecode instructions* $\mathtt{B}$ *guarded by the condition* $\mathtt{G_b}$ *and whose successor blocks are* $\mathtt{q_1}, \cdots, \mathtt{q_n}$. *The* recursive representation *of* $\mathtt{p}$ *is:*

$$\mathtt{p}(\bar{\mathtt{l}}, \bar{\mathtt{s}}, \mathtt{r}) \leftarrow \mathtt{G_p}, \mathtt{B}, (\mathtt{q_1}(\bar{\mathtt{l}}, \bar{\mathtt{s}}', \mathtt{r}); \cdots ; \mathtt{q_n}(\bar{\mathtt{l}}, \bar{\mathtt{s}}', \mathtt{r}))$$

where:

- $\bar{l}$ *is a tuple of variables which corresponds to the method's local variables,*
- $\bar{s}$ *and* $\bar{s}'$ *are tuples of variables which respectively correspond to the active stack elements at the block's entry and exit,*
- r *is a single variable which corresponds to the method's return value (omitted if there is not return value),*
- G_p *and* B *are obtained from the block's guard and bytecode instructions by adding the local variables and stack elements on which they operate as explicit arguments.*

We denote by $\mathtt{calls}(\mathtt{B})$ *the set of method invocation instructions within* $\mathtt{B}$ *and by* $\mathtt{bytecode}(\mathtt{B})$ *the other instructions.* □

The formal translation of bytecode instructions in `B` to calls within the recursive rules is presented in [2]. In this translation, it is interesting to note that the stack positions are visible in the rules by explicitly defining them as local variables. This intermediate representation is convenient for analysis as in one pass we can eliminate almost all stack variables which results in a more efficient analysis.

EXAMPLE 4.2. *The rules that correspond to the blocks* $Block_0^{Cons}$, $Block_1^{Cons}$ *and* $Block_2^{Cons}$ *in Fig. 1 are:*

$$\begin{array}{l}
\mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{r}) \leftarrow \\
\quad \mathtt{new}(\mathtt{Cons}, \mathtt{s}_0), \mathtt{dup}(\mathtt{s}_0, \mathtt{s}_1), \mathtt{Cons.<init>}(\mathtt{s}_1), \\
\quad \mathtt{astore}(\mathtt{s}_0, \mathtt{aux}'), \mathtt{aload}(\mathtt{aux}', \mathtt{s}_0'), \mathtt{aload}(\mathtt{this}, \mathtt{s}_1'), \\
\quad \mathtt{getfield}(\mathtt{Cons.elem}, \mathtt{s}_1', \mathtt{s}_1''), \\
\quad \mathtt{putfield}(\mathtt{Cons.elem}, \mathtt{s}_0', \mathtt{s}_1''), \\
\quad \mathtt{aload}(\mathtt{aux}', \mathtt{s}_0''), \mathtt{aload}(\mathtt{this}, \mathtt{s}_1'''), \\
\quad \mathtt{getfield}(\mathtt{Cons.next}, \mathtt{s}_1''''), \\
\quad (\mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}) ; \\
\qquad \mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r})).
\end{array}$$

$$\begin{array}{l}
\mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1, \mathtt{r}) \leftarrow \\
\quad \mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_1, \mathtt{Nil})), \\
\quad \mathtt{Nil.copy}(\mathtt{s}_1, \mathtt{s}_1'), \\
\quad \mathtt{copy}_3^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1', \mathtt{r}).
\end{array}$$

$$\begin{array}{l}
\mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1, \mathtt{r}) \leftarrow \\
\quad \mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_1, \mathtt{Cons})), \\
\quad \mathtt{Cons.copy}(\mathtt{s}_1, \mathtt{s}_1'), \\
\quad \mathtt{copy}_3^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1', \mathtt{r}).
\end{array}$$

The rule $\mathtt{copy}_0^{\mathtt{Cons}}$ *is not guarded and has two continuation blocks, while the other rules are guarded by the type of*

the object of s_1 (the top of the stack) and have only one successor. The bytecode instructions were transformed to include explicitly the stacks elements and the local variables on which they operate, moreover, all variables are in single static assignment *form. Note that calls to methods take the same form as calls to blocks, which makes all different forms of loops to fit in the same setting.* □

4.2 Size Analysis

A size analysis is then performed on the recursive representation in order to infer the *calls-to size-relations* between the variables in the head of the rule and the variables used in the calls (to rules) which occur in the body for each program rule. Derivation of constraints is a standard abstract interpretation over a constraints domain such as Polyhedra [2, 3]. Such relations are essential for defining the cost of one block in terms of the cost of its successors. The analysis is done by abstracting the bytecode instructions into the linear constraints they impose on their arguments, and then computing a fixpoint that collects *calls-to* relations.

DEFINITION 4.3 (calls-to size-relations). *Consider the rule in Def. 4.1, its calls-to* size-relations *are triples of the form*

$\langle \mathtt{p}(\bar{\mathtt{x}}), \mathtt{p}'(\bar{\mathtt{z}}), \varphi \rangle$ *where* $\mathtt{p}'(\bar{\mathtt{z}}) \in \mathtt{calls(B)} \cup \mathtt{q_1}(\bar{\mathtt{y}}) \cup \ldots \cup \mathtt{q_n}(\bar{\mathtt{y}})$

The size-relation φ *is given as a conjunction of linear constraints. The tuples of variables $\bar{x}$, $\bar{y}$ and $\bar{z}$ correspond to the variables of the corresponding block.* □

In Java bytecode, we consider three cases within size relations: for integer variables, *size-relations* are constraints on the possible values of variables; for reference variables, they are constraints on the length of the longest reachable paths [21], and for arrays they are constraints on the length of the array. Note that using the path-length notion cyclic structures are not handled since to guarantee soundness the corresponding references are abstracted to "unknown-length" and therefore cost that depends on them cannot be inferred.

EXAMPLE 4.4. *The calls-to-size relation for the first rule in Ex. 4.2 is formed by the triples:*

$\langle \mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}), \mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}), \varphi \rangle$
$\langle \mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}), \mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}), \varphi \rangle$

where φ includes, among others, the constraint $this{=}s_1''''{+}1$ which states that the list that $this$ points to is longer by one than the list that s_1'''' points to (s_1'''' corresponds to $this.next$). The meaning of the above relations is explained in Section 2. Note that the call to the constructor Cons.<init> *is ignored for simplicity.* □

4.3 Heap Space Cost Relations

In order to define our heap space cost analysis, we start by defining a cost model which defines the cost of memory allocation instructions (e.g., new, newarray and anewarray) as the the number of heap (memory) units they consume. The remaining bytecode instructions do not add any cost.

DEFINITION 4.5 (cost model for heap space). *We define a cost model $\mathcal{M}_{heap}$ which takes a bytecode instruction* bc *and returns a positive expression as follows:*

$$\mathcal{M}_{heap}(\mathtt{bc}) = \begin{cases} size(\mathtt{Class}) & \text{if } bc{=}\mathtt{new(Class,\_)} \\ S_{\mathtt{PrimType}} * \mathtt{L} & \text{if } bc{=}\mathtt{newarray(PrimType,L,\_)} \\ S_{\mathtt{ref}} * \mathtt{L} & \text{if } bc{=}\mathtt{anewarray(Class,L,\_)} \\ 0 & \text{otherwise} \end{cases}$$

where $S_{\mathtt{PrimType}}$ and $S_{\mathtt{ref}}$ denote, respectively, the heap consumption of primitive types and references. Function size is defined as follows:

$$size(O) = \begin{cases} \sum_{F \in \mathtt{Class.field}} size(type(F)) & \text{if } O{=}\mathtt{Class} \\ S_{\mathtt{PrimType}} & \text{if } O \text{ is a primitive type} \\ S_{\mathtt{ref}} & \text{if } O \text{ is a reference type} \end{cases}$$

where the type of a field in a Class *(i.e.,* Class.field*) can be either primitive or reference.* □

In Java bytecode, types are classified into primitive (its size is represented by $S_{\mathtt{PrimType}}$ in our model) and reference types ($S_{\mathtt{ref}}$). In a particular assessment, one has to set the concrete values for $S_{\mathtt{PrimType}}$ and $S_{\mathtt{ref}}$ of the JVM implementation.

For each rule in the recursive representation of the program and its corresponding size relation, the analysis generates the cost equations which define the heap consumption of executing the block (or possibly a method call) by relying on the above cost model. A *heap space cost relation* is defined as the set of cost equations for each block of the bytecode (or rule in the recursive representation).

DEFINITION 4.6 (heap space cost relation). *Consider a rule $\mathcal{R}$ of the form $\mathtt{p}(\bar{\mathtt{x}}) \leftarrow \mathtt{G_p}, \mathtt{B}, (\mathtt{q_1}(\bar{\mathtt{y}}); \cdots; \mathtt{q_n}(\bar{\mathtt{y}}))$ and let the linear constraints φ be a conjunction of all call-to size-relations within the rule. The* heap space cost *equations for $\mathcal{R}$ are generated as follows:*

$$C_p(\bar{x}) = \sum_{b \in bytecode(B)} \mathcal{M}_{heap}(b) + \sum_{r(\bar{z}) \in calls(B)} + C_r(\bar{z}) + C_{p\_cont}(\bar{y}) \quad \varphi$$
$$C_{p\_cont}(\bar{y}) = C_{q_1}(\bar{y}) \quad G_{q_1}$$
$$\ldots$$
$$C_{p\_cont}(\bar{y}) = C_{q_n}(\bar{y}) \quad G_{q_n}$$

where $\mathtt{G_{q_i}}$ is the guard of $\mathtt{q_i}$. The heap space cost relation associated to the recursive representation of a method is defined as the set of cost equations for its blocks. □

When the rule has multiple *continuation*s, it is transformed into several equations. We specify the cost of each continuation in a separate equation because the guards for determining the alternative path $\mathtt{q_i}$ that the execution will take (with $i = 1, \ldots, n$) are only known at the end of the execution of the bytecode B; thus, they cannot be evaluated before B is executed. The guards appear also decorating the equations. In the implementation, when a rule has only one continuation, it gives rise to a single equation which contains the size relation φ as an attachment.

Java source code

```
abstract class Data{
  abstract public Data copy();
}
class Polynomial extends Data{
  private int deg;
  private int[] coefs;
  public Polynomial() {
    coefs = new int[11];}
  public Data copy() {
    Polynomial aux = new Polynomial();
    aux.deg = deg;
    for (int i=0;i<=deg && i<=10;i++)
      aux.coefs[i] = coefs[i];
    return aux;}}
```

```
class Vector3D extends Data{
  private int x;
  private int y;
  private int z;
  public Vector3D(int x,int y,int z) {
    this.x = x;
    this.y = y;
    this.z = z;}
  public Data copy() {
    return new Vector3D(x,y,z);
  }
}
```

```
class Results{
  Data[] rs;
  public Results() {
    rs = new Data[25];
  }
```

```
public Results copy() {
  Results aux = new Results();
  for (int i = 0;i < 25;i++)
    aux.rs[i] = rs[i].copy();
  return aux;}}
```

Heap space cost equations

Equation	Guard	Size rels.
$C_{copy}(a) = S_{ref} + 25{*}S_{ref} + C_0(a,0)$		
$C_0(a,i) = \underbrace{S_{int} + S_{ref}}_{size(Polyn)} + 11{*}S_{int} + C_0(a,j)$	$\langle \hat{a}.rs[i] \in Polyn \rangle$	$\{i{<}25,\ j = i{+}1\}$
$C_0(a,i) = \underbrace{3{*}S_{int}}_{size(Vect3D)} + C_0(a,j)$	$\langle \hat{a}.rs[i] \in Vect3D \rangle$	$\{i{<}25,\ j = i{+}1\}$
$C_0(a,i) = 0$		$\{i{>}{=}25\}$

Figure 2. Constant heap space example

EXAMPLE 4.7. *The heap space cost equations generated for the rule* $copy_0^{Cons}$ *of Ex. 4.2 and the size relation of Ex. 4.4 are (see Sect. 2):*

$$C_0^{Cons}(this, aux) = size(Cons) + CC_0^{Cons}(this, aux', s_0'', s_1'''')\{this = s_1'''' + 1, \ldots\}$$

$$CC_0^{Cons}(this, aux', s_0'', s_1'''') = \begin{cases} C_1^{Cons}(this, aux', s_0'', s_1'''') & \hat{s}_1'''' \in Nil \\ C_2^{Cons}(this, aux', s_0'', s_1'''') & \hat{s}_1'''' \in Cons \end{cases}$$

The cost of $Block_0^{Cons}$ *is captured by* C_0^{Cons}*, among all bytecode instructions in* $Block_0^{Cons}$*, we count only the creation of the object of class* `Cons`*. The continuation of* $Block_0^{Cons}$ *is captured in the relation* CC_0^{Cons}*, where depending on the type of the object* s_1''''*, we choose between two* mutually exclusive *equations* C_1^{Cons} *or* C_2^{Cons}*.* □

In addition, the analyzer performs a slicing step, which aims at removing variables that do not affect the cost. And also tries to simplify the equations as much as possible by applying unfolding steps. These steps lead to simpler cost relations. Due to lack of space, during the rest of the paper we will apply them without giving details on how they were performed.

5. Example Applications of Heap Space Analysis

In this section, we show the most salient features of our heap space analysis by means of a series of examples. All examples are written in object-oriented style and make intensive use of the heap. We intend to illustrate how our analysis is able to deal with standard data-structures like lists, trees and arrays with several dimensions as well as with multiple inheritance, class fields, virtual invocation, etc. We show examples which present heap usage which depends proportionally to the *data size*, namely in some cases it depends on class fields while in another one on the input arguments. An interesting point is that heap consumption is, in the different examples, constant, linear, polynomial or exponentially proportional to the data sizes.

For each example, we show the Java source code and its heap space cost relation. Each relation consists of three parts: the equations, the guards and the size relations. The applicability conditions of each equation are defined by the guards and the size relations. Guards usually provide non-numeric conditions while size relations provide conditions on the sizes of the corresponding variables. In addition, size relations describe how the data changes when the control moves from one to another part of the program. Since our

Java source code

```
abstract class List{
  abstract public List copy();
}
class Nil extends List{
  public List copy() {
    return this;
}
```

```
class Cons extends List
  private Data elem;
  private List next;
  public List copy() {
    Cons aux = new Cons();
    aux.elem = this.elem.copy();
    aux.next = this.next.copy();
    return aux;}
```

Heap space cost equations

Equation	Guard	Size rels.
$C_{copy}(a) = \overbrace{2{*}S_{ref}}^{size(Cons)} + \overbrace{S_{int}+S_{ref}+11{*}S_{int}}^{this.elem.copy()}$	$\left\langle \begin{array}{l} \hat{a}.elem \in Polyn \wedge \\ \hat{a}.next \in Nil \end{array} \right\rangle$	$\{a=2\}$
$C_{copy}(a) = 2{*}S_{ref}+S_{int}+S_{ref} +11{*}S_{int}+C_{copy}(b)$	$\left\langle \begin{array}{l} \hat{a}.elem \in Polyn \wedge \\ \hat{a}.next \in Cons \end{array} \right\rangle$	$\left\{ \begin{array}{l} a{\geq}3, b{\geq}2 \\ a{>}b \end{array} \right\}$
$C_{copy}(a) = \underbrace{2{*}S_{ref}}_{size(Cons)} + \underbrace{3{*}S_{int}}_{this.elem.copy()}$	$\left\langle \begin{array}{l} \hat{a}.elem \in Vect3D \wedge \\ \hat{a}.next \in Nil \end{array} \right\rangle$	$\{a=2\}$
$C_{copy}(a) = 2{*}S_{ref}+3{*}S_{int}+C_{copy}(b)$	$\left\langle \begin{array}{l} \hat{a}.elem \in Vect3D \wedge \\ \hat{a}.next \in Cons \end{array} \right\rangle$	$\left\{ \begin{array}{l} a{\geq}3, b{\geq}2 \\ a{>}b \end{array} \right\}$

Figure 3. Generic list example

system only deals with integer primitive types, we use the cost model presented in Sect. 4.3 with the constants S_{int} and S_{ref} to denote the basic sizes for integers and reference types, respectively. Also note that we provide the Java source code instead of the bytecode just for clarity and space limitations. The analyzer works directly on the bytecode which can be found in the appendix.

5.1 Constant Heap Space Usage

In the first example we consider a method with constant heap space usage, i.e., its heap consumption does not depend on any input argument. Fig. 2 shows both the source code and the heap space cost equations generated by the analyzer. The program implements a data hierarchy which will be used throughout the section. It consists of an abstract class, *Data* and two subclasses, *Polynomial* and *Vector3D*. The class *Polynomial* defines a polynomial expression of degree up to 10 with integer coefficients, the coefficients are stored in the array field *coefs* and the degree in the integer field *deg*. Its *copy* method returns a deep copy of the corresponding polynomial by creating a new array of 11 integers and copying the first $deg{+}1$ original coefficients. The class *Vector3D* represents an integer vector with 3 dimensions. The class *Results* stores 25 objects of type *Data*, which in execution time will be *Polynomial* or *Vector3D* objects. Its *copy* method produces a deep copy of the whole structure where each of the 25 elements is copied by its corresponding *copy* method (hence dynamically resolved).

The cost equations generated by the analyzer for the method *Results.copy* are shown in Fig. 2 (at the bottom left). The first equation $C_{copy}(a)$ defines the heap consumption of the method in terms of its first argument a which corresponds to the abstraction of its *this* reference variable (i.e., its size). It counts the heap space allocated for the creation of an object of type *Results*, namely S_{ref}; the space allocated by its constructor, namely $25{*}S_{ref}$; and the space allocated when executing the loop. The heap space allocated by the loop is captured by C_0 and it depends on the type of the object at the current position of the array (which is specified in the guards by checking the class of $\hat{a}.rs[i]$) such that the call to its corresponding *copy* method contributes $S_{int}+S_{ref}{+}11{*}S_{int}$ if it is an instance of *Polynomial* and $3{*}S_{int}$ if it is an instance of *Vector3D*.

As already mentioned, a further issue is how to automatically infer *closed form* solutions (i.e., without recurrences) from the generated cost relations. In our examples, we can directly apply the method of [1] to compute an upper bound in closed form. However, we will not go into details of this process as it is not a concern of this paper and we will simply show the asymptotic complexity that can be directly obtained from such upper bounds. We can observe from the equations that the asymptotic complexity is $O(1)$, as equation C_{copy} is a constant plus C_0, and C_0 is called a constant number of times (in this case 25 times). By assuming that $S_{int}=4$ and $S_{ref}=4$, we can obtain the following upper bound $C_{copy}=4+25*4+25*52=1404$.

5.2 Bounds Proportional to the Input Data Size

For the second example, we consider a generic data structure of type *List*. Both the source code and the heap space cost equations obtained by our analyzer are depicted in Fig. 3. The list is implemented taking advantage of the polymorphism as in the style of the example in Sect. 1, but in this case the elements of the list are objects extending from *Data*

Java source code

```
class Score{
  private int gt1, gt2;
  public Score() {
    gt1 = 0;
    gt2 = 0;
  }
class Scoreboard{
  private Score[][][] scores;
public Scoreboard(int a,int b) {
  scores = new Score[a][][];
  for (int i = 1;i <= a;i++) {
    scores[i-1] = new Score[i][];
    for (int j = 0;j < (i-1);j++) {
      scores[i-1][j] = new Score[b];
      for (int k = 0;k < b;k++)
        scores[i-1][j][k] = new Score();}}}}
```

Heap space cost equations

Equation			Guard	Size rels.
$C_{<init>}(a,b)$	=	$a{*}S_{ref} + C_1(a,b,1)$		
$C_1(a,b,i)$	=	$i{*}S_{ref} + C_2(b,i,0) + C_1(a,b,d)$		$\{i{\leq}a,\ d{=}i{+}1\}$
$C_1(a,b,i)$	=	0		$\{i > a\}$
$C_2(b,i,j)$	=	$b{*}S_{ref} + C_3(b,0) + C_2(b,i,d)$		$\{j{<}(i{-}1),\ d{=}j{+}1\}$
$C_2(b,i,j)$	=	0		$\{j \geq (i{-}1)\}$
$C_3(b,k)$	=	$2{*}S_{int} + C_3(b,c)$		$\{k{<}b,\ c{=}k{+}1\}$
$C_3(b,k)$	=	0		$\{k \geq b\}$

Figure 4. Multi-dimensional arrays example

(see the classes in Fig. 2) rather than integer primitive types. The *List.copy* method returns a deep copy of the list which, in addition to copying the whole list structure, it copies each element by using the corresponding *Data.copy* method (resolved at execution time).

At the bottom of Fig. 3, we show the heap space cost equations our analyzer generates for the method *List.copy* of class *Cons*. The equation $C_{copy}(a)$ defines the heap consumption of the whole method in terms of its first argument a which represents the size of its *this* reference variable. There are four equations for C_{copy}, two of them (the second and the fourth one) are recursive and correspond to the case in which the rest of the list is not empty, i.e., $\hat{a}.next \in Cons$. Note that, in such recursive equations, the size analysis is able to infer the constraint $a > b$, thus ensuring that recursive calls are made with a strictly decreasing value. The other two equations are constant and correspond to the base case (i.e. the rest of the list is empty). This is abstracted in the size relations with the constraint $a = 2$. Note that the heap usage depends on whether we invoke the *copy* method of a *Polynomial* or a *Vector3D* object. By considering the worst cases for all equations, we can infer the upper bound $C_{copy}(a){\leq}(5*S_{ref}+15*S_{int})*a \equiv O(a)$ which describes a heap consumption linear in a, the size of the list.

5.3 Multi-Dimensional Arrays

Let us consider the example in Fig. 4. The class *Scoreboard* is instrumental to show how our heap space analysis deals with complex multi-dimensional array creation. The class has a 3-dimensional array field. The constructor takes two integers a and b and creates an array such that: the first dimension is a; the second dimension ranges from 1 to a; and the third dimension is b. Each array entry $scores[i][j][k]$ stores an object of type *Score*.

At the bottom of Fig. 4 we can see the heap space cost equations generated by the analyzer for the constructor of class *Scoreboard*. The equation $C_{<init>}(a,b)$ represents the heap space consumption of the constructor where a and b correspond to the size of its input parameters. It counts the heap consumed by constructing the first array dimension, $a{*}S_{ref}$, plus the heap consumption when executing the outermost loop which is represented by the call $C_1(a,b,1)$. The heap consumption modeled by C_1 includes the amount of heap allocated for the second array dimension in each iteration, $i{*}S_{ref}$, and the consumption of executing the middle loop which is represented by the call $C_2(b,i,0)$. Note that size analysis infers that within C_1, the value of i increases by 1 at each iteration ($d{=}i{+}1$) until it converges to a ($i{\leq}a$). The equation C_2 defines the heap consumption of the middle loop, which includes the heap allocated for the third array dimension, $b*S_{ref}$, plus the consumption of executing the innermost loop which is represented by the call $C_3(b,0)$. Finally, C_3 models the heap space required for creating $b{-}k$ *Score* objects by the innermost loop. In this case, we can infer the upper bound $C_{<init>}(a,b) \leq (((2{*}S_{int}{*}b) + b{*}S_{ref}){*}a + a{*}S_{ref}){*}a + a{*}S_{ref} \equiv O(b{*}a^2)$.

5.4 Complex Data Structures

For the last example, let us consider a more complex tree-like data structure which is depicted in Fig. 5. The class *MultiBST* implements a binary search tree data structure where each node has an object of type List (from Fig. 3) and two successors of type *MultiBST* which correspond to the right and left branches of the tree. The constructor method creates an empty tree whose *data* field is initialized to an empty list, i.e., an instance of class *Nil*. The *copy* method performs a deep copy of the whole tree by relying on the *copy* method of class *List*.

Java source code

```
class BST {
  private List data;
  private MultiBST lc;
  private MultiBST rc;
  public MultiBST() {
    data = new Nil();
    lc = null; rc = null; }
```

```
public MultiBST copy() {
  MultiBST aux = new MultiBST();
  aux.data = data.copy();
  if (l==null) aux.lc=null; else aux.lc=lc.copy();
  if (r==null) aux.rc=null; else aux.rc=rc.copy();
  return aux;}}
```

Heap space cost equations

Equation	Guard	Size rels.
$C(a) = 3{*}S_{ref} + D(d)$	$\langle \hat{a}.lc = null, \hat{a}.rc = null \rangle$	$\{a{>}0, a{>}d\}$
$C(a) = 3{*}S_{ref} + D(d) + C(l)$	$\langle \hat{a}.lc \neq null, \hat{a}.rc = null \rangle$	$\{a{>}0, a{>}d, a{>}l\}$
$C(a) = 3{*}S_{ref} + D(d) + C(r)$	$\langle \hat{a}.lc = null, \hat{a}.rc \neq null \rangle$	$\{a{>}0, a{>}d, a{>}r\}$
$C(a) = 3{*}S_{ref} + D(d) + C(l) + C(r)$	$\langle \hat{a}.lc \neq null, \hat{a}.rc \neq null \rangle$	$\{a{>}0, a{>}d, a{>}l, a{>}r\}$

Figure 5. Multi binary search tree example

The heap space cost equations generated by the analyzer for the method *MultiBST.copy* are depicted in Fig. 5 (at the bottom). The equations defining $C(a)$ represent the heap space usage of the whole method in terms of the parameter a, which corresponds to the maximal path-length in the tree. There are four cases which correspond to the different possible values for the left and right branches (equal or different from null). Consider for example the last equation: $3 * S_{ref}$ is the heap allocated by the new instruction; $D(d)$ is the heap consumption for copying an object of type *List* which corresponds to C_{copy} from Fig. 3; $C(l)$ and $C(r)$ correspond to the heap consumption of copying the left and right branches respectively. From the cost relation, we infer the upper bound $C(a){\leq}(3{*}S_{ref}{+}D(a)){*}2^a$ where $D(a)$ corresponds to the cost of copying the *data* field (see Sec. 5.2).

6. Active Heap Space with Garbage Collection

One of the safety principles in the Java language is ensured by the use of a garbage collector which avoids errors by the programmer related to deallocation of objects from the heap. The aim of this section is to furnish the heap usage cost relations with *safe annotations* which mark the heap space that will be deallocated by the garbage collector upon exit from the corresponding method. The annotations are then used to infer heap space upper bounds for methods upon exit.

In order to generate such annotations, we rely on the use of *escape analysis* (see, e.g., [8, 15]). Essentially, we assume that the heap allocation instructions `new`, `newarray` and `anewarray` have been respectively transformed by new instructions `new_gc`, `newarray_gc` and `anewarray_gc` as long as it is guaranteed that the lifetime of the corresponding allocated heap space does not exceed the instruction's static scope. In this case, the heap space can be safely deallocated upon exit from the corresponding method. This preprocessing transformation can be done in a straightforward way by using the information inferred by escape analysis. In the following, we refer by *transformed* bytecode instructions to the above transformation performed on the heap allocation instructions. Also, we use $gc(H)$ to denote that the heap space H will safely be garbage collected upon exit from the corresponding method (according to escape analysis) and $ngc(H)$ to denote that it might not be garbage collected.

DEFINITION 6.1. *We define a cost model for heap space with garbage collection which takes a* transformed *bytecode instruction* bc *and returns a positive symbolic expression as:*

$$\mathcal{M}^{gc}_{heap}(\mathtt{bc}) = \begin{cases} ngc(size(\mathtt{Class})) & \textit{if bc}{=}\mathtt{new(Class,\_)} \\ gc(size(\mathtt{Class})) & \textit{if bc}{=}\mathtt{new\_gc(Class,\_)} \\ ngc(S_{\mathtt{PrimType}} * \mathtt{L}) & \textit{if bc}{=}\mathtt{newarray(PrimType,L,\_)} \\ gc(S_{\mathtt{PrimType}} * \mathtt{L}) & \textit{if bc}{=}\mathtt{newarray\_gc(PrimType,L,\_)} \\ ngc(S_{\mathtt{ref}} * \mathtt{L}) & \textit{if bc}{=}\mathtt{anewarray(Class,L,\_)} \\ gc(S_{\mathtt{ref}} * \mathtt{L}) & \textit{if bc}{=}\mathtt{anewarray\_gc(Class,L,\_)} \\ 0 & \textit{otherwise} \end{cases}$$

where $S_{\mathtt{PrimType}}$, $S_{\mathtt{ref}}$ *and* $size()$ *are as in Def. 4.5.* □

The above cost model returns a *symbolic* positive expression which contains the annotations gc and ngc as described above. Therefore, when generating the heap space cost relations as described in Def. 4.6 w.r.t. $\mathcal{M}^{gc}_{heap}$, the cost relations will be of the following form:

$$C(\bar{x}) = gc(H_{gc}) + ngc(H_{ngc}) + \sum C_r(\bar{z}) \quad \varphi$$

where we assume that all symbolic expressions wrapped by gc (resp. by ngc) are grouped together within each cost equation and denote the total heap space H_{gc} that will be garbage collected (resp. H_{ngc} which might not be) after the application of such equation.

EXAMPLE 6.2. *Suppose we add the following methods*

```
abstract List map(Func o);                    // List
List map(Func o) { return this; }             // Nil
List map(Func o) {                            // Cons
    List tail = this.next.map(o);
    Cons head = new Cons();
    head.next = tail;
    head.elem = o.f(new Integer(this.elem));
    return head;
}
```

respectively to the classes List, Nil *and* Cons *which are depicted in Fig. 1. The method* map *clones the corresponding list structure, but the value of the field* elem *in the clone is the result of applying the method* o.f *on the corresponding value in the cloned list. Note that the method* o.f*, takes as input an object of type* Integer*, therefore* this.elem *(which is of type* int*) is first converted to* Integer *by creating a temporary corresponding* Integer *object. For simplicity, we do not give a specific definition for* Func*, but we assume that its method f (which is called using* o.f*) does not allocate any heap space and that it returns a value of type* int*. Using* escape analysis*, the creation of the temporary* Integer *object can be annotated as local to* map*, therefore we replace the corresponding* `new` *instruction by* `new_gc`*. Assuming that the size of an* Integer *object is 4 bytes, and using the cost model of Def. 6.1, we obtain the following cost equations:*

Equation	*Size relations*
$C^{Nil}_{map}(a) = 0$	$\{a{=}1\}$
$C^{Cons}_{map}(a) = gc(4) + ngc(8)$	$\{a{=}2\}$
$C^{Cons}_{map}(a) = gc(4) + ngc(8) + C^{Cons}_{map}(b)$	$\{a{\geq}3, b{\geq}1, a{=}b{+}1\}$

The symbolic expression $gc(4)$ in the above equations corresponds to the heap space allocated for the temporary Integer *object which can be garbage collected upon exit from* map*, and $ngc(8)$ corresponds to the heap space allocated for the* Cons *object. As before,* a *corresponds to the size of the* this *reference variable (i.e., the list length) and* b *to* this.next. □

Using the refined cost relations we can infer different information about the heap space usage depending on the interpretation given to the gc and ngc annotations. Let us first consider the following definitions:

$$\boxed{\forall\, H,\, gc(H) = 0 \text{ and } ngc(H) = H} \qquad (1)$$

where we do not count the heap space that will be deallocated upon exit from the corresponding method. By applying Eq. (1) to a cost relation C_m of a method m, we can infer an upper bound U^{gc}_m of the *active heap space* upon the exit from m, i.e., the heap space consumed by m which might not be deallocated upon exit. In this setting, for the cost relations of Ex. 6.2 we infer the closed form $U^{gc}_{map} \equiv C^{Cons}_{map}(a) = 8 * (a-1)$. It is important to note that, in general, such upper bound does not ensure that the heap space required for executing m does not exceed U^{gc}_m, i.e., it is not an upper bound of the heap usage *during* the execution of m but rather only *after* its execution. Actually, in this simple example, we can observe already that during the execution of the method *map*, if all objects are heap allocated, we need more than $8 * (a-1)$ heap units (as the objects of type *Integer* will be heap allocated and they are not accounted in the upper bound). However, one of the applications of escape analysis is to determine which objects can be stack allocated instead of heap allocated in order to avoid invoking the garbage collector which is time consuming [8]. For instance, in the above example, the objects of type *Integer* can be safely stack allocated. When this stack allocation optimization is performed, then U^{gc}_m is indeed an upper bound for the heap space required to execute m.

In order to infer upper bounds for the heap space required during the execution of m, we define gc and ngc as follows:

$$\boxed{\forall\, H,\, gc(H) = H \text{ and } ngc(H) = H} \qquad (2)$$

In this case, we obtain the same cost relations as in Def. 4.6 which correspond to the worst case heap usage in which we do not discount any deallocation by the garbage collector. In this setting, for the cost relation of Ex. 6.2 we infer the closed form $U_{map}(a) \equiv C^{Cons}_{map}(a) = 12 * (a-1)$.

Analysis for finding upper bounds on the memory high-watermark cannot be directly done using cost relations as introducing decrements in the equations requires computing lower bounds. As a further issue, the *active heap space* upper bound, U^{gc}_m, can be used to improve the accuracy of the upper bound on the heap space required for executing a sequence of method calls. For example, an upper bound of the heap space required for executing a method m_1 and upon its return immediately executing a method m_2 can be approximated by $max(U_{m_1}, U^{gc}_{m_1} + U_{m_2})$ which is more precise than taking $U_{m_1} + U_{m_2}$ as it takes into account that after executing m_1 we can apply garbage collection and only then executing m_2. This idea is the basis for a post-processing that could be done on the program in order to obtain more accurate upper bounds on the heap usage at a *program point* level. This is a subject of ongoing research.

7. Experiments

In order to assess the practicality of our heap space analysis, we have implemented a prototype inter-procedural analyzer in `Ciao` [10] as an extension of the one in [3]. We still have not incorporated an escape analysis in our implementation and hence the upper bounds inferred correspond to those generated using Eq. (2) of Sect. 6. The experiments have been performed on an Intel P4 Xeon 2 GHz with 4 GB of RAM, running GNU Linux FC-2, 2.6.9. Table 1 shows the run-times of the different phases of the heap space analysis process. The name of the main class to be analyzed is given in the first column, *Benchmark*, and its size (the sum of all its *class* file sizes) in KBytes is given in the second column, *Size*. Columns 3-6 shows the runtime of the different phases in milliseconds, they are computed as the arithmetic mean of five runs: *RR* is the time for obtaining the recursive representation (building CFG, eliminating stack elements, etc., as outlined in Sec. 4.1); *Size An.* is the time for the abstract-interpretation based size analysis for computing size relations; *Cost* is the time taken for building the heap space cost relations for the different blocks and representing them in a simplified form; and *Total* shows the total times of the whole analysis process. In the last column, *Complexity*, we depict the asymptotic complexity of the (worst-case) heap space cost obtained from the cost relations.

Benchmark	Size	RR	Size An.	Cost	Total	Complexity	
ListInt	0.86	24	53	7	83	$O(n)$	$n \equiv$ list length
Results	1.31	83	275	15	374	$O(1)$	–
BSTInt	0.48	37	113	5	156	$O(2^n)$	$n \equiv$ tree depth
List	1.79	71	207	16	293	$O(n)$	$n \equiv$ list length
Queue	1.93	219	570	24	813	$O(n)$	$n \equiv$ queue length
Stack	1.38	89	643	17	749	$O(n)$	$n \equiv$ stack length
BST	1.43	97	238	14	349	$O(2^n)$	$n \equiv$ tree depth
Scoreboard	0.65	280	1539	12	1830	$O(a^2 * b)$	$\{a, b\} \equiv$ input args.
MultiBST	2.35	166	510	34	709	$O(n * 2^n)$	$n \equiv$ tree depth

Table 1. Measured time (in ms) of the different phases of cost analysis

Regarding the benchmarks we have used, on one hand, we have benchmarks implementing some classic data structures using an object-oriented programming style, which expose the analyzer's ability in handling such classical data structures as well as sophisticated object-oriented programming features. In particular, *ListInt*, *List*, *Queue*, *Stack*, *BSTInt*, *BST* and *MultiBST* implement respectively integer and generic lists, generic queues, generic stacks, integer and generic binary search trees which allow data repetitions. On the other hand, we have some benchmarks which expose more particular issues of heap space analysis, such as *Results* which has constant heap space usage and *Scoreboard* which presents a multidimensional arrays creation. For all benchmarks, we have analyzed the corresponding *copy* method which performs a deep copy of the corresponding structure.

We can observe in the table that computing size relations is the most expensive step as it requires a global analysis of the program, whereas *RR* and *Cost* basically involve a single pass on the code. Our prototype implementation supports the full instructions set of sequential Java bytecode, however, it is still preliminary, and there is plenty of room for optimization, mainly in the size analysis phase, which in addition assumes the absence of cyclic data structures, which can be verified using the non-cyclicity analysis [23].

8. Conclusions and Related Work

We have presented an automatic analysis of heap usage for Java bytecode, based on generating at compile-time cost relations which define the heap space consumption of an input bytecode program. By means of a series of examples which allocate lists, trees, trees of lists, arrays, etc. in the heap, we have shown that our analysis is able to infer non-trivial bounds for them (including polynomial and exponential complexities). We believe that the experiments we have presented show that our analysis improves the state of the practice in heap space analysis of Java bytecode.

Related work in heap space analysis includes advanced techniques developed in functional programming, mainly based on type systems with resource annotations (see, e.g., [24, 17, 25, 19]) and, hence, they are quite different technically to ours. But heap space analysis is comparatively less developed for low-level languages such as Java bytecode. A notable exception is the work in [11], where a memory consumption analysis is presented. In contrast to ours, their aim is to verify that the program executes in bounded memory by simply checking that the program does not create new objects inside loops, but they do not infer bounds as our analysis does. Moreover, it is straightforward to check that new objects are not created inside loops from our cost relations. Another related work includes research in the MRG project [5, 7], which focuses on building a proof-carrying code [22] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. The analysis is developed for a functional language which then compiles to a (subset of) Java bytecode and it is restricted to linear bounds. In [6] the Bytecode Specification Language is used to annotate Java bytecode programs with memory consumption behaviour and policies, and then verification tools are used to verify those policies.

For Java-like languages, the work of [18] presents a type system for heap analysis without garbage collection, it is developed at the level of the source code and based on amortised analysis (hence it is technically quite different to our work) and, unlike us, they do not present an inference method for heap consumption. On the other hand, the work of [9] deals also with Java source code, it is able to infer polynomial complexity though it does not handle recursion.

Some works consider explicit deallocation of objects by decreasing the cost by the size of the deallocated object (see, e.g., [18, 17]). This approach is interesting when one wants to observe the heap consumption at certain program points. However, it cannot be directly incorporated in our cost relations because they are intended to provide a *global* upper bound of a method's execution. Naturally, it should happen that allocated objects are correctly deallocated and hence our cost relations would provide zero as (global) upper bound. Other work which considers cost with garbage collection is [24]. Unlike ours, it is developed for pure functional programs where the garbage collection behaviour is easier to predict as programs do not have assignments.

In the future, we want to extend our work in several directions. On the practical side, we want to incorporate an escape

analysis to transform the bytecode as outlined in Sect. 6. Regarding scalability, it is a question of performance vs. precision trade-off and depends much on the underlying abstract domain used by the size analysis. We believe our analysis would scale without sacrificing precision if an efficient domain like octagons is used together with [1]. On the theoretical side, we plan to adapt our analysis to infer upper bounds on the heap usage at given program points in the presence of garbage collection. We also would like to develop an analysis which infers upper bounds on the call stack usage.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Cost Equation Systems. *Submitted*, 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *16th European Symposium on Programming, ESOP'07*, Lecture Notes in Computer Science. Springer, March 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *Proc. of BYTECODE'07*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007.

[4] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.

[5] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, number 3362 in LNCS. Springer, 2005.

[6] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 86–95. IEEE Computer Society, 2005.

[7] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *Proc. of LPAR'04*, LNCS 3452, pages 347–362. Springer, 2004.

[8] Bruno Blanchet. Escape Analysis for Java$^{\text{tm}}$: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003.

[9] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[10] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[11] D. Cachera, D. Pichardie T. Jensen, and G. Schneider. Certified memory usage analysis. In *FM'05*, number 3582 in LNCS. Springer, 2005.

[12] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proc. of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2005.

[13] W. Chin, H. Nguyen, S. Qin, and M. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, LNCS 3672, pages 70–86. Springer, 2005.

[14] K. Crary and S. Weirich. Resource bound certification. In *Proc. of POPL'00*, pages 184–198. ACM Press, 2000.

[15] Patricia M. Hill and Fausto Spoto. Deriving Escape Analysis by Abstract Interpretation. *Higher-Order and Symbolic Computation*, (19):415–463, 2006.

[16] M. Hofmann. Certification of Memory Usage. In *Theoretical Computer Science, 8th Italian Conference, ICTCS*, volume 2841 of *Lecture Notes in Computer Science*, page 21. Springer, 2003.

[17] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *30th ACM Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM Press, 2003.

[18] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[19] J. Hughes and L. Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proc. of ICFP'99*, pages 70–81. ACM Press, 1999.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[21] Patricia M.Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. EAAI*, 2006.

[22] G. Necula. Proof-Carrying Code. In *POPL'97*. ACM Press, 1997.

[23] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of the 7th workshop on Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006. Springer-Verlag.

[24] L. Unnikrishnan, S. Stoller, and Y. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, Lecture Notes in Computer Science, pages 70–85. Springer, 2003.

[25] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.

Constancy Analysis

Samir Genaim and Fausto Spoto

[1] CLIP, Technical University of Madrid (UPM), Spain
[2] Università di Verona, Italy
samir@clip.dia.fi.upm.es, fausto.spoto@univr.it

Abstract. A reference variable x is constant in a piece of code C if the execution of C does not modify the heap structure reachable from x. This information lets us infer purity of method arguments, an important ingredient during the analysis of programs dealing with dynamically allocated data structures. We define here an abstract domain expressing constancy as an abstract interpretation of concrete denotations. Then we define the induced abstract denotational semantics for Java-like programs and show how constancy information improves the precision of existing static analyses such as sharing, cyclicity and path-length.

1 Introduction

A major difference between pure functional/logic programming and imperative programming is that the latter uses destructive updates. That is, data structures are *mutable*: they are built *and later modified*. This can be both recognized as a superiority of imperative programming, since it allows one to write faster and simpler code, and as a drawback, since if two variables share a data structure then a destructive update to the data reachable from one variable may affect the data reachable from the other. This often leads to subtle programming bugs.

It is hence important to control what a method invocation modifies. Some methods do not modify the data structures reachable from their parameters. Others only modify those reachable from some but not all parameters. Namely, some parameters are *constant* or *read-only*, others may be modified. If all parameters of a method are constant, the method is *pure* [10]. Knowledge about purity is important since pure methods can be invoked in any order, which lets compilers apply aggressive optimizations; pure methods can be used in program assertions [7]; they can be skipped during many static analyses or more precisely approximated than other methods. This results in more efficient and more precise analyses. For instance, sharing analysis [11] can safely assume that sharing is not introduced during the execution of a pure method. In general, all static analyses tracking properties of the heap benefit from information about purity.

For these reasons, software specification has found ways of expressing purity of methods and constant parameters. The notable example is the Java Modeling Language [7], which uses the `assignable` clause to specify those heap positions that might be mutated during the execution of a method. Those clauses are manually provided and used by many static analyzers, such as ESC/Java [6]

and ChAsE [4]. However, those tools do not verify the correctness of the user-provided `assignable` clauses, or use potentially incorrect verification techniques. A formally correct verification technique is defined in [14], but has never been implemented. In [10] a formally correct analysis for purity is presented, it is based on a preliminary points-to and escape analysis, and an implementation exists and has been applied to some small size examples. In [8] a correct and precise algorithm for statically inferring the reference immutability qualifiers of the Javari language has been presented. The algorithm has been implemented in the Javarifier tool.

In this paper, we investigate an alternative technique aiming at determining which parameters of a method are constant. We use abstract interpretation [5] and perform a static analysis over the reduced product of the sharing domain in [11] (the *sharing component*) and a new abstract domain expressing the set of variables bound to data structures mutated during the execution of a piece of code (the *purity component*). The use of reduced product is justified since the sharing component helps the purity component during a destructive update, by identifying which variables share the updated data structure and hence lose their purity; conversely, the sharing component uses the purity component during method calls, since only variables sharing with non-pure parameters of a method m can be made to share during the execution of m.

Our technique is sometimes less precise than [10], since it does not use the field names (i.e., we do not keep information on which field has been updated, but rather that a field has been updated). However, it is implemented in a completely flow-sensitive and context-sensitive fashion, which improves its precision. Moreover, it is expressed in terms of Boolean formulas implemented through binary decision diagrams, resulting in fast analyses scaling to quite big programs. Our contributions are hence: (1) a definition of the reduced product of sharing and purity; (2) its application to large programs; (3) a comparison of the precision of sharing analysis alone with that of sharing analysis in reduced product with purity; and (4) an evaluation of the extra precision induced by the purity information during static analyses tracking properties of the heap, namely, possible cyclicity of data structures [9] and path-length of data structures [13].

The paper is organized as follows: Section 2 defines the syntax and semantics of a simple Object-Oriented language; Section 3 develops our constancy analysis for that language; Section 4 provides an experimental evaluation.

2 Our Simple Object-Oriented Language

This section presents syntax and denotational semantics of a simple Object-Oriented language that we use through the paper. Its commands are normalized versions of corresponding Java commands: the language supports reference and integer types; in method calls, only syntactically distinct variables can be actual parameters, which is a form of normalization and does not prevent them from being bound to shared data-structures at run-time; in assignments, the left hand side is either a variable or the field of a variable; Boolean conditions are kept

generic, they are conditions that are evaluated to either $true$ or $false$; iterative constructs, such as the `while` loop, are not supported since they can be implemented through recursion. These assumptions are only for the sake of clear and simple presentation and can be relaxed without affecting subsequent results. A program has a set of *variables* $\mathcal{V}$ (including *out* and *this*) and a finite poset of *classes* $\mathbb{K}$. The *commands* of the language are

$$\begin{aligned} com ::= \quad & v := c \mid v := w \mid v := \texttt{new}\ \kappa \mid v := w + z \mid v := w.\texttt{f} \mid v.\texttt{f} := w \mid \\ & v := v_0.\texttt{m}(v_1, \ldots, v_n) \mid \texttt{if}\ e\ \texttt{then}\ com_1\ \texttt{else}\ com_2 \mid com_1; com_2 \end{aligned}$$

$v, w, z, v_0, v_1, \ldots, v_n \in \mathcal{V}$ are distinct variables, $c \in \mathbb{Z} \cup \{\mathsf{null}\}$, $\kappa \in \mathbb{K}$ and e is a Boolean expression. The signature of a method $\kappa.\texttt{m}(t_1, \ldots, t_p){:}t$ refers to a method called `m` expecting p parameters of type $t_1, \ldots, t_p \in \mathbb{K} \cup \{\mathsf{int}\}$, respectively, returning a value of type t and *defined* in class κ with a statement

$$t\ \texttt{m}(w_1{:}t_1, \ldots, w_n{:}t_n)\ \texttt{with}\ \{w_{n+1}{:}t_{n+1}, \ldots, w_{n+m}{:}t_{n+m}\}\ \texttt{is}\ com,$$

where $w_1, \ldots, w_n, w_{n+1}, \ldots, w_{n+m} \in \mathcal{V}$ are distinct, not in $\{out, this\}$ and have *type* $t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+m} \in \mathbb{K} \cup \{\mathsf{int}\}$, respectively. Variables $w_1, \ldots, w_n$ are the *formal parameters* of the method and $w_{n+1}, \ldots, w_{n+m}$ are its *local variables*. The method also uses a variable *out* of type t to store its *return value*. For a given method signature $m = \kappa.\texttt{m}(t_1, \ldots, t_p) : t$, we define $m^b = com$, $m^i = \{this, w_1, \ldots, w_n\}$, $m^o = \{out\}$, $m^l = \{w_{n+1}, \ldots, w_{n+m}\}$ and $m^s = m^i \cup m^o \cup m^l$. Classes might declare fields of type $t \in \mathbb{K} \cup \{\mathsf{int}\}$.

We use a denotational semantics, hence compositional, in the style of [15]. However, we use a more complex notion of state, which assumes an infinite set of *locations*. Basically, a state is a pair which consists of a frame and a heap, where a frame maps variables to values and a heap maps locations to objects. Note that since we assume a denotational semantics, a state has a single frame, rather than an activation stack of frames as it is required in operational semantics. We let $\mathbb{L}$ denote an infinite set of *locations*, and let $\mathbb{V}$ denotes the set of *values* $\mathbb{Z} \cup \mathbb{L} \cup \{\texttt{null}\}$. A *frame* over a finite set of variables V is a mapping that maps each variable in V into a value from $\mathbb{V}$; a *heap* is a partial map from $\mathbb{L}$ into *objects*. An object is a pair that consists of its class tag κ and a frame that maps its *fields* (identifiers) into values from $\mathbb{V}$; we say that it *belongs to* class κ or *has* class κ. Given a class κ, we assume that $newobj(\kappa)$ return a new object where its fields are initialized to 0 or depending on their types. If ϕ is a frame and $v \in V$, then $\phi(v)$ is the value of variable v. If μ is a heap and $\ell \in \mathbb{L}$, then $\mu(\ell)$ is the object bound in μ to ℓ. If o is an object, then $o.tag$ denotes its class and $o.\phi$ denotes its frame; if f is a field of o, then sometimes we use $o.f$ to refer to (or set) its value instead of going through its frame.

Definition 1 (computional state). *Let V denotes the set of variables in scope at a given program point p. The set of possible* states *at p is*

$$\Sigma_V = \left\{ \langle \phi, \mu \rangle \;\middle|\; \begin{array}{l} 1.\ \phi \text{ is a frame over } V \text{ and } \mu \text{ is a heap} \\ 2.\ \mathrm{rng}(\phi) \cap \mathbb{L} \subseteq \mathrm{dom}(\mu) \\ 3.\ \forall \ell \in \mathrm{dom}(\mu).\ \mathrm{rng}(\mu(\ell).\phi) \cap \mathbb{L} \subseteq \mathrm{dom}(\mu) \end{array} \right\}$$

Conditions 2 *and* 3 *guarantee the absence of dangling pointers. Given* $\sigma = \langle \phi, \mu \rangle \in \Sigma_V$*, we use* ϕ_σ *and* μ_σ *to refer to its frame and heap respectively.* □

Now we define the notion of *Denotations* which are the input/output semantics of a piece of code. Basically they are mappings from states to states which describe how the input state is changed when the corresponding code is executed. *Interpretations* are a special case of denotations which provide a denotation for each method in terms of its input and output variables.

Definition 2. *A* denotation δ *from* V *to* V' *is a partial function from* Σ_V *to* $\Sigma_{V'}$*. We often refer to* $\delta(\sigma) = \sigma'$ *as* $(\sigma, \sigma') \in \delta$*. The set of denotations from* V *to* V' *is* $\Delta(V,V')$*. An* interpretation ι *maps methods to denotations and is such that* $\iota(m) \in \Delta(m^i, m^i \cup m^o)$ *for each method* $m = \kappa.\mathtt{m}(t_1, \ldots, t_p) : t$ *in the given program. The set of all possible interpretations is written as* $\mathbb{I}$*.* □

The denotational semantics associates a denotation to each command of the language. Let V denotes a set of variables. Let $\iota \in \mathbb{I}$. We define the *denotation for commands* $\mathcal{C}^\iota_V[\![\_]\!] : com \mapsto \Delta(V,V)$, as their input/output behaviour:

$$
\begin{aligned}
\mathcal{C}^\iota_V[\![v{:=}c]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto c]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}w]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \phi_\sigma(w)]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}\mathtt{new}\ \kappa]\!] &= \{(\sigma, \sigma[\mu_\sigma(\ell) \mapsto newobj(\kappa)]) \mid \sigma \in \Sigma_V,\ \ell \notin \mathrm{dom}(\mu_\sigma)\} \\
\mathcal{C}^\iota_V[\![v{:=}w + z]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \phi_\sigma(w) + \phi_\sigma(z)]) \mid \sigma \in \Sigma_V\} \\
\mathcal{C}^\iota_V[\![v{:=}w.f]\!] &= \{(\sigma, \sigma[\phi_\sigma(v) \mapsto \mu_\sigma\phi_\sigma(w).f]) \mid \sigma \in \Sigma_V, \phi_\sigma(w) \neq \mathtt{null}\} \\
\mathcal{C}^\iota_V[\![v.f{:=}w]\!] &= \{(\sigma, \sigma[\mu_\sigma\phi_\sigma(v).f \mapsto \phi_\sigma(w)]) \mid \sigma \in \Sigma_V, \phi_\sigma(v) \neq \mathtt{null}\} \\
\mathcal{C}^\iota_V\left[\!\!\left[\begin{array}{r}\mathtt{if}\ e\ \mathtt{then}\ com_1 \\ \mathtt{else}\ com_2\end{array}\right]\!\!\right] &= \begin{array}{l}\{(\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_1]\!] \mid \sigma \models e \approx true\} \cup \\ \{(\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_2]\!] \mid \sigma \models e \approx false\}\end{array} \\
\mathcal{C}^\iota_V[\![com_1; com_2]\!] &= \{(\sigma, \sigma'') \mid (\sigma, \sigma') \in \mathcal{C}^\iota_V[\![com_1]\!] \wedge (\sigma', \sigma'') \in \mathcal{C}^\iota_V[\![com_2]\!]\}
\end{aligned}
$$

The denotation for a method call $\mathcal{C}^\iota_V[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!]$ should consider the denotation $\iota(m)$ (where m is the called method) and extend it to fit in the calling scope and update the variable v. Assume the method signature is $\mathtt{m}(t_1, \ldots, t_p){:}t$, and that we have a lookup procedure $\mathcal{L}$ that, for any given $\sigma \in \Sigma_V$, fetches the actual method that is called depending on the run-time class of v_0. Then the method call denotation is defined as follows:

$$
\left\{ (\sigma, \langle \phi_\sigma[v \mapsto \phi''_\sigma(out)], \mu''_\sigma \rangle) \;\middle\|\; \begin{array}{l} 1.\ \sigma \in \Sigma_V, \phi_\sigma(v_0) \in \mathrm{dom}(\mu_\sigma); \\ 2.\ m = \mathcal{L}(v_0, \sigma, \mathtt{m}(t_1, \ldots, t_p){:}t); \\ 3.\ (\sigma', \sigma'') \in \iota(m); \\ 4.\ \mu_\sigma \equiv \mu'_\sigma, \forall 0 \leq i \leq p.\ \phi_\sigma(v_i) = \phi'_\sigma(w_i) \end{array} \right\}
$$

The concrete denotational semantics of a program is the least fixpoint of the following transformer of interpretations [3].

Definition 3 (Denotational semantics). *The denotational semantics of a program* P *is defined as* $\bigcup_{i \geq 0} T^i_P(\iota_0)$ *, i.e. the* least fixed point *of* T_P *where* T_P *is:*

$$T_P(\iota) = \left\{ (m, X) \;\middle\|\; \begin{array}{l} 1.\ m \in P \\ 2.\ \sigma \in \Sigma_{m^s}, \forall v \in m^l.\ \phi_\sigma(v) = 0 \text{ or } \phi_\sigma(v) = \texttt{null} \\ 3.\ X = \{(\sigma|_{m^i}, \sigma'|_{m^i \cup m^o}) \mid (\sigma, \sigma') \in \mathcal{C}^\iota_{m^s}[\![m^b]\!]\} \end{array} \right\}$$

and $\iota_0 = \{(m, \emptyset) \mid m \in P\}$ *and* $\forall \iota_1, \iota_2 \in \mathbb{I}$ *the union* $\iota_1 \cup \iota_2$ *is defined as* $\{(m, X_1 \cup X_2) \mid m \in P, (m, X_1) \in \iota_1, (m, X_2) \in \iota_2\}$ □

3 Constancy Analysis

We want to design an analysis to infer definite information about constant data structures. This can be done by tracking data structures that are not modified (definite information), or by tracking data structures that might be modified (may information). We follow the latter approach as we believe it easier. In addition, we want to analyze methods in a context independent way, and later adapt the result to any calling context.

Example 1. Consider the following method:

```
A m(x:A, y:A) with {} is y:=y.next; x.next:=y; out:=y;
```

The only command that might modify the heap structure is "`x.next:=y`". Note that "`y:=y.next`" does not affect the heap structure but rather changes the heap location stored in `y`. This method might be called in different contexts where the actual parameters: (1) do not have any common data structure; or (2) have a common data structure. In the first case, "`x.next:=y`" might modify only the data structure pointed by the first argument. In the second case, it might modify a common data structure for `x` and `y`, and therefore we say that both arguments might be modified. We describe this behaviour by the Boolean formula $\check{x} \wedge (\check{y} \leftrightarrow x\check{}y)$, which is interpreted as: (1) in any calling context, the data structure the first argument points to when the method is called might be modified by the method (expressed by $\check{x}$); and (2) the data structure that the second argument points to when the method is called, might be modified by the method (expressed by $\check{y}$) iff x and y might share a data structure when the method is called (expressed by $x\check{}y$).

□

We define now the set of reachable heap locations from a given reference variable, which we need to define the notion of *constant heap structure*.

Definition 4 (reachable heap locations). *Let* μ *be a heap. The set of* locations reachable from $\ell \in \mathrm{dom}(\mu)$ *is* $L(\mu, \ell) = \cup\{L^i(\mu, \ell) \mid i \geq 0\}$ *where* $L^0(\mu, \ell) = \mathrm{rng}(\mu(\ell).\phi) \cap \mathbb{L}$ *and* $L^{i+1}(\mu, \ell) = \cup\{\mathrm{rng}(\mu(\ell')) \cap \mathbb{L} \mid \ell' \in L^i(\mu, \ell)\}$. *The set of reachable heap locations from* v *in* $\sigma \in \Sigma_V$*, denoted* $L_V(\sigma, v)$*, is* $\{\phi_\sigma(v)\} \cup L(\mu_\sigma, \phi_\sigma(v))$ *if* $\phi_\sigma(v) \in dom(\mu_\sigma)$*; and the empty set otherwise.* □

Definition 5 (constant reference variable). *A reference variables $v \in V$ is constant with respect to a denotation δ, denoted $\mathsf{c}(v, \delta)$, iff for any $(\sigma_1, \sigma_2) \in \delta$ all locations in $L_V(\sigma_1, v)$ are constant with across δ, namely $\forall \ell \in L_V(\sigma_1, v)$, $\mu_{\sigma_1}(\ell)$ and $\mu_{\sigma_2}(\ell)$ have the same class tag and agree on their* reference *field values.* □

The definition above considers modifications of fields of reference type only. The reason for concentrating on reference fields is that we have developed this analysis for a specific need which requires tracking updates only in the shape of the data structure (see Section 4). Tracking updates of integer fields can simply done by modifying the above definition to consider those updates. In what follows, a modification of a variable stands for a modification of the shape of the heap structure reachable from that variable.

Definition 6 (common heap location). *$x, y \in V$ have a common heap location (share) in a state $\sigma \in \Sigma_V$ if and only if $L_V(\sigma, x) \cap L_V(\sigma, y) \neq \emptyset$* □

We define now an abstract domain which captures a set of variables that *might* be modified by a concrete denotation.

Definition 7 (update abstract domain). *The update abstract domain U_V is a partial order $\langle \wp(V), \subseteq \rangle$. Its concretization function $\gamma_V{:}\mathsf{U}_V \to \Delta(V,V')$ is defined as $\gamma_V(X) = \{\delta \mid \forall v \in V.\ \neg \mathsf{c}(v, \delta) \to (v \in X)\}$.* □

As we have seen in Example 1, information about possible sharing between variables is important for a precise constancy analysis. There are many ways for inferring such information. Here, we use the pair-sharing domain [11]. Moreover, constancy information improves the precision of method calls in pair sharing analysis. This is because the execution of a method m can introduce sharing between non-constant parameters only. Hence we design an analysis over the (reduced) product of the update domain U_V and of the pair-sharing domain SH_V, denoted by $\mathsf{SH}{\times}\mathsf{U}_V$. Informally, the pair sharing domain abstracts an element $s \in \wp(\Sigma_V)$ to a set sh of symmetric pairs of the form (x, y) where $x, y \in V$. If $(x, y) \in sh$ then x and y *might* share in s, and if $(x, y) \notin sh$ then they cannot share, so that if $(x, x) \notin sh$ then x must be `null` in s. In what follows, instead of saying *might share* we simply say share.

Figure 1 defines abstract denotations for our simple language over $\mathsf{SH}{\times}\mathsf{U}_V$. They are Boolean functions corresponding to the elements of $\mathsf{SH}{\times}\mathsf{U}_V$. For a piece of code C, the Boolean variables:

- $x\check{}y$ and $x\hat{}y$ indicate if x and y *share* before and after executing C, respectively. Since pair sharing is symmetric, $x\check{}y$ and $y\check{}x$ are equivalent Boolean variables; and
- $\check{x}$ and $\hat{x}$ indicate if x is modified with respect to its value before and after C (by the program execution), respectively.

Each abstract denotation is defined in terms of a Boolean function $\varphi \wedge \psi$, where φ propagates (forward) *sharing* information and ψ propagates (backwards) *update* information. In what follows we explain the meaning of each abstract denotation:

$$
\begin{aligned}
\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge \varphi_1 \\
&\quad -\varphi_1 = (\wedge\{\neg x\hat{\cdot}v \mid x \in V\}) \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge \varphi_1 \wedge \varphi_2 \\
&\quad -\varphi_1 = \wedge\{x\hat{\cdot}v \leftrightarrow x\check{\cdot}w \mid x \in V\backslash\{v\}\} \\
&\quad -\varphi_2 = w\check{\cdot}w \leftrightarrow v\hat{\cdot}v \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{new}\ \kappa]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \mathsf{Id}_{sh}(V\backslash\{v\}) \wedge v\hat{\cdot}v \wedge \varphi_1 \\
&\quad -\varphi_1 = (\wedge\{\neg x\hat{\cdot}v \mid x \in V \setminus \{v\}\}) \\
&\quad -\psi = \mathsf{Id}_{u}(V\backslash\{v\}) \wedge (\check{v} \leftrightarrow \vee\{v\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}) \\
\mathcal{A}^{\iota}_{V}[\![v{:=}w.f]\!] &= \mathcal{A}^{\iota}_{V}[\![v{:=}w]\!] \\
\mathcal{A}^{\iota}_{V}[\![v.f{:=}w]\!] &= \varphi \wedge \psi \\
&\quad -\varphi = \wedge\{x\hat{\cdot}y \leftrightarrow x\check{\cdot}y \vee (x\check{\cdot}w \wedge y\check{\cdot}v) \mid x, y \in V\} \\
&\quad -\psi = \{\check{x} \leftrightarrow v\check{\cdot}x \vee \hat{x} \mid x \in V\} \\
\mathcal{A}^{\iota}_{V}[\![\texttt{if}\ e\ \ldots]\!] &= \mathcal{A}^{\iota}_{V}[\![c_1]\!] \vee \mathcal{A}^{\iota}_{V}[\![c_2]\!] \\
\mathcal{A}^{\iota}_{V}[\![c_1; c_2]\!] &= \mathcal{A}^{\iota}_{V}[\![c_1]\!] \circ \mathcal{A}^{\iota}_{V}[\![c_2]\!] \\
\mathcal{A}^{\iota}_{V}[\![v{:=}v_0.\mathtt{m}(v_1, \ldots, v_p)]\!] &= \phi \wedge \varphi \wedge \psi \\
&\quad \phi_m = \vee\{\iota(m) \mid m \text{ might be called }\} \\
&\quad \phi = \phi_m[s_i \mapsto v_i, \mathit{out} \mapsto v, \mathit{this} \mapsto v_0] \\
&\quad \varphi = \wedge\{x\hat{\cdot}y \leftrightarrow x\check{\cdot}y \vee \varphi_1 \mid x, y \in V\backslash\{v_0, \ldots, v_p\}\} \\
&\quad \varphi_1 = \vee\{(x\check{\cdot}v_i \wedge y\check{\cdot}v_j \wedge v_i\hat{\cdot}v_j \wedge (\check{v}_i \vee \check{v}_j)) \mid i, j \in \{0, \ldots, p\}\} \\
&\quad \psi = \psi_1 \wedge (\check{v} \leftrightarrow \psi_3 \vee \psi_2(v)) \\
&\quad \psi_1 = \wedge\{\check{x} \leftrightarrow \hat{x} \vee \psi_2(x) \mid x \in V\backslash\{v, v_0, \ldots, v_p\}\} \\
&\quad \psi_2(x) = \vee\{(x\check{\cdot}v_i \wedge \check{v}_i) \mid i \in \{0, \ldots, p\}\} \\
&\quad \psi_3 = \{x\check{\cdot}y \wedge \hat{y} \mid y \in V\backslash\{v\}\}
\end{aligned}
$$

Fig. 1. Abstract Denotations over $\mathsf{SH}{\times}\mathsf{U}_V$

- $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!]$: (**SH**) sharing between $x, y \in V\backslash\{v\}$ is preserved ($\mathsf{Id}_{sh}(V\backslash\{v\})$); and nothing can share with v after C (φ_1). (**U**) $x \in V\backslash\{v\}$ is modified before C iff it is modified after C, and v is modified before C iff it shares with some y before C and y is modified after C.
- $\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!]$: (**SH**) sharing between $x, y \in V\backslash\{v\}$ is preserved ($\mathsf{Id}_{sh}(V\backslash\{v\})$); since v becomes an alias for w then v can share with $x \in V\backslash\{v\}$ after C iff x shares with w before C (φ_1); and v can share with itself after C (i.e., not `null`) iff w shares with itself before C (φ_2). (**U**) the same as for "$v{:=}$`null`".
- $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{new}\ \kappa]\!]$: the same as $\mathcal{A}^{\iota}_{V}[\![v{:=}\texttt{null}]\!]$ except that v shares with itself after executing the statement.
- $\mathcal{A}^{\iota}_{V}[\![v{:=}w.f]\!]$: the same as $\mathcal{A}^{\iota}_{V}[\![v{:=}w]\!]$ since the analysis is field insensitive.
- $\mathcal{A}^{\iota}_{V}[\![v.f{:=}w]\!]$: (**SH**) $x, y \in V$ share after C iff before C, they shared or x shared with w and y with v; (**U**) $x \in V$ is modified before C, iff it shares with v before C or x is modified after C.
- $\mathcal{A}^{\iota}_{V}[\![\texttt{if}\ e\ \ldots]\!]$: combines the branches through logical or.

- $\mathcal{A}^\iota_V[\![c_1; c_2]\!]$: combines $\mathcal{A}^\iota_V[\![c_1]\!]$ and $\mathcal{A}^\iota_V[\![c_2]\!]$. This is simply done by matching the output variables of the first denotation with the input variables of the second denotation.
- $\mathcal{A}^\iota_V[\![v{:=}v_0.\mathtt{m}(v_1,\ldots,v_p)]\!]$: (1) First we fetch the abstract denotations of all methods that might be called, and we combine them through logical or into ϕ_m; (2) Assuming that the method denotations use $s_i \neq v_i$ for the i-th formal parameter, we rename all sharing information by changing each s_i into v_i and *out* into v. We get ϕ. (3) We add sharing information for variables which are not in $V\backslash\{v, v_0, \ldots, v_p\}$. The sharing component φ states that x and y might share after the call iff they shared before (i.e. $x\check{\cdot}y$) or they shared with arguments v_i and v_j where v_i and v_j share after the call, and either v_i or v_j has been modified (expressed by φ_1); (4) We add the constancy information which states that $x \in V\backslash\{v\}$ is modified before iff it is modified after, or if it shares with a variable that is modified by the method. For v it is a bit different since we exclude the case that if v is modified after then it is modified before, since we possibly assign to it a new reference.

The abstract denotation for a method:

$$t\ \mathtt{m}(w_1{:}t_1,\ldots,w_n{:}t_n)\ \mathtt{with}\ w_{n+1}{:}t_{n+1},\ldots,w_{n+m}{:}t_{n+m}\ \mathtt{is}\ com,$$

is then defined as $\phi_m = \exists V'.\ \mathcal{A}^\iota_V[\![com]\!] \wedge \varphi_1 \wedge \varphi_2$ where:

- $S = \{s_1, \ldots, s_n\}$ such that $S \cap m^s = \emptyset$, and $V = m^s \cup S$
- $\varphi_1 = \{\neg x\check{\cdot}y \mid x \in m^l \cup \{out\}, y \in m^s\}$
- $\varphi_2 = \{s_i\check{\cdot}x \leftrightarrow w_i\check{\cdot}x \mid 1 \leq i \leq n, x \in m^i\}$
- $V' = \{x\check{\cdot}y, x\hat{\cdot}y, \check{x}, \hat{x} \mid x \notin S \cup \{this, out\}, y \in V\} \cup \{o\check{u}t\}$

The idea is that we: (1) extend m^l to V in order to include shallow variable s_i for each method argument w_i; (2) compute $\mathcal{A}^\iota_V[\![com]\!]$; (3) add φ_1 which indicates that local variables are initialized to `null`; (4) add φ_2 which creates the connection between the shallow variables and the actual parameters; (5) eliminate all local information by removing the Boolean variables V'. The abstract denotational semantics can be then defined similar to the concrete one in Definition 3, where the initial method summaries are $false$ ans summaries are combined (during the fixpoint iterations) using the logical or $\vee$.

Example 2. Applying the above abstract semantics to the method defined in Example 1 results in a Boolean formula whose constancy component is $(t\check{h}is \leftrightarrow t\hat{h}is) \wedge \check{x} \wedge (\check{y} \leftrightarrow (x\check{\cdot}y \vee \hat{y}))$. For simplicity we ignore the part of ϕ_m that talks about sharing.

4 Experiments

We show here some experiments with our domain for sharing and constancy analysis. They have been performed with the Julia analyzer [12] on a Linux

Program	M	Sharing		Non-Cyclicity	
		T	P	T	P
`JLex`	446	1595 (2324)	34.30% (34.84%)	506 (415)	34.03% (35.21%)
`JavaCup`	933	5707 (6486)	22.24% (23.76%)	853 (953)	59.23% (76.13%)
`Kitten`	2131	20976 (27824)	17.90% (19.11%)	2538 (3177)	36.34% (41.13%)
`jEdit`	3206	47408 (49356)	21.12% (21.28%)	4969 (5963)	43.49% (47.50%)
`Julia`	4028	79199 (129562)	9.71% (10.25%)	8014 (12018)	33.40% (38.17%)

Fig. 2. The effect of the purity component on Sharing and Non-Cyclicity. (M) number of methods; (T) run-time in milliseconds excluding preprocessing; (P) precision.

machine based on a 64 bits dual core AMD Opteron processor 280 running at 2.4Ghz, with 2 gigabytes of RAM and 1 megabyte of cache, by using Sun Java Development Kit version 1.5. All programs have been analyzed including all library methods that they use inside the `java.lang.*` and `java.util.*` hierarchies.

Figure 2 compares sharing analysis alone with sharing analysis in reduced product with constancy (Section 3), and its effect on non-cyclicity analysis [9]. In each column, numbers in parentheses correspond to the analysis using the reduced product. For each program, it reports the number of methods analyzed, including the libraries, and time and precision of the corresponding analysis with and without constancy. For sharing, the precision is the amount of pairs of variables of reference type that are proved not to share at the program points preceding the update of an instance field, the update of an array element or a method call. This is sensible since there is where sharing analysis is used by subsequent analyses. That figure suggests that the constancy component slightly improves the precision of sharing analysis. However, the importance of constancy is shown when we consider its effects on a static analysis that uses constancy information. This is the case of non-cyclicity analysis, which finds variables bound to non-cyclical data structures [9]. Figure 2 shows that the computation of cyclicity analysis after a simple sharing analysis leads to less precise results than the same computation after a sharing and constancy analysis. Here, precision is the number of field accesses that read the field of a non-cyclical object. This is sensible since there is where non-cyclicity is typically used.

The importance of constancy analysis becomes more apparent when it supports a static analysis that uses constancy, sharing and cyclicity information. This is the case of *path-length* [13]. It approximates the length of the maximal path of pointers one can follow from each variable. This information is the basis of a termination [1] and resource bound analyses [2] for programs dealing with dynamic data structures. Figure 3 shows the effects of constancy on path-length and termination analysis (available in [12]) of a set of small programs that do not use libraries except for `java.lang.Object`. Times are in milliseconds and precision is the number of methods proved to terminate. Constancy information

Program	M	T	P
`Init`	10	102 (140)	8 (8)
`List`	11	624 (512)	6 (11)
`Diff`	5	6668 (9040)	5 (5)
`Hanoi`	7	548 (868)	7 (7)
`BTree`	7	306 (415)	6 (7)
`BSTree`	10	234 (273)	9 (10)
`Virtual`	11	357 (418)	10 (11)
`ListInt`	11	767 (507)	6 (11)

Program	M	T	P
`Nested`	4	324 (447)	4 (4)
`Double`	5	270 (268)	5 (5)
`FactSum`	6	169 (178)	6 (6)
`Sharing`	7	309 (501)	6 (7)
`Factorial`	5	102 (196)	5 (5)
`Ackermann`	5	1308 (1732)	5 (5)
`BubbleSort`	5	871 (951)	5 (5)
`FactSumList`	8	278 (703)	7 (8)

Fig. 3. The effect of the purity information on Termination analysis. (M) number of methods; (T) run-time in milliseconds excluding preprocessing; (P) precision.

results in proving that all terminating methods terminate (only 2 methods of `Init` are not proved to terminate: they actually diverge). Without constancy information, many terminating methods are not proved to terminate.

These experiments suggest that constancy information contributes to the precision of sharing, cyclicity, path-length and hence termination analysis. Computing constancy information with sharing requires more time than computing sharing alone (Figure 2). Performing other analyses by using the constancy information increases the times further (Figures 2 and 3). Nevertheless, this is justified by the extra precision of the results.

5 Acknowledgments

This work of Samir Genaim was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project, and a *Juan de la Cierva* Fellowship awarded by the Spanish Ministry of Science and Education. The authors would like to thank the anonymous referees for their useful comments.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, Lecture Notes in Computer Science, Oslo, Norway, June 2008. Springer-Verlag, Berlin. To appear.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19-20:149–197, 1994.
4. N. Cataño and M. Huisman. Chase: A Static Checker for JML's Assignable Clause. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proc. of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2003.
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.
6. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, COMPAQ Systems Research Center, 1998.
7. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML. Technical Report 96-06p, Iowa State University, 2001.
8. Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 9–11, 2008.
9. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006.
10. A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In R. Cousot, editor, *Proc. of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215, Paris, France, 2005. Springer.
11. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin, editor, *Proc. of Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, September 2005.
12. F. Spoto. The JULIA Static Analyser. `profs.sci.univr.it/~spoto/julia`, 2008.
13. F. Spoto, P. M. Hill, and E. Payet. Path-Length Analysis for Object-Oriented Programs. In *Proc. of Emerging Applications of Abstract Interpretation*, Vienna, Austria, March 2006. `profs.sci.univr.it/~spoto/papers.html`.
14. F. Spoto and E. Poll. Static Analysis for JML's assignable Clauses. In G. Ghelli, editor, *Proc. of FOOL-10, the 10th ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, New Orleans, Louisiana, USA, January 2003. ACM Press. Available at `www.sci.univr.it/~spoto/papers.html`.
15. G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

Efficient Context-Sensitive Shape Analysis with Graph Based Heap Models

Mark Marron[1], Manuel Hermenegildo[1,2], Deepak Kapur[1], and Darko Stefanovic[1]

[1]University of New Mexico
{marron,kapur,darko}@cs.unm.edu
[2] Technical University of Madrid and IMDEA-Software
herme@fi.upm.es

Abstract. The performance of heap analysis techniques has a significant impact on their utility in an optimizing compiler. Most shape analysis techniques perform interprocedural dataflow analysis in a context-sensitive manner, which can result in analyzing each procedure body many times (causing significant increases in runtime even if the analysis results are memoized). To improve the effectiveness of memoization (and thus speed up the analysis) *project*/*extend* operations are used to remove portions of the heap model that cannot be affected by the called procedure (effectively reducing the number of different contexts that a procedure needs to be analyzed with). This paper introduces *project*/*extend* operations that are capable of accurately modeling properties that are important when analyzing non-trivial programs (sharing, nullity information, destructive recursive functions, and composite data structures). The techniques we introduce are able to handle these features while significantly improving the effectiveness of memoizing analysis results (and thus improving analysis performance). Using a range of well known benchmarks (many of which have not been successfully analyzed using other existing shape analysis methods) we demonstrate that our approach results in significant improvements in both accuracy and efficiency over a baseline analysis.

1 Introduction

Recent work on shape analysis techniques [25,28,1,14,15,9,8] has resulted in a number of techniques that are capable of accurately representing the properties (connectivity, interference, and shape) that are needed for a range of optimization and parallelization applications. However, the computational cost of performing these analyses has limited their applicability. A significant component of the analysis runtime is due to the need to perform a context-sensitive interprocedural analysis, where each procedure body may be analyzed multiple times (once for each different calling context).

The practice of using a memo-table to avoid recomputing analysis results and the use of a *project* operation to remove portions of the heap that cannot affect or be affected by the called procedure are standard techniques for minimizing the number of times each function needs to be analyzed during interprocedural dataflow analysis [2,17,16,19]. The two major goals of the *project* operation are improving the effectiveness of memoizing analysis results by removing portions of the heap that could cause spurious inequalities

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 245–259, 2008.

between calling contexts and preventing the loss of precision that occurs when recursive procedures use a summary representation for multiple out-of-scope references (e.g. local reference variables with the same name but that exist in different call frames).

The *project* operation for heap models and the utility of locality axioms have been analyzed in a number of papers [22,21,7,12,4]. These techniques use variations on the notion of a *frame rule* as presented in [11,20] and identify a number of features of the *project* operation that are of particular importance for interprocedural analysis using heap domains. A major distinction is made between the projection operation in *cutpoint*-free cases, where there are no pointers that cross from a section of the heap that is *unreachable* from the procedure arguments into a section of the heap that is *reachable* from the procedure arguments, and cases where such pointers may exist.

This paper presents a method for using cutpoints to support interprocedural heap analysis. We then use the technique to quickly analyze (10's of seconds) programs that are larger (by a factor of 2-4) and more varied (in terms of data structures and algorithms) than any other analysis technique to date. Our first contribution is the reformulation of the project/extend operations in [21] so that they can be used in a graph based (as opposed to an access path based) heap model which allows us to use a very compact and efficient representation of heap connectivity. Our second contribution is the extension of the original approach to handle two classes of programatic events that are critical to analyzing real world programs, analyzing programs that involve non-trivial sharing and composite data structures [1,15] and propagating nullity test information from callee to caller scope. Finally we use the results of the heap analysis to drive the parallelization of a range of benchmarks (several of which have not been successfully analyzed/parallelized using shape information) achieving an average parallel speedup of 1.69 on a dual-core machine.

2 Example Code

To develop intuition about the mechanism and purpose of *project*/*extend* operations we look at a simple function (Figure 1) that illustrates the basic functioning of the *project*/*extend* operations and the propagation of nullity information from the callee to the caller scope. Our lists are made of objects of *type* `LNode`, each `LNode` object has two fields, a `nx` field which refers to the next element in the list and a field `f` which stores a boolean.

```
LNode LInit(LNode l)
  if(l == null)
    return;

  tin = l.nx;
  LInit(tin);
  l.f = true;
```

Fig. 1. Recursive List Initialize

Accurately analyzing the initialization method (LInit) requires the analysis to propagate information inferred about cutpoints in the callee scope back into the caller scope. If the analysis is unable to use the `l == null` test in the callee scope to infer that `l.nx` is `null` in the caller scope then the analysis will not be able to infer that after the method returns the argument list is either `null` or must have the `true` value in all the `f` fields.

3 Heap Model

We model the concrete heap as a labeled, directed multi-graph (V,E) where each vertex $v \in V$ is an object in the store or a variable in the environment, and each labeled directed edge $e \in E$ represents a pointer between objects or a reference from a variable to an object. Each edge is given a label that is an identifier from the program, an edge $(a,b) \in E$ labeled with p, we use the notation $a \xrightarrow{p} b$ to indicate that a points to the object b via the field name (or identifier) p.

A *region* of memory $\Re$ is a subset of the objects in memory, with all the pointers that connect these objects and all the cross-region pointers that start or end at an object in this region. Formally, let $C \subseteq V$ be a subset of objects, and let $P_i = \{p \mid \exists a,b \in C, a \xrightarrow{p} b\}$ and $P_c = \{p \mid \exists a \in C, x \notin C, a \xrightarrow{p} x \vee x \xrightarrow{p} a\}$ be respectively the set of internal and cross-region pointers for C. Then a region is the tuple (C,P_i,P_c). For a region $\Re = (C,P_i,P_c)$ and objects $a,b \in C$, we say a and b are *connected* in $\Re$ if they are in the same weakly-connected component of the graph (C,P_i). Objects a and b are *disjoint* in $\Re$ if they are in different weakly-connected components of the graph.

3.1 Abstract Heap Model

The underlying abstract heap domain is a graph where each node represents a region of the heap or a variable and each edge represents a set of pointers or a variable target. The nodes and edges are augmented with additional instrumentation predicates. The abstract domain evaluates the predicates using a *3-valued* semantics: predicates are either definitely true, definitely false, or unknown [25]. Our analysis tracks the following set of instrumentation predicates. Our choice of predicates is influenced by common predicates tracked in previous papers on shape analysis [5,24,28,20].

Types. For each type t in the program, there is an instrumentation predicate (also written t) that is true at a concrete heap node if any concrete object represented by the node may have type t.

Linearity. Each abstract node has a *linearity* that represents whether it represents at most one concrete node (linearity 1) or any set of 0 or more concrete nodes (written #).

Abstract Layout. To track the connectivity and shape of the region a node abstracts, the analysis uses *abstract layout* predicates *Singleton*, *List*, *Tree*, *MultiPath*, or *Cycle*. The *Singleton* predicate states that there are no pointers between any of the objects represented by an abstract node. The *List* predicate is similar to the inductive *List* predicate

in separation logic [20]. The other predicates correspond to the definitions for Trees, Dags, and Cycles in the literature, for the formal definitions see [14].

Interference. The heap model uses two properties to track the potential that two references can reach the same memory location in the region that a node represents.

The first property is for references that are represented by different edges in the heap model. Given the concretization function γ and two edges e_1, e_2 that are incoming edges to the node n, the predicate that defines *inConnected* in the abstract domain is: e_1, e_2 are *inConnected* with respect to n if it is possible that $\exists r_1 \in \gamma(e_1) \wedge \exists r_2 \in \gamma(e_2) \wedge \exists a, b \in \gamma(n)$ s.t. (r_1 refers to a) $\wedge$ (r_2 refers to b)$\wedge$ (a, b connected). For improved precision we also track *may* and *must* aliasing (e_1, e_2 are *inConnected* and $a = b$) between the references the edges abstract (*must* aliasing is only meaningful if the edge represents a single references, see [15] for an approach that generalizes *must-aliasing* to sets of references).

The second property is for the case where the references are represented by the same edge. To model this the *interfere* property is introduced. An edge e represents interfering references if there may exist references $r_1, r_2 \in \gamma(e)$ such that the objects that r_1, r_2 refer to are connected/aliased. A three-element lattice, $np < ip < ap$, *np* for edges with all non-interfering references and *ip* for potentially interfering references and *ap* for potentially aliasing references, is used to represent the interference property.

The Heap Graph. Each node in the graph either represents a region of the heap or a variable. The variable nodes are labeled with the variable that they represent. Nodes representing the concrete heap regions contain a record that tracks the types of the concrete objects that the node represents (*types*), the number of objects (either 1 or #) that may be in the region (*count*), and the abstract layout of a node (*layout*). Each node also tracks the connectivity relation between pairs of incoming edges. A binary relation *connR* is used to track the *inConnected* relation. Although the connectivity relation is a property of the nodes, for readability in the figures we associate the information with the edges. Thus, each node is represented as a record of the form `[types layout count]`.

As in the case of the nodes, each edge contains a record that tracks additional information about the edge. The *offset* component indicates the offsets (labels) of the references that are abstracted by the edge. The number of references that the edge may represent is tracked with the *maxCut* property. The *interfere* property tracks the possibility that the edge represents references that interfere. Finally, we have a field *connto* which is a list of all the other edges/variables that the edge may be connected to according to the *connR* relation (we add a (`!`) for the edges in the list that represent references which *may* alias and a ($\sim$) if the edges represent single references that *must* alias). To simplify the figures if the *connto* field is empty we omit it entirely from the record in the figure. Since the variable edges always represent single references and the offset label is implicitly the name of the variable the record simply contains the *connR* information or is omitted entirely if the *connR* relation is empty. To simplify the discussion of the examples each edge also has a unique label. The pointer edges in the figures are represented as records {`label offset maxCut interfere connto`}.

The abstract heap domain is restricted via a normal form [14,15]. The normal form ensures that the heap graph remains finite, and that equality comparisons are efficient. The local data flow analysis is performed using a *Hoare (Partially Disjunctive) Power Domain* [13,26] over these graphs. Interprocedural analysis is performed in a context-sensitive manner and the procedure analysis results are memoized. At each call/return site the portion of the heap graphs passed to the call are joined into a single graph. The design of the join operation is such that, in general, information lost in the join can be recovered when needed later in the program. The decision to perform joins at call sites (programs tend to have uniform expectations of the portion of the heap passed to and returned from calls) and to perform the join only on the portion of the heap passed to the called method results in very little loss of precision while ensuring the abstract model remains compact.

Abstract Call Stack. Our concrete model for the *call stack* is a function $S_m : (LV \times \mathbb{N}) \mapsto O$, where LV is the set of local variable names and $\mathbb{N}$ represents the depth in the call sequence (main is at depth 1) and O is the set of all live objects. Thus, the pair $(\mathtt{v}, 4)$ refers to the value of the variable $\mathtt{v}$ in the scope of the 4^{th} call frame.

To represent the concrete call stack we introduce *stack variables* which represent the values of local variables on the stack (for a variation on this approach see [22]). In our extension each *stack variable* summarizes all the possible targets (in a given graph) for a given variable name on the stack. Given a variable name $\mathtt{v}$ and a heap graph G we define a variable name $\mathtt{v'}$ for use in the abstract domain (we will select a better naming scheme in Section 4) where: $\mathtt{v'}$ is the abstraction of all the variables in the call stack, $\exists i \in \mathbb{N},$ node $n \in G,$ object o_n s.t. $o_n \in \gamma(n) \wedge S_m(\mathtt{v}, i) = o_n$.

By associating the set of stack locations that are abstracted with the set of targets in a given abstract heap graph, we can naturally partition the *stack variables* along with the heap graphs. Since each *stack variable* is associated with only the values on the stack that point into a region of the heap represented by the given heap graph, it is straightforward to partition and join them when partitioning the heap graphs.

Thus, during the local analysis the heap graph represents the portion of the program heap that is visible from the local variables and is augmented with some number of *stack variables* and *cutpoint variables* which relate variable values and the heap in the caller scope to the portions of the heap reachable from callee scope local variables.

For efficiency and in order to ensure analysis termination the naming scheme we choose will result in situations where multiple cutpoint (or stack) edges are given the same name. This may result in some amount of information loss (particularly with respect to reachability and aliasing). To minimize the loss that occurs we introduce an instrumentation domain for the stack/cutpoint variable edges, *nameColl* = {*pdj*, *pua*, *pa*}. Where *pdj* indicates a cutpoint/stack name representing (a single edge) or edges where the edges do not represent any pairwise *connected* references, *pua* indicates a name representing multiple edges where there are no pairwise *aliases*, while *pa* is the indicates the name represents edges that they may have pairwise *aliasing*. Thus, the cutpoint variable edges are represented with records {`maxCut interfere connto nameColl`} (stack variables are not used in this example).

4 Stack Variables, Cutpoint Labels

When performing the project operation in heaps with cutpoints we need to name the *stack variables* as well as the *cutpoint* edges. We use a simple technique for the stack variables: given a variable name `v` defined in the caller function `fcaller` we use the name `$fcaller*v` to represent this variable in the callee scope. This naming scheme can create false dependencies on the local scope names unless the variable information is normalized during the comparisons of entries in the memo-table.

Naming edges that cross the cutpoints is more complex since we need to balance the accuracy of the analysis with the potential of introducing spurious differences resulting from isomorphic (or nearly so) cutpoint edges being given different names. For the renaming of the cutpoint edges we assume that special names for the arguments to the function have been introduced. The first pointer parameter is referred to by the special variable name `p1` and the i^{th} pointer argument is referred to by the variable `pi`.

Figure 2(c) shows a recursive call to `LInit` where the special argument name `p1` has been added to represent the value of the first argument to the function. In this figure the edge *e1* is a cutpoint edge since it starts in the portion of the heap that is unreachable from the argument variables and ends in a portion of the heap that is reachable from the argument variables (this differs slightly from the definition for cutpoints in [21] but allows us to handle edges uniformly).

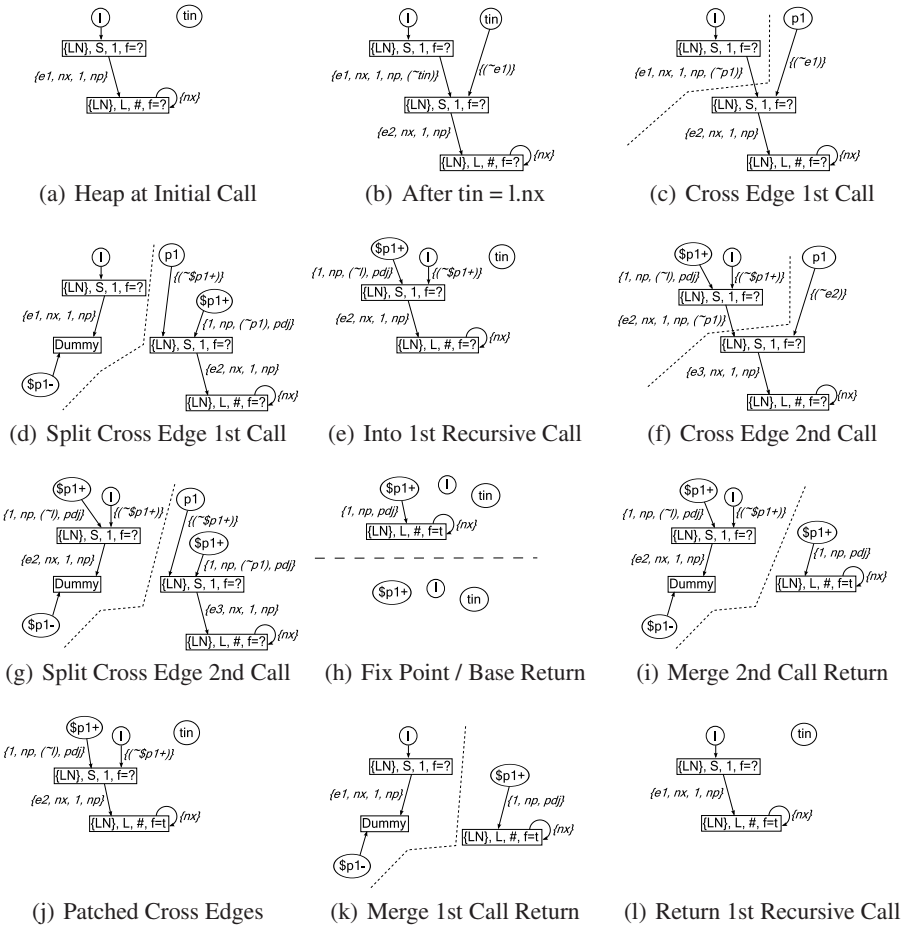
For each cutpoint edge we generate a pair of names: one is used in the unreachable section of the heap graph and one in the reachable section, which allows an abstract heap model to represent both incoming and outgoing cutpoint edges that are isomorphic and exist in the same abstract heap component without loss of precision.

If we are adding a cutpoint for the method call `fcaller` and the edge *e*, which is a cutpoint, starting at *n* and ending at n', and has edge label `fe`. We can find the shortest path (`f1 ... fk`) from any of the `pi` variables to n' (using lexographic comparison on the path names to break ties). Using the `pi` argument variable and the path (`f1 ... fk`) we derive the cutpoint `basename` = `fcaller*pi*f1*...*fk*fe` We compute a pair of static names (*unreachN*, *reachN*) where *unreachN* = `$basename-` and *reachN* = `$basename+`. In Figure 2(d) the cutpoint name `$p1+` (for brevity we simply label the cutpoint with the `pi` variable) is used to represent the endpoint of the cutpoint edge in the reachable component of the heap and `$p1-` to track a dummy node associated with the cutpoint edge in the unreachable component of the heap.

5 Example

The example program, Figure 1, recursively initializes the `f` fields in a linked list to the value `true`. Figure 2(a) shows the abstract heap model at the entry of the first call to the procedure (for simplicity we ignore any caller scope variables).

In Figure 2(a), variable `l` refers to a node that represents `LNode` objects (*types* = {`LNode`}, abbreviated to `LN`), that represents a region with no internal connections (*Layout* = *S*), which contains a single object (*count* = 1), and where all the incoming edges represent disjoint pointers (the `connto` lists on the edges are omitted). In this figure we also have that the elements in the list have unknown truth values in the `f`



(a) Heap at Initial Call (b) After tin = l.nx (c) Cross Edge 1st Call

(d) Split Cross Edge 1st Call (e) Into 1st Recursive Call (f) Cross Edge 2nd Call

(g) Split Cross Edge 2nd Call (h) Fix Point / Base Return (i) Merge 2nd Call Return

(j) Patched Cross Edges (k) Merge 1st Call Return (l) Return 1st Recursive Call

Fig. 2. Recursive Calls

fields (`f=?`). There is a single edge out of this node representing pointers stored in the `nx` field of the object represented by the node. This edge represents a single pointer (*maxCut* = 1) and all the pointers are non-interfering (*interfere* = *np*). Finally, this edge refers to a node that also represents `LNode` objects but may represent many of these objects (*count* = #) and, since the *Layout* value is *List*, we know that the objects may be connected in a list-like shape. Since there is a single incoming edge and it represents a single pointer, we can safely assume that this edge refers to the head of the list structure.

Figure 2(b) shows the abstract heap model just after executing the statement `tin = l.nx`. Since we know that *e1* refers to the head element of the list from Figure 2(a) we replaced the single *List*-shaped node with a node representing the unique head element and a node representing the tail of the list. Since the head element is unique we set the *count* of this new node to 1. Additionally, the only possible layout for a node of *count* 1 is *Singleton*. Finally, if a node represents a single object then all the outgoing field edges

can each represent a single pointer. Thus, we set the outgoing edge to have a *maxCut* = 1. Also note that after the load the analysis has determined that `tin` and `e1` must alias (indicated by the ∼`e1` and ∼`tin` entries in the connectivity lists).

Figure 2(c) shows the state of the abstract heap at the entry of the *project* procedure. The special name *p1* has been added to represent the value of the first pointer argument to the function and we have added a dotted line to indicate the reachable and unreachable portions of the heap. Note that the edge *e1* is a cutpoint edge according to our definition.

The result of the project operation is shown in Figure 2(d). The *e1* edge, which was a cutpoint edge for the call, has been remapped to a dummy node and the static cutpoint names `$p1-` and `$p1+` (for brevity we omit the procedure name and edge labels from the static names) have been introduced at the dummy node and at the target of this edge in the reachable section. Since this cutpoint edge only represents the single cutpoint edge generated in this call frame `nameColl = pdj`. Also note that the analysis has determined that the formal parameter `p1` must alias the cutpoint edge `$p1+`.

Figure 2(e) shows the resulting abstract heap that is passed into the callee scope for analysis. Since all the local variables in the caller scope either did not refer to nodes in the callee reachable section or are dead after the call return we do not have to give them stack names and can remove them entirely from the heap model. Figure 2(f) shows the abstract heap at the entry to the project function for the second recursive call. Again we have a cutpoint edge *e2*. Note that the reachable cutpoint label, `$p1+` introduced in the previous call is now in the unreachable portion of the heap, thus (`$p1+`) does not conflict with the unreachable name added in this call (`$p1-`). The result of the project operation is shown in Figure 2(g).

Figure 2(h) shows the eventual fixpoint approximation (above the dotted line) of the analysis of this function and also the base case return value (below the dotted line). Notice in the base case return value we were able to determine that the test `l == null` implies that `l` must be null and since we preserved must alias information through the cutpoint introduction we can infer that `l` must alias `$p1+`, which implies the cutpoint edge (`$p1+`) must also be null. Thus, the analysis can infer that on return the cutpoint edge is either `null` or is non-null and refers to some list in which all the `f` fields have been set to `true` (`f=t` in the figure).

In Figure 2(i) we show how the fixpoint approximation for the reachable section of the heap is recombined with the unreachable section of the heap using the *extend* operation. After the recombination we get the abstract heap model shown in Figure 2(j). In Figure 2(i) we have unioned the graphs and are ready to patch up the cutpoint cross edge information. The static name `$p1+` in the reachable portion of the heap has been used to compute the associated unreachable name (`$p1-`). Then the algorithm identifies the edge associated with the dummy node referred to by `$p1-` (*e2*) and remapped this edge to end at the target of `$p1+` (`tin` has been nullified since it is dead).

Figure 2(k) shows the *extend* operation at the return from the first recursive call which is similar to the situation in the second recursive call. The resulting abstract heap is shown in Figure 2(l) which can be joined with the result of the base case test and then completes the analysis of the method. As desired, the analysis has determined that the recursive list initialize procedure preserves the list shape of the argument list and that all of the `f` fields in the list have been set to `true` (`f=t` in the figures).

6 Project and Extend Algorithms

Project. We assume that before the *projectHeap* function is invoked all of the special argument variable names have been added to the heap model. This allows *projectHeap* (Algorithm 1 below) to easily compute the section of the heap model that is reachable in the callee procedure and then compute the set of nodes that comprise the unreachable portion of the heap model.

Algorithm 1. projectHeap

input : h: the heap model to be partitioned
output: h_r, h_u: the reachable and unreachable partitions, *snu*, *ncs*: the static names used and newly created
reachNodes ← set of nodes reachable from args;
unreachNodes ← set of nodes unreachable from args;
crossEdges ← set of edges that start in *unreachNodes* and end in *reachNodes*;
snu ← ∅;
ncs ← ∅;
foreach edge *e* in *crossEdges* **do**
    (sn, isnew) ← procCrossEdge(*h*, *e*, *reachNodes*);
    snu.add(*sn*);
    if *isnew* **then** *ncs*.add(*sn*);

h_u ← subgraph of h on the nodes *unreachNodes* ∪ {dummy nodes from procCrossEdge};
h_r ← subgraph of h on the nodes *reachNodes*;
return *(*h_r*,* h_u*,* snu, ncs*)*;

For each edge that crosses from the unreachable section into the reachable section we add a pair of static names to represent the edge (Algorithm 2). Since the heap model stores a number of domain properties in each edge, we create a dummy node and remap the edge to end at this node. Then, the *unreachN* static name is set to refer to this dummy node. In the reachable portion of the heap graph we simply set the *reachN* static name to refer to the target of the cross edge.

When adding the *reachN* static name to the reachable section of the heap graph the name may or may not already be present in the heap graph. If the name is not present then we add it to the static name map and for later use we note that this is the call where the name is introduced. Otherwise a name collision has occurred and we must mark the edges representing the possible cutpoints appropriately (for simplicity we mark all the edges). If there may be aliasing we note that the cutpoints from different frames may have aliasing targets (*pa*) and similarly if the new cutpoint edge may be connected with an existing cutpoint edge we mark them as being pairwise connected (*pua*). The functions *makeEdgeForUnreachCutpoint* and *makeEdgeForReachCutpoint* are used to produce edges to represent the cutpoint (based on the static name and the cutpoint edge properties) in the unreachable and reachable portions of the heap.

Once all of the cutpoint edges have been replaced by the required static names, the heap can be transformed into the unreachable version (where all the nodes in the reachable section and all the variables/static names that only refer to reachable nodes have

been removed) and the reachable version (where the nodes in the unreachable section and the associated names have been removed).

Algorithm 2. procCrossEdge

input : h: the heap, e: the cross edge, *reachNodes*: set of reachable nodes
output: *rsn*: the name used, *isnew*: true if *rsn* a new name
$n_e \leftarrow$ the node e ends at;
$n_i \leftarrow$ new dummy node;
(ursn, rsn) $\leftarrow$ genStaticNamePairForEdge(h, e);
$e_u \leftarrow$ makeEdgeForUnreachCutpoint(e, *ursn*);
set endpoint of e_u to n_i;
add e_u as an edge for *ursn*;
$e_r \leftarrow$ makeEdgeForReachCutpoint(e, *rsn*);
set endpoint of e_r to n_e;
remap the endpoint of e to n_i;
if the name *rsn* exists and has edges pointing to a node in *reachNodes* **then**
    rsnes $\leftarrow \{e' | e'$ is an edge for the cutpoint var *rsn*$\}$;
    add e_r as an edge for *rsn*;
    if e_r *is inConnected with an edge in* rsnes **then** set edges in *rsnes* and e_r to *pua*;
    if e_r *may alias with an edge in* rsnes **then** set edges in *rsnes* and e_r to *pa*;
    return *(*rsn*,* false*)*;
else
    add the name *rsn* to h;
    add e_r as an edge for *rsn*;
    return *(*rsn*,* true*)*;

Extend. After the call return we need to rejoin the unreachable portion of the heap that we extracted before the procedure call entry with the result we obtained from analyzing the callee procedure. This is done by looking at each of the static names that was used to represent a cutpoint edge and reconnecting as required. Then, each of the newly introduced cutpoint names can be removed from the heap model. The pseudo-code to do this is shown in Algorithm 3.

This algorithm merges all edges with the same reachable cutpoint name so that there is at most one target edge for a given cutpoint name in the reachable heap h_r (this simplifies the algorithm and is in our experience is quite accurate). The algorithm then pairs up the two cutpoint names and remaps the edge we saved in the unreachable section to the target node in the reachable section subject to a number of tests to propagate sharing information (the nullity information is propagated due to the fact that the dummy node and all incoming edges are always removed but the foreach loop on the targets of *ursn* does not execute since the target set is empty). The e_r.*nameColl* = *pua* test is true if this edge represents sets of pointers that do not have pairwise aliases. Thus, we mark the newly remapped edge and e_r as pairwise unaliased. Similarly, the e_r.*nameColl* = *pdj* test is true if this edge represents cutpoint/stack edges that are pairwise disjoint. Thus, we mark the newly remapped edge and e_r as pairwise disjoint.

Algorithm 3. extendHeap

```
input  : h_r, h_u: the reachable and unreachable partitions, snu, ncs: the static names used and
         newly created
output: h: the joined heap model
h ← new heap();
h.heapGraph ← mergeGraphs(h_r.heapGraph, h_u.heapGraph);
foreach static name sn in snu do
    ursn ← reachNameToUnreachName(sn);
    n_r ← the target of sn in h_r.nameMap;
    foreach node n_u that is a target of ursn in h_u.nameMap do
        e_r ← the single incoming edge to n_u;
        remap e_r to end at the target of n_r;
        e_r.interfere = e_r.interfere ⊔ n_r.interfere;
        if e_r.nameColl = pua then set e_r and n_r as unaliased;
        if e_r.nameColl = pdj then set e_r and n_r as disjoint;

    h_u.removeNodeAllEdges(target of ursn);
    h_u.unmapStaticName(ursn);
    if sn in ncs then h_r.unmapStaticName(sn);

h.nameMap ← mergeNameMaps(h_r.nameMap, h_u.nameMap);
return h
```

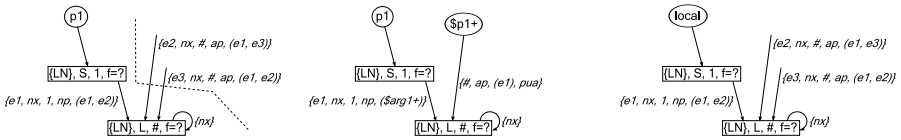
The major components of this algorithm are the separation of the *mergeGraphs* action from the *mergeNameMaps* action and the elimination of the static cutpoint edge names that were introduced for this call.

The *mergeGraphs* function computes the union of the graph structures that represent the abstract heap objects, while the *mergeNameMaps* function computes the union of the name maps (which are maps from the stack/variable/cutpoint names to the nodes in the graph structure that represent them). This separation allows the algorithm to nullify the names created for this call which prevents the propagation of unneeded cutpoint edge targets to the caller scope. The function *unmapStaticName* is used to eliminate a given static name from the abstract heap model name map.

Example Name Collision. The introduction of the *nameColl* domain minimizes the precision loss that occurs when a cutpoint or stack variable name collision occurs. Figure 3 shows an example of such a situation. In this figure we show part of a heap where the edges *e2* and *e3* are both cutpoint edges and they do not represent any pairwise aliasing pointers (no ! in the *connTo* lists) although they each represent sets of pointers that may alias, *interfere* = *ap*.

In this example our naming scheme will result in *e2* and *e3* being represented with the same cutpoint name. However, our method will mark this cutpoint edge as *nameColl* = *pua* (Figure 3(b)). This means that on return the *extend* algorithm will set the edges that are mapped to this cutpoint as being pairwise unaliased (Figure 3(c)) as desired. Thus, even though there was a name collision for the cutpoints we avoided (in this case completely) the loss of sharing information about the heap.

256 M. Marron et al.



(a) Colliding Names (b) To Same Cutpoint (c) PUA on Return

Fig. 3. Name Collision

7 Experimental Results

The proposed approach has been implemented and the effectiveness and efficiency of the analysis have been evaluated on the source code for programs from a variation of the Jolden [3,18] suite and several programs from SPEC JVM98 [27] (raytrace, modified to be single threaded, db and compress). The analysis algorithm is written in C++ and was compiled using MSVC 8.0. The parallelization benchmarks were run using the Sun 1.6 JVM. All runs are from our 2.8 GHz PentiumD machine with 1 GB of RAM.

We ran the analysis with the project/extend operations enabled (the *Project* column) and disabled (the *No-Project* column) and recorded the analysis time, the average number of times a method needed to be analyzed, and used the resulting shape information to parallelize the programs, shown in Figure 4. The results indicate that the project/extend operations have a significant impact on the performance of the analysis, reducing the number of contexts that each function needs to be analyzed in (on average reducing the number of contexts by a factor of 4.3) which results in a substantial decrease in analysis times (by a factor of 18.4). As expected this reduction becomes more pronounced as the size and complexity of the benchmarks increases, in the case of raytrace the analysis time without the project/extend operation is impractically large (772.6 seconds) but when we use the project/extend operations the analysis time is reduced to 35.11 seconds.

We used the shape information from the analysis to drive the parallelization of the benchmarks by using multiple threads in loops and calls, resulting in the speedup columns in Figure 4. Given the shape information produced by the analysis it is straight forward to compute what parts of the heap are read and written by a loop body or method call and thus which loops and calls can be executed in parallel (in raytrace we treated the memoization of intersect computations as spurious dependencies). Once the analysis identified locations that could be parallelized we inserted calls to a simple thread pool (since our current work is focused on the analysis this is done by hand but can be fully automated [6,23,10]). In 8 of 9 benchmarks that are suitable for shape driven parallelization (compress, db and mst do not have any data structure operations that are amenable to shape driven parallelization) we achieve a promising speedup, averaging a factor of 1.69 over the benchmarks.

Our experimental results show that the information provided by the analysis can be effectively used (in conjunction with existing techniques) to drive the parallelization of programs. To the best of our knowledge this analysis is the only shape analysis that is able to provide the information required to perform shape driven parallelization for five of these benchmarks (em3d, health, voronoi, bh and raytrace). Given the speed with

Benchmark Info			No-Project			Project		
Benchmark	Stmt	Method	Time	Avg Cont.	Speedup	Time	Avg Cont.	Speedup
bisort	260	13	0.86s	10.6	1.00	0.28s	1.9	1.72
em3d	333	13	0.12s	2.5	1.75	0.08s	1.8	1.75
mst	457	22	0.06s	3.2	NA	0.04s	3.0	NA
tsp	510	13	1.51s	22.4	1.84	0.17s	7.0	1.84
perimeter	621	36	54.57s	105.9	1.00	2.97s	50.2	1.00
health	643	16	3.24s	12.9	1.00	2.26s	4.2	1.76
voronoi	981	63	20.89s	61.4	1.00	2.67s	37.2	1.68
power	1352	29	5.71s	26.8	1.93	0.17s	1.3	1.93
bh	1616	51	8.64s	32.8	1.75	2.68s	7.3	1.75
compress	1102	41	0.29s	2.9	NA	0.18s	2.2	NA
db	1214	30	0.94s	3.7	NA	0.68s	2.8	NA
raytrace	3705	173	772.60s	293.1	1.00	35.11s	15.6	1.76
Overall	12794	523	869.43s	48.2	1.36	47.29s	11.2	1.69

Fig. 4. The Stmt and Method columns list the number of statements and methods for each benchmark. The columns for the No-Project and Project variations of the analysis list: the analysis time in seconds, the average number of times each method was analyzed and parallel speedup achieved on a 2 core 2.8 GHz PentiumD processor.

which the analysis is able to produce the information needed for the parallelization and the consistent parallel speedup that is obtained in the benchmarks (1.69 over all of the benchmarks and 1.77 if we exclude the benchmark mst), we find the results encouraging.

Of particular interest is the raytrace benchmark. This program is 2-4 times larger than any benchmarks used in the related work, builds and traverses several heap structures that have significant sharing between components. It also makes heavy use of virtual methods and recursion. This benchmark presents significant challenges in terms of the complexity and size of the program as well as in terms of the range of heap structures that need to be represented in order to accurately and efficiently analyze the program. Our analysis is able to manage all of these aspects and is able to produce a precise model of the heap (allowing us to obtain a speedup of 1.76 using heap based parallelization techniques). Further, the analysis is able to produce this result while maintaining a tractable analysis runtime.

8 Conclusion

We presented and benchmarked project/extend operations for a store-based heap model that is capable of precisely representing a range of shape, connectivity and sharing properties. The project and extend operations we introduced are designed to minimize the analysis time by reducing the number of unique calling contexts for each function and to minimize the imprecision introduced by the collisions that occur between stack/cutpoint names.

Our experimental results using the project/extend operations are very positive. The analysis was able to efficiently analyze benchmarks that build and manipulate a variety

of data structures. Our benchmark set includes a number of kernels that were originally designed as challenge problems for automatic parallelization (the Jolden suite) and several benchmarks from the SPEC JVM98 suite (including a single threaded version of raytrace). Our experimental results demonstrate that the project/extend operations are effective in minimizing the number of contexts that need to be analyzed (on average a factor of 4.3 reduction), improving analysis accuracy (seen as improved parallelization results, in 4 out of 12 benchmarks) and substantially reducing the analysis runtime (by a factor of nearly 20). Our heap analysis was also able to provide sufficient information to successfully parallelize the majority of benchmarks we examined, including several that cannot be successfully analyzed/parallelized using other proposed shape analysis methods.

Acknowledgments

This work is supported under subcontract R7A824-79200004 from the Los Alamos Computer Science Institute and Rice University and by the National Science Foundation (grant 0540600). Manuel Hermenegildo is also supported by the Prince of Asturias Chair at UNM, and projects MEC-MERIT, CAM-PROMESAS, and EU-MOBIUS.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Bruynooghe, M.: A Practical Framework for the Abstract Interpretation of Logic Programs. J. Log. Program 10, 91–124 (1991)
3. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in Java. In: PACT (2001)
4. Chong, S., Rugina, R.: Static analysis of accessed regions in recursive data structures. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 463–482. Springer, Heidelberg (2003)
5. Ghiya, R., Hendren, L.J.: Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: POPL (1996)
6. Ghiya, R., Hendren, L.J., Zhu, Y.: Detecting parallelism in C programs with recursive data structures. In: Koskimies, K. (ed.) CC 1998. LNCS, vol. 1383, pp. 159–173. Springer, Heidelberg (1998)
7. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
8. Gulwani, S., Tiwari, A.: An abstract domain for analyzing heap-manipulating low-level software. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 379–392. Springer, Heidelberg (2007)
9. Guo, B., Vachharajani, N., August, D.: Shape analysis with inductive recursion synthesis. In: PLDI (2007)
10. Hendren, L.J., Nicolau, A.: Parallelizing programs with recursive data structures. IEEE TPDS 1(1) (1990)
11. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: POPL (2001)

12. Jeannet, B., Loginov, A., Reps, T.W., Sagiv, S.: A relational approach to interprocedural shape analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 246–264. Springer, Heidelberg (2004)
13. Manevich, R., Sagiv, S., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
14. Marron, M., Kapur, D., Stefanovic, D., Hermenegildo, M.: A static heap analysis for shape and connectivity. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) KSEM 2006. LNCS, vol. 4382, pp. 345–363. Springer, Heidelberg (2007)
15. Marron, M., Majumdar, R., Stefanovic, D., Kapur, D.: Dominance: Modeling heap structures with sharing. Tech. report, CS Dept., Univ. of New Mexico (August 2007)
16. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL (2004)
17. Muthukumar, K., Hermenegildo, M.V.: Compile-time derivation of variable dependency using abstract interpretation. J. Log. Program (1992)
18. Modified Jolden Benchmarks (August 2007), `http://www.cs.unm.edu/~marron`
19. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
20. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: LICS (2002)
21. Rinetzky, N., Bauer, J., Reps, T.W., Sagiv, S., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL (2005)
22. Rinetzky, N., Sagiv, S.: Interprocedural shape analysis for recursive programs. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 133–149. Springer, Heidelberg (2001)
23. Rugina, R., Rinard, M.C.: Automatic parallelization of divide and conquer algorithms. In: PPOPP (1999)
24. Sagiv, S., Reps, T.W., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. In: POPL (1996)
25. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL (1999)
26. Smyth, M.B.: Power domains and predicate transformers: A topological view. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 662–675. Springer, Heidelberg (1983)
27. Standard Performance Evaluation Corporation. JVM98 Version 1.04 (August 1998), `http://www.spec.org/osg/jvm98/jvm98/doc/index.html`
28. Wilhelm, R., Sagiv, S., Reps, T.W.: Shape analysis. In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 1–17. Springer, Heidelberg (2000)

Precise Set Sharing Analysis for Java-Style Programs

Mario Méndez-Lojo[1] and Manuel V. Hermenegildo[1,2]

[1] University of New Mexico (USA)
[2] Technical University of Madrid (Spain)

Abstract. Finding useful *sharing* information between instances in object-oriented programs has recently been the focus of much research. The applications of such static analysis are multiple: by knowing which variables definitely do not share in memory we can apply conventional compiler optimizations, find coarse-grained parallelism opportunities, or, more importantly, verify certain correctness aspects of programs even in the absence of annotations. In this paper we introduce a framework for deriving precise sharing information based on abstract interpretation for a Java-like language. Our analysis achieves precision in various ways, including supporting multivariance, which allows separating different contexts. We propose a combined *Set Sharing + Nullity + Classes* domain which captures which instances do not share and which ones are definitively null, and which uses the classes to refine the static information when inheritance is present. The use of a set sharing abstraction allows a more precise representation of the existing sharings and is crucial in achieving precision during interprocedural analysis. Carrying the domains in a combined way facilitates the interaction among them in the presence of multivariance in the analysis. We show through examples and experimentally that both the set sharing part of the domain as well as the combined domain provide more accurate information than previous work based on pair sharing domains, at reasonable cost.

1 Introduction

The technique of Abstract Interpretation [8] has allowed the development of sophisticated program analyses which are at the same time provably correct and practical. The semantic approximations produced by such analyses have been traditionally applied to high- and low-level optimizations during program compilation, including program transformations. More recently, promising applications of such semantic approximations have been demonstrated in the more general context of program development, such as verification and static debugging.

Sharing analysis [14,20,26] aims to detect which variables do not share in memory, i.e., do not point (transitively) to the same location. It can be viewed as an abstraction of the graph-based representations of memory used by certain classes of alias analyses (see, e.g., [31,5,13,15]). Obtaining a safe (over-) approximation of which instances might share allows parallelizing segments of code,

F. Logozzo et al. (Eds.): VMCAI 2008, LNCS 4905, pp. 172–187, 2008.

improving garbage collection, reordering execution, etc. Also, sharing information can improve the precision of other analyses.

Nullity analysis is aimed at keeping track of null variables. This allows for example verifying properties such as the absence of null-pointer exceptions at compile time. In addition, by combining sharing and null information it is possible to obtain more precise descriptions of the state of the heap.

In type-safe, object-oriented languages *class* analysis [1,3,10,22], (sometimes called *type* analysis) focuses on determining, in the presence of polymorphic calls, which particular implementation of a given method will be executed at runtime, i.e., what is the specific class of the called object in the hierarchy. Multiple compilation optimizations benefit from having precise class descriptions: inlining, dead code elimination, etc. In addition, class information may allow analyzing only a subset of the classes in the hierarchy, which may result in additional precision.

We propose a novel analysis which infers in a combined way *set sharing*, *nullity*, and *class* information for a subset of Java that takes into account most of its important features: inheritance, polymorphism, visibility of methods, etc. The analysis is multivariant, based on the algorithm of [21], which allows separating different contexts, thus increasing precision. The additional precision obtained from context sensitivity has been shown to be important in practice in the analysis of object-oriented programs [30].

The objective of using a reduced cardinal product [9] of these three abstract domains is to achieve a good balance between precision and performance, since the information tracked by each component helps refine that of the others. While in principle these three analyses could be run separately, because they interact (we provide some examples of this), this would result in a loss of precision or require an expensive iteration over the different analyses until an overall fixpoint is reached [6,9]. In addition note that since our analysis is multivariant, and given the different nature of the properties being tracked, performing analyses separately may result in different sets of abstract values (contexts) for each analysis for each program point. This makes it difficult to relate which abstract value of a given analysis corresponds to a given abstract value of another analysis at a given point. At the other end of things, we prefer for clarity and simplicity reasons to develop directly this three-component domain and the operations on it, rather than resorting to the development of a more unified domain through (semi-)automatic (but complex) techniques [6,7]. The final objectives of our analysis include verification, static debugging, and optimization.

The closest related work is that of [26] which develops a *pair*-sharing [27] analysis for object-oriented languages and, in particular, Java. Our description of the (set-)sharing part of our domain is in fact based on their elegant formalization. The fundamental difference is that we track *set* sharing instead of *pair* sharing, which provides increased accuracy in many situations and can be more appropriate for certain applications, such as detecting independence for program parallelization. Also, our domain and abstract semantics track additionally nullity and classes in a combined fashion which, as we have argued above, is

$$
\begin{array}{lll}
\mathit{prog} & ::= & \mathit{class\_decl}^* \\
\mathit{class\_decl} & ::= & \texttt{class}\ \mathrm{k}_1\ [\texttt{extends}\ \mathrm{k}_2]\ \mathit{decl}^*\ \mathit{meth\_decl}^* \\
\mathit{meth\_decl} & ::= & \mathit{vbty}\ (\mathrm{t}_{ret}|\texttt{void})\ \mathrm{meth}\ \mathit{decl}^*\ \mathit{com} \\
\mathit{vbty} & ::= & \texttt{public} \mid \texttt{private} \\
\mathit{com} & ::= & \mathrm{v} = \mathit{expr} \mid \mathrm{v}.f = \mathit{expr} \\
 & & \mid \mathit{decl} \mid \texttt{skip} \\
 & & \mid \texttt{return}\ \mathit{expr} \mid \mathit{com};\mathit{com} \\
 & & \mid \texttt{if}\ \mathrm{v}\ (== \mid !=)\ (\texttt{null}|\mathrm{w})\ \mathit{com}\ \texttt{else}\ \mathit{com} \\
\mathit{decl} & ::= & \mathrm{v{:}t} \\
\mathit{var\_lit} & ::= & \mathrm{v} \mid \mathrm{a} \\
\mathit{expr} & ::= & \texttt{null} \mid \texttt{new}\ \mathrm{k} \mid \mathrm{v}.f \mid \mathrm{v.m}(\mathrm{v}_1, \ldots \mathrm{v}_n) \mid \mathit{var\_lit}
\end{array}
$$

Fig. 1. Grammar for the language

particularly useful in the presence of multivariance. In addition, we deal directly with a larger set of object features such as inheritance and visibility. Finally, we have implemented our domains (as well as the pair sharing domain of [26]), integrated them in our multivariant analysis and verification framework [17], and benchmarked the system. Our experimental results are encouraging in the sense that they seem to support that our contributions improve the analysis precision at reasonable cost.

In [23,24], the authors use a *distinctness* domain in the context of an abstract interpretation framework that resembles our sharing domain: if two variables point to different abstract locations, they do not share at the concrete level. Their approach is closer to shape analysis [25] than to sharing analysis, which can be inferred from the former. Although information retrieved in this way is generally more precise, it is also more computationally demanding and the abstract operations are more difficult to design. We also support some language constructs (e.g., visibility of methods) and provide detailed experimental results, which are not provided in their work.

Most recent work [28,18,30] has focused on context-sensitive approaches to the points-to problem for Java. These solutions are quite scalable, but flow-insensitive and overly conservative. Therefore, a verification tool based on the results of those algorithms may raise spurious warnings. In our case, we are able to express sharing information in a safe manner, as invariants that all program executions verify at the given program point.

2 Standard Semantics

The source language used is defined as a subset of Java which includes most of its object-oriented (inheritance, polymorphism, object creation) and specific (e.g., access control) features, but at the same time simplifies the syntax, and does not deal with interfaces, concurrency, packages, and static methods or variables. Although we support primitive types in our semantics and implementation, they will be omitted from the paper for simplicity.

```
class Element {
   int value;
   Element next;}

class Vector {
   Element first;

    public void add(Element el) {
        Vector v  = new Vector();
        el.next  = null;
        v.first = el;
        append(v);
    }
}
```

```
public void append(Vector v) {

  if (this != v) {
      Element e = first;
      if (e == null)
          first = v.first;
      else {
         while (e.next != null)
            e = e.next;
         e.next = v.first;
      }
  }
}
```

Fig. 2. Vector example

The rules for the grammar of this language are listed in Fig. 1. The `skip` statement, not present in the Java standard specification [11], has the expected semantics. Fig. 2 shows an example program in the supported language, an alternative implementation for the `java.util.Vector` class of the JDK in which vectors are represented as linked lists. Space constraints prevent us from showing the full code here,[1] although the figure does include the relevant parts.

2.1 Basic Notation

We first introduce some notation and auxiliary functions used in the rest of the paper. By $\mapsto$ we refer to total functions; for partial ones we use $\rightarrow$. The powerset of a set s is $\mathcal{P}(s)$; $\mathcal{P}^+(s)$ is an abbreviation for $\mathcal{P}(s) \setminus \{\emptyset\}$. The *dom* function returns all the elements for which a function is defined; for the codomain we will use *rng*. A substitution $f[k_1 \mapsto v_1, \ldots, k_n, \mapsto v_n]$ is equivalent to $f(k_1) = v_1, \ldots, f(k_n) = v_n$. We will overload the operator for lists so that $f[K \mapsto V]$ assigns $f(k_i) = v_i$, $i = 1, \ldots, m$, assuming $|K| = |V| = m$. By $f|_{\_S}$ we denote removing S from $dom(f)$. Conversely, $f|_S$ restricts $dom(f)$ to S. For tuples $(f_1, \ldots, f_m)|_S = (f_1|_S, \ldots, f_m|_S)$. Renaming in the set s of every variable in S by the one in the same position in T ($|S| = |T|$) is written as $s|^T_S$. This operator can also be applied for renaming single variables. We denote by $\mathcal{B}$ the set of Booleans.

2.2 Program State and Sharing

With $\mathcal{M}$ we designate the set of all method names defined in the program. For the set of distinct identifiers (variables and fields) we use $\mathcal{V}$. We assume that $\mathcal{V}$ also includes the elements *this* (instance where the current method is executed),

[1] Full source code for the example can be found in
`http://www.clip.dia.fi.upm.es/~mario`

and *res* (for the return value of the method). In the same way, $\mathcal{K}$ represents the program-defined classes. We do not allow `import` declarations but assume as member of $\mathcal{K}$ the predefined class `Object`.

$\mathcal{K}$ forms a lattice implied by a subclass relation $\downarrow$: $\mathcal{K} \rightarrow \mathcal{P}(\mathcal{K})$ such that if $t_2 \in \downarrow t_1$ then $t_2 \leq_{\mathcal{K}} t_1$. The semantics of the language implies $\downarrow\texttt{Object} = \mathcal{K}$. Given $def : \mathcal{K} \times \mathcal{M} \mapsto \mathcal{B}$, that determines whether a particular class provides its own implementation for a method, the Boolean function $redef : \mathcal{K} \times \mathcal{K} \times \mathcal{M} \mapsto \mathcal{B}$ checks if a class k_1 redefines a method existing in the ancestor k_2: $redef(k_1, k_2, \mathrm{m}) = true$ iff $\exists k$ s.t. $def(k, \mathrm{m})$, $k_1 \leq_{\mathcal{K}} k <_{\mathcal{K}} k_2$.

Static types are accessed by means of a function $\pi : \mathcal{V} \mapsto \mathcal{K}$ that maps variables to their declared types. The purpose of an *environment* π is twofold: it indicates the set of variables accessible at a given program point and stores their declared types. Additionally, we will use the auxiliary functions $F(k)$ (which maps the fields of $k \in \mathcal{K}$ to their declared type), and $type_\pi(expr)$, which maps expressions to types, according to π.

The description of the memory state is based on the formalization in [26,12]. We define a frame as any element of $Fr_\pi = \{\, \phi \mid \phi \in dom(\pi) \mapsto Loc \cup \{\mathrm{null}\}\}$, where $Loc = \mathbb{I}^+$ is the set of memory locations. A frame represents the first level of indirection and maps variable names to locations except if they are null. The set of all objects is $Obj = \left\{\, k \star \phi \mid k \in \mathcal{K}, \phi \in Fr_{F(k)} \right\}$. Locations and objects are linked together through the memory $Mem = \{\, \mu \mid \mu \in Loc \mapsto Obj\}$. A new object of class k is created as $new(k) = k \star \phi$ where $\phi(f) = null \ \ \forall f \in F(k)$. The object pointed to by v in the frame ϕ and memory μ can be retrieved via the partial function $obj(\phi \star \mu, v) = \mu(\phi(v))$. A valid heap configuration (concrete state $\phi \star \mu$) is any element of $\Sigma_\pi = \{\, (\phi \star \mu) \mid \phi \in Fr_\pi, \mu \in Mem\}$. We will sometimes refer to a pair $(\phi \star \mu)$ with δ.

The set of locations $R_\pi(\phi \star \mu, v)$ reachable from $v \in dom(\pi)$ in the particular state $\phi \star \mu \in \Sigma_\pi$ is calculated as $R_\pi(\phi \star \mu, v) = \cup \left\{\, R^i_\pi(\phi \star \mu, v) \;\middle|\; i \geq 0 \right\}$, the base case being $R^0_\pi(\phi \star \mu, v) = \{(\phi(v))|_{Loc}\}$ and the inductive one $R^{i+1}_\pi(\phi \star \mu, v) = \cup \left\{\, rng(\mu(l).\phi))|_{Loc} \mid l \in R^i_\pi(\phi \star \mu, v) \right\}$. Reachability is the basis of two fundamental concepts: sharing and nullity. Distinct variables $V = \{v_1, \ldots, v_n\}$ *share* in the actual memory configuration δ if there is at least one common location in their reachability sets, i.e., $share_\pi(\delta, V)$ is true iff $\cap_{i=1}^n R_\pi(\delta, v_i) \neq \emptyset$. A variable $v \in dom(\pi)$ is *null* in state δ if $R_\pi(\delta, v) = \emptyset$. Nullity is checked by means of $nil_\pi : \Sigma_\pi \times dom(\pi) \mapsto \mathcal{B}$, defined as $nil_\pi(\phi \star \mu, v) = true$ iff $\phi(v) = null$.

The *run-time type* of a variable in scope is returned by $\psi_\pi : \Sigma_\pi \times dom(\pi) \mapsto \mathcal{K}$, which associates variables with their dynamic type, based on the information contained in the heap state: $\psi_\pi(\delta, v) = obj(\delta, v).k$ if $\overline{nil_\pi}(\delta, v)$ and $\psi_\pi(\delta, v) = \pi(v)$ otherwise. In a type-safe language like Java runtime types are congruent with declared types, i.e., $\psi_\pi(\delta, v) \leq_{\mathcal{K}} \pi(v) \ \ \forall v \in dom(\pi), \forall \delta \in \Sigma_\pi$. Therefore, a correct approximation of ψ_π can always be derived from π. Note that at the same program point we might have different run-time type states ψ^1_π and ψ^2_π depending on the particular program path executed, but the static type state is unique.

Denotational (compositional) semantics of sequential Java has been the subject of previous work (e.g., [2]). In our case we define a simpler version of that semantics for the subset defined in Sect. 2, described as transformations in the frame-memory state. The descriptions are similar to [26]. Expression functions $\mathcal{E}^I_\pi[\![]\!] : expr \mapsto (\Sigma_\pi \mapsto \Sigma_{\pi'})$ define the meaning of Java expressions, augmenting the actual scope $\pi' = \pi[res \mapsto type_\pi(exp)]$ with the temporal variable res. Command functions $\mathcal{C}^I_\pi[\![]\!] : com \mapsto (\Sigma_\pi \mapsto \Sigma_\pi)$ do the same for commands; semantics of a method `m` defined in class k is returned by the function $I(k.\mathtt{m}) : \Sigma_{input(k.m)} \to \Sigma_{output(k.m)}$. The definition of the respective environments, given a declaration in class k as t_{ret} `m`$(this : k, p_1 : t_1 \ldots p_n : t_n)$ `com`, is $input(k.m) = \{this \mapsto k, p_1 \mapsto t_1, \ldots, p_n \mapsto t_n\}$ and $output(k.m) = input(k.m)[out \mapsto t_{ret}]$.

Example 1. Assume that, in Figure 2, after entering in the method `add` of the class `Vector` we have an initial state $(\phi_0 \star \mu_0)$ s.t. $loc_1 = \phi_0(el) \neq null$. After executing `Vector v = new Vector()` the state is $(\phi_1 \star \mu_1)$, with $\phi_1(v) = loc_2$, and $\mu_1(loc_2).\phi(first) = null$. The field assignment `el.next = null` results in $(\phi_2 \star \mu_2)$, verifying $\mu_2(loc_1).\phi(next) = null$. In the third line, `v.first = el` links loc_1 and loc_2 since now $\mu_3(loc_2).\phi(first) = loc_1$. Now v and el share, since their reachability sets intersect *at least* in $\{loc_1\}$. Finally, assume that `append` attaches v to the end of the current instance $this$ resulting in a memory layout $(\phi_4 \star \mu_4)$. Given $loc_3 = obj((\phi_4 \star \mu_4)(this)).\phi(first)$, it should hold that $\mu_4(\ldots \mu_4(loc_3).\phi(next) \ldots).\phi(next) = loc_2$. Now $this$ shares with v and therefore with el, because loc_1 is reachable from loc_2.

3 Abstract Semantics

An abstract state $\sigma \in D_\pi$ in an environment π approximates the sharing, nullity, and run-time type characteristics (as described in Sect. 2.2) of set of concrete states in Σ_π. Every abstract state combines three abstractions: a sharing set $sh \in \mathcal{DS}_\pi$, a nullity set $nl \in \mathcal{DN}_\pi$, and a type member $\tau \in \mathcal{DT}_\pi$, i.e., $D_\pi = \mathcal{DS}_\pi \times \mathcal{DN}_\pi \times \mathcal{DT}_\pi$.

The sharing abstract domain $\mathcal{DS}_\pi = \{\{v_1, \ldots, v_n\} \mid \{v_1, \ldots, v_n\} \in \mathcal{P}(dom(\pi)), \cap_{i=1}^n C_\pi(v_i) \neq \emptyset\}$ is constrained by a class reachability function which retrieves those classes that are reachable from a particular variable: $C_\pi(v) = \cup\{C^i_\pi(v) \mid i \geq 0\}$, given $C^0_\pi(v) = \downarrow\pi(v)$ and $C^{i+1}_\pi(v) = \cup\{rng(F(k)) \mid k \in C^i_\pi(v)\}$. By using class reachability, we avoid including in the sharing domain sets of variables which cannot share in practice because of the language semantics. The partial order $\leq_{\mathcal{DS}_\pi}$ is set inclusion.

We define several operators over sharing sets, standard in the sharing literature [14,19]. The binary union $\uplus : \mathcal{DS}_\pi \times \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$, calculated as $S_1 \uplus S_2 = \{Sh_1 \cup Sh_2 \mid Sh_1 \in S_1, Sh_2 \in S_2\}$ and the closure under union $* : \mathcal{DS}_\pi \mapsto \mathcal{DS}_\pi$ operators, defined as $S^* = \{\cup SSh \mid SSh \in \mathcal{P}^+(S)\}$; we later filter their results using class reachability. The relevant sharing with respect to v is $sh_v = \{s \in sh \mid v \in s\}$, which we overloaded for sets. Similarly, $sh_{-v} = \{s \in sh \mid v \notin s\}$. The projection $sh|_V$ is equivalent to $\{S \mid S = S' \cap V, S' \in sh\}$.

$$\mathcal{SE}^I_\pi[\![\texttt{null}]\!](sh, nl, \tau) = (sh, nl', \tau')$$
$$nl' = nl[res \mapsto null]$$
$$\tau' = \tau[res \mapsto \downarrow object]$$

$$\mathcal{SE}^I_\pi[\![\texttt{new } k]\!](sh, nl, \tau) = (sh', nl', \tau')$$
$$sh' = sh \cup \{\{res\}\}$$
$$nl' = nl[res \mapsto nnull]$$
$$\tau' = \tau[res \mapsto \{\kappa\}]$$

$$\mathcal{SE}^I_\pi[\![v]\!](sh, nl, \tau) = (sh', nl', \tau')$$
$$sh' = (\{\{res\}\} \uplus sh_v) \cup sh_{-v}$$
$$nl' = nl[res \mapsto nl(v)]$$
$$\tau' = \tau[res \mapsto \tau(v)]$$

$$\mathcal{SE}^I_\pi[\![v\texttt{.f}]\!](sh, nl, \tau) = \begin{cases} \bot \text{ if } nl(v) = null \\ (sh', nl', \tau') \text{ otherwise} \end{cases}$$
$$sh' = sh_{-v} \cup \bigcup\{\mathcal{P}^+(s|_{-v} \cup \{res\}) \uplus \{\{v\}\} \mid s \in sh_v\}$$
$$nl' = nl[res \mapsto unk, v \mapsto nnull]$$
$$\tau' = \tau[res \mapsto\downarrow F(\pi(v)(\texttt{f}))]$$

$$\mathcal{SE}^I_\pi[\![v\texttt{.m}(v_1, \ldots, v_n)]\!](sh, nl, \tau) = \begin{cases} \bot \text{ if } nl(v) = null \\ \sigma' \text{ otherwise} \end{cases}$$
$$\sigma' = \mathcal{SE}^I_\pi[\![\texttt{call}(v, m(v_1, \ldots, v_n))]\!](sh, nl', \tau)$$
$$nl' = nl[v \mapsto nnull]$$

Fig. 3. Abstract semantics for the expressions

The nullity domain is $\mathcal{DN}_\pi = \mathcal{P}(dom(\pi) \mapsto \mathcal{NV})$, where $\mathcal{NV} = \{null, nnull, unk\}$. The order $\leq_{\mathcal{NV}}$ of the nullity values ($null \leq_{\mathcal{NV}} unk$, $nnull \leq_{\mathcal{NV}} unk$) induces a partial order in $\mathcal{DN}_\pi$ s.t. $nl_1 \leq_{\mathcal{DN}_\pi} nl_2$ if $nl_1(v) \leq_{\mathcal{NV}} nl_2(v)$ $\forall v \in dom(\pi)$. Finally, the domain of types maps variables to sets of types congruent with π: $\mathcal{DT}_\pi = \{(v, \{t_1, \ldots, t_n\}) \in dom(\pi) \mapsto \mathcal{P}(\mathcal{K}) \mid \{t_1, \ldots, t_n\} \subseteq \downarrow\pi(v)\}$.

We assume the standard framework of abstract interpretation as defined in [8] in terms of Galois insertions. The concretization function $\gamma_\pi : D_\pi \mapsto \mathcal{P}(\Sigma_\pi)$ is $\gamma_\pi(sh, nl, \tau) = \{\delta \in \Sigma_\pi | \ \forall V \subseteq dom(\pi), share_\pi(\delta, V)$ and $\nexists W, V \subset W \subseteq dom(\pi)$ s.t. $share_\pi(\delta, W) \Rightarrow V \in sh$, and $R_\pi(\delta, v) = \emptyset$ if $nl(v) = null$, and $R_\pi(\delta, v) \neq \emptyset$ if $nl(v) = nnull$, and $\psi_\pi(\delta, v) \in \tau(v)$, $\forall v \in dom(\pi)\}$.

The abstract semantics of expressions and commands is listed in Figs. 3 and 4. They correctly approximate the standard semantics, as proved in [16]. As their concrete counterparts, they take an expression or command and map an input state $\sigma \in D_\pi$ to an output state $\sigma' \in D^\sigma_{\pi'}$ where $\pi = \pi'$ in commands and $\pi' = \pi[res \mapsto type_\pi(expr)]$ in expression $expr$. The semantics of a method call is explained in Sect. 3.1. The use of set sharing (rather than pair sharing) in the semantics prevents possible losses of precision, as shown in Example 2.

Example 2. In the `add` method (Fig. 2), assume that $\sigma = (\{\{this, el\}, \{v\}\}, \{this/nnull, el/nnull, v/nnull\})$ right before evaluating `el` in the third line (we skip type information for simplicity). The expression `el` binds to res the location of el, i.e., forces el and res to share. Since $nl(el) \neq null$ the new sharing is $sh' = (\{\{res\}\} \uplus sh_{el}) \cup sh_{-el} = (\{\{res\}\} \uplus \{\{this, el\}\}) \cup \{\{v\}\} = \{\{res, this, el\}, \{v\}\}$.

$$\mathcal{SC}^I_\pi[\![v\texttt{=}expr]\!]\sigma = ((sh'|_{-v})|^v_{res}, nl'|^v_{res}, \tau''|_{-res})$$
$$\tau'' = \tau'[v \mapsto (\tau'(v) \cap \tau'(res))]$$
$$(sh', nl', \tau') = \mathcal{SE}^I_\pi[\![expr]\!]\sigma$$

$$\mathcal{SC}^I_\pi[\![v\texttt{.f=}expr]\!]\sigma = (sh'', nl'', \tau')|_{-res}$$
$$sh'' = \begin{cases} \bot & \text{if } nl'(v) = null \\ sh' & \text{if } nl'(res) = null \\ sh^y \cup sh'_{-\{v,res\}} & \text{otherwise} \end{cases}$$
$$nl'' = nl'[v \mapsto nnull]$$
$$sh^y = (\bigcup\{\mathcal{P}(s|_{-v} \cup \{res\}) \uplus \{\{v\}\} \mid s \in sh'_v\} \cup \bigcup\{\mathcal{P}(s|_{-res} \cup \{v\}) \uplus \{\{res\}\} \mid s \in sh'_{res}\})^*$$
$$(sh', nl', \tau') = \mathcal{SE}^I_\pi[\![expr]\!]\sigma$$

$$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==null } com_1 \texttt{ else } com_2]\!]\sigma = \begin{cases} \sigma'_1 & \text{if } nl(v) = null \\ \sigma'_2 & \text{if } nl(v) = nnull \\ \sigma_1 \sqcup \sigma_2 & \text{if } nl(v) = unk \end{cases}$$
$$\sigma'_i = \mathcal{SC}^I_\pi[\![com_i]\!]\sigma$$
$$\sigma_1 = \mathcal{SC}^I_\pi[\![com_1]\!](sh|_{-v}, nl[v \mapsto null], \tau[v \mapsto \downarrow\pi(v)])$$
$$\sigma_2 = \mathcal{SC}^I_\pi[\![com_2]\!](sh, nl[v \mapsto nnull], \tau)$$

$$\mathcal{SC}^I_\pi[\![\texttt{if } v\texttt{==}w\ com_1 \texttt{ else } com_2]\!](sh, nl, \tau) = \begin{cases} \sigma'_1 & \text{if } nl(v) = nl(w) = null \\ \sigma'_2 & \text{if } sh|_{\{v,w\}} = \emptyset \\ \sigma'_1 \sqcup \sigma'_2 & \text{otherwise} \end{cases}$$
$$\sigma'_i = \mathcal{SC}^I_\pi[\![com_i]\!](sh, nl, \tau)$$

$$\mathcal{SC}^I_\pi[\![com_1\texttt{;}com_2]\!]\sigma = \mathcal{SC}^I_\pi[\![com_2]\!](\mathcal{SC}^I_\pi[\![com_1]\!]\sigma)$$

Fig. 4. Abstract semantics for the commands

In the case of pair-sharing, the transfer function [26] for the same initial state $sh = \{\{this, el\}, \{v, v\}\}$ returns $sh'_p = \{\{res, el\}, \{res, this\}, \{this, el\}, \{v, v\}\}$, which translated to set sharing results in $sh'' = \{\{res, el\}, \{res, this\}, \{res, this, el\}, \{this, el\}, \{v\}\}$, a less precise representation (in terms of $\leq_{\mathcal{DS}_\pi}$) than sh'.

Example 3. Our multivariant analysis keeps two different call contexts for the `append` method in the `Vector` class (Fig. 2). Their different sharing information shows how sharing can improve nullity results. The first context corresponds to external calls (invocation from other classes), because of the `public` visibility of the method: $\sigma_1 = (\{\{this\}, \{this, v\}, \{v\}\}, \{this/nnull, v/unk\}, \{this/\{vector\}, v/\{vector\}\})$. The second corresponds to an internal (within the class) call, for which the analysis infers that *this* and v do not share: $\sigma_2 = (\{\{this\}, \{v\}\}, \{this/nnull, v/unk\}, \{this/\{vector\}, v/\{vector\}\})$. Inside `append`, we avoid creating a circular list by checking that $this \neq v$. Only then is the last element of *this* linked to the first one of v. We use `com` to represent the series of commands `Element e = first; if (e==null)...else..` and `bdy` for the whole body of the method. Independently of whether the input state is σ_1 or σ_2 our analysis infers that $\mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_1 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_2 = (\{\{this, v\}\}, \{this/nnull, v/nnull\}, \{this/\{vector\}, v/\{vector\}\}) = \sigma_3$. However, the more precise sharing information in σ_2 results in a more precise analysis

Algorithm 1. Extend operation

input : state before the call σ, result of analyzing the call σ_λ and actual parameters A
output: resulting state σ_f

if $\sigma_\lambda = \bot$ **then**
  $\sigma_f = \bot$
else
  let $\sigma = (sh, nl, \tau)$, and $\sigma_\lambda = (sh_\lambda, nl_\lambda, \tau_\lambda)$, and $AR = A \cup \{res\}$

$$
\begin{aligned}
star &= (sh_A \cup \{\{res\}\})^* \\
sh_{ext} &= \{s \mid s \in star, s|_{AR} \in sh_\lambda\} \\
sh_f &= sh_{ext} \cup sh_{-A} \\
nl_f &= nl[res \mapsto nl_\lambda(res)] \\
\tau_f &= \tau[res \mapsto \tau_\lambda(res)] \\
\sigma_f &= (sh_f, nl_f, \tau_f)
\end{aligned}
$$

end

of `bdy`, because of the guard `(this!=v)`. In the case of the external calls, $\mathcal{SC}^I_\pi[\![\texttt{bdy}]\!]\sigma_1 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_1 \sqcup \mathcal{SC}^I_\pi[\![\texttt{skip}]\!]\sigma_1 = \sigma_1 \sqcup \sigma_3 = \sigma_1$. When the entry state is σ_2, the semantics at the same program point is $\mathcal{SC}^I_\pi[\![\texttt{bdy}]\!]\sigma_2 = \mathcal{SC}^I_\pi[\![\texttt{com}]\!]\sigma_2 = \sigma_3 < \sigma_1$. So while the internal call requires $v \neq null$ to terminate, we cannot infer the final nullity of that parameter in a public invocation, which might finish even if v is null.

3.1 Method Calls

The semantics of the expression $\texttt{call}(v, m(v_1, \ldots, v_n))$ in state $\sigma = (sh, nl, \tau)$ is calculated by implementing the top-down methodology described in [21]. We will assume that the formal parameters follow the naming convention F in all the implementations of the method; let $A = \{v, v_1, \ldots, v_n\}$ and $F = dom(input(k.m))$ be ordered lists. We first calculate the projection $\sigma_p = \sigma|_A$ and an entry state $\sigma_y = \sigma_p|^F_A$. The abstract execution of the call takes place only in the set of classes $K = \tau(v)$, resulting in an exit state $\sigma_x = \bigsqcup\{\mathcal{SC}^I_\pi[\![k'.m]\!]\sigma_y | k' = lookup(k, m), k \in K\}$, where *lookup* returns the body of k's implementation of m, which can be defined in k or inherited from one of its ancestors. The abstract execution of the method in a subset $K \subseteq \downarrow\pi(v)$ increases analysis precision and is the ultimate purpose of tracking run-time types in our abstraction. We now remove the local variables $\sigma_b = \sigma_x|_{F \cup \{out\}}$ and rename back to the scope of the caller: $\sigma_\lambda = \sigma_b|^{A \cup \{res\}}_{F \cup \{out\}}$; the final state σ_f is calculated as $\sigma_f = extend(\sigma, \sigma_\lambda, A)$. The $extend : D_\pi \times D_\pi \times \mathcal{P}(dom(\pi)) \mapsto D_\pi$ function is described in Algorithm 1.

In Java references to objects are passed by value in a method call. Therefore, they cannot be modified. However, the call might introduce new sharing between actual parameters through assignments to their fields, given that the formal parameters they correspond to have not been reassigned. We keep the original information by copying all the formal parameters at the beginning of each call,

as suggested in [23]. Those copies cannot be modified during the execution of the call, so a meaningful correspondence can be established between A and F.

We can do better by realizing that analysis might refine the information about the actual parameters within a method and propagating the new values discovered back to σ_f. For example, in a method `foo(Vector v){if v!=null skip else throw_null}`, it is clear that we can only finish normally if $nl_x(v) = nnull$, but in the actual semantics we do not change the nullity value for the corresponding argument in the call, which can only be more imprecise. Note that the example is different from `foo(Vector v){v = new Vector}`, which also finishes with $nl_x(v) = nnull$. The distinction over whether new attributes are preserved or not relies on keeping track of those variables which have been assigned inside the method, and then applying the propagation only for the unset variables.

Example 4. Assume an extra snippet of code in the `Vector` class of the form `if (v2!=null) v1.append(v2) else com`, which is analyzed in state $\sigma = (\{\{v_1\}, \{v_2\}\}, \{v_1/nnull,\ v_2/nnull\}, \{v_1/\{vector\}, v_2/\{vector\}\})$. Since we have nullity information, it is possible to identify the block `com` as dead code. In contrast, sharing-only analyses can only tell if a variable is definitely null, but never if it is definitely non-null. The call is analyzed as follows. Let $A = \{v_1, v_2\}$ and $F = \{this, v\}$, then $\sigma_p = \sigma|_A = \sigma$ and the entry state σ_y is $\sigma|_A^F = (\{\{this\}, \{v\}\}, \{this/nnull, v/nnull\}, \{this/\{vector\}, v/\{vector\}\})$. The only class where `append` can be executed is `Vector` and results (see Example 3) in an exit state for the formal parameters and the return variable $\sigma_b = (\{\{this, v\}\}, \{this/nnull, v/nnull, out/null\}, \{this/\{vector\}, v/\{vector\}, out/\{void\}\})$, which is further renamed to the scope of the caller obtaining $\sigma_\lambda = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull, res/null\}, \{v_1/\{vector\}, v_2/\{vector\}, res/\{void\}\})$. Since the method returns a `void` type we can treat res as a primitive (null) variable so $\sigma_f = extend(\sigma, \sigma_\lambda, \{v_1, v_2\}) = (\{\{v_1, v_2\}\}, \{v_1/nnull, v_2/nnull, res/null\}, \{v_1/\{vector\}, v_2/\{vector\}, res/\{void\}\})$.

Example 5. The *extend* operation used during interprocedural analysis is a point where there can be significant loss of precision and where set sharing shows its strengths. For simplicity, we will describe the example only for the sharing component; nullity and type information updates are trivial. Assume a scenario where a call to `append(v1,v2)` in sharing state $sh = \{\{v_0, v_1\}, \{v_1\}, \{v_2\}\}$ results in $sh_\lambda = \{\{v_1, v_2\}\}$. Let A and AR be the sets $\{v_1, v_2\}$ and $\{v_1, v_2, res\}$ respectively. The *extend* operation proceeds as follows: first we calculate *star* as $(sh_A \cup \{\{res\}\})^* = (sh \cup \{\{res\}\})^* = (\{\{v_0, v_1\}, \{v_1\}, \{v_2\}, \{res\}\})^* = \{\{v_0, v_1\}, \{v_0, v_1, v_2\}, \{v_0, v_1, v_2, res\}, \{v_0, v_1, res\}, \{v_1\}, \{v_1, v_2\}, \{v_1, v_2, res\}, \{v_1, res\}, \{v_2\}, \{v_2, res\}, \{res\}\}$, from which we delete those elements whose projection over AR is not included in sh_λ, obtaining $sh_{ext} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}$. The resulting sharing component is the union of that sh_{ext} with $sh_{-A} = \emptyset$, so $sh_{f1} = sh_{ext} = \{\{v_0, v_1, v_2\}, \{v_1, v_2\}\}$.

When the same sh and sh_λ are represented in their pair sharing versions $sh^p = \{\{v_0, v_1\}, \{v_o, v_0\}, \{v_1, v_1\}, \{v_2, v_2\}\}$ and $sh^p_\lambda = \{\{v_1, v_2\}, \{v_1, v_1\}, \{v_2, v_2\}\}$, the *extend* operation in [26] introduces spurious sharings in sh_f because of the lower precision of the pair-sharing representation. In this case, $sh^p_{f2} = (sh \cup$

$sh^p_\lambda)^*_A = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_1, v_2\}, \{v_0, v_0\}, \{v_1, v_1\}, \{v_2, v_1\}\}$. This information, expressed in terms of set sharing, results in $sh_{f2} = \{\{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_1, v_2\}, \{v_1, v_2\}, \{v_0\}, \{v_1\}, \{v_2\}\}$, which is much less precise that sh_{f1}.

4 Experimental Results

In our analyzer the abstract semantics presented in the previous section is evaluated by a highly optimized fixpoint algorithm, based on that of [21]. The algorithm traverses the program dependency graph, dynamically computing the strongly-connected components and keeping detailed dependencies on which parts of the graph need to be recomputed when some abstract value changes during the analysis of iterative code (loops and recursions). This reduces the number of steps and iterations required to reach the fixpoint, which is specially important since the algorithm implements *multivariance*, i.e., it keeps different abstract values at each program point for every calling context, and it computes (a superset of) all the calling contexts that occur in the program. The dependencies kept also allow relating these values along execution paths (this is particularly useful for example during error diagnosis or for program specialization).

We now provide some precision and cost results obtained from the implementation in the framework described in [17] of our set-sharing, nullity, and class ($SSNlTau$) analysis. In order to be able to provide a comparison with the closest previous work, we also implemented the pair sharing (PS) analysis proposed in [26]. We have extended the operations described in [26], enabling them to handle some additional cases required by our benchmark programs such as primitive variables, visibility of methods, etc. Also, to allow direct comparison, we implemented a version of our $SSNlTau$ analysis, which is referred to simply as SS, that tracks set sharing using only declared type information and does not utilize the (non-)nullity component. In order to study the influence of tracking run-time types we have implemented a version of our analysis with set sharing and (non-)nullity, but again using only the static types, which we will refer to as $SSNl$. In these versions without dynamic type inference only declared types can affect τ and thus the dynamic typing information that can be propagated from initializations, assignments, or correspondence between arguments and formal parameters on method calls is not used. Note however that the version that includes tracking of dynamic typing can of course only improve analysis results in the presence of polymorphism in the program: the results should be identical (except perhaps for the analysis time) in the rest of the cases. The polymorphic programs are marked with an asterisk in the tables.

The benchmarks used have been adapted from previous literature on either abstract interpretation for Java or points-to analysis [26,24,23,29]. We added two different versions of the `Vector` example of Fig. 2. Our experimental results are summarized in Tables 5, 6, and 7.

The first column ($\#tp$) in Tables 5 and 6 shows the total number of program points (commands or expressions) for each program. Column $\#rp$ then provides, for each analysis, the total number of *reachable* program points, i.e., the

		PS				SS				
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$
dyndisp (*)	71	68	3	114	30	68	3	114	29	-2
clone	41	38	3	42	52	38	3	50	81	55
dfs	102	98	4	103	68	98	4	108	68	0
passau (*)	167	164	3	296	97	164	3	304	120	23
qsort	185	142	43	182	125	142	43	204	165	32
integerqsort	191	148	43	159	110	148	43	197	122	10
pollet01 (*)	154	126	28	276	196	126	28	423	256	30
zipvector (*)	272	269	3	513	388	269	3	712	1029	164
cleanness (*)	314	277	37	360	233	277	37	385	504	116
overall	1497	1330	167	2045	1299	1330	167	2497	2374	82.75

Fig. 5. Analysis times, number of program points, and number of abstract states

		SSNl					SSNlTau				
	$\#tp$	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$	$\#rp$	$\#up$	$\#\sigma$	t	$\%\Delta t$
dyndisp (*)	71	61	10	103	53	77	61	10	77	20	-33
clone	41	31	10	34	100	92	31	10	34	90	74
dfs	102	91	11	91	129	89	91	11	91	181	166
passau (*)	167	157	10	288	117	18	157	10	270	114	17
qsort	185	142	43	196	283	125	142	43	196	275	119
integerqsort	191	148	43	202	228	107	148	43	202	356	224
pollet01 (*)	154	119	35	364	388	98	98	56	296	264	35
zipvector (*)	272	269	3	791	530	36	245	27	676	921	136
cleanness (*)	314	276	38	383	276	38	266	48	385	413	77
overall	1497	1294	203	2452	2104	61.97	1239	258	2227	2634	102.77

Fig. 6. Analysis times, number of program points, and number of abstract states

number of program points that the analysis explores, while $\#up$ represents the $(\#tp - \#rp)$ points that are not analyzed because the analysis determines that they are unreachable. It can be observed that tracking (non-)nullity (Nl) reduces the number of reachable program points (and increases conversely the number of unreachable points) because certain parts of the code can be discarded as dead code (and not analyzed) when variables are known to be non-null. Tracking dynamic types (Tau) also reduces the number of reachable points, but, as expected, only for (some of) the programs that are polymorphic. This is due to the fact that the class analysis allows considering fewer implementations of methods, but obviously only in the presence of polymorphism.

Since our framework is multivariant and thus tracks many different *contexts* at each program point, at the end of analysis there may be more than one abstract state associated with each program point. Thus, the number of abstract states inferred is typically larger than the number of reachable program points. Column $\#\sigma$ provides the total number of these abstract states inferred by the analysis. The level of multivariance is the ratio $\#\sigma/\#rp$. It can be observed that the simple

	PS		SS	
	$\#sh$	$\%sh$	$\#sh$	$\%sh$
dyndisp (*)	640	60.37	435	73.07
clone	174	53.10	151	60.16
dfs	1573	96.46	1109	97.51
passau (*)	5828	94.56	3492	96.74
qsort	1481	67.41	1082	76.34
integerqsort	2413	66.47	1874	75.65
pollet01 (*)	793	89.81	1043	91.81
zipvector (*)	6161	68.71	5064	80.28
cleanness (*)	1300	63.63	1189	70.61
overall	20363	73.39	15439	80.24

Fig. 7. Sharing precision results

set sharing analysis (SS) creates more abstract states for the same number of reachable points. In general, such a larger number for $\#\sigma$ tends to indicate more precise results (as we will see later). On the other hand, the fact that addition of Nl and Tau reduces the number of reachable program points interacts with precision to obtain the final $\#\sigma$ value, so that while there may be an increase in the number of abstract states because of increased precision, on the other hand there may be a decrease because more program points are detected as dead code by the analysis. Thus, the $\#\sigma$ values for $SSNl$ and $SSNlTau$ in some cases actually decrease with respect to those of PS and SS.

The t column in Tables 5 and 6 provides the running times for the different analyses, in milliseconds, on a Pentium M 1.73Ghz, 1Gb of RAM, running Fedora Core 4.0, and averaging several runs after eliminating the best and worst values. The $\%\Delta t$ columns show the percentage variation in the analysis time with respect to the reference pair-sharing (PS) analysis, calculated as $\Delta_{dom}\%t = 100*(t_{dom} - t_{PS})/t_{PS}$. The more complex analyses tend to take longer times, while in any case remaining reasonable. However, sometimes more complex analyses actually take less time, again because the increased precision and the ensuing dead code detection reduces the amount of program that must be analyzed.

Table 7 shows precision results in terms of sharing, concentrating on the SP and SS domains, which allow direct comparison. A more usage-oriented way of measuring precision would be to study the effect of the increased precision in an application that is known to be sensitive to sharing information, such as, for example, program parallelization [4]. On the other hand this also complicates matters in the sense that then many other factors come into play (such as, for example, the level of intrinsic parallelism in the benchmarks and the parallelization algorithms) so that it is then also harder to observe the precision of the analysis itself. Such a client-level comparison is beyond the scope of this paper, and we concentrate here instead on measuring sharing precision directly.

Following [6], and in order to be able to compare precision directly in terms of sharing, column $\#sh$ provides the sum over all abstract states in all reachable program points of the cardinality of the sharing *sets* calculated by the analysis.

For the case of pair sharing, we converted the pairs into their equivalent set representation (as in [6]) for comparison. Since the results are always correct, a smaller number of sharing sets indicates more precision (recall that $\top$ is the power set). This is of course assuming σ is constant, which as we have seen is not the case for all of our analyses. On the other hand, if we compare PS and SS, we see that SS has consistently more abstract states than PS and consistently lower numbers of sharing sets, and the trend is thus clear that it indeed brings in more precision. The only apparent exception is *pollet01* but we can see that the number of sharing sets is similar for a significantly larger number of abstract states.

An arguably better metric for measuring the relative precision of sharing is the ratio $\%_{Max} = 100 * (1 - \#sh/(2^{\#vo} - 1))$ which gives $\#sh$ as a percentage of its maximum possible value, where $\#vo$ is the total number of object variables in all the states. The results are given in column $\%sh$. In this metric 0% means all abstract states are $\top$ (i.e., contain no useful information) and 100% means all variables in all abstract states are detected not to share. Thus, larger values in this column indicate more precision, since analysis has been able to infer smaller sharing sets. This relative measure shows an average improvement of 7% for SS over PS.

5 Conclusions

We have proposed an analysis based on abstract interpretation for deriving precise sharing information for a Java-like language. Our analysis is multivariant, which allows separating different contexts, and combines Set Sharing, Nullity, and Classes: the domain captures which instances definitely do not share or are definitively null, and uses the classes to refine the static information when inheritance is present. We have implemented the analysis, as well as previously proposed analyses based on Pair Sharing, and obtained encouraging results: for all the examples the set sharing domains (even without combining with Nullity or Classes) offer more precision than the pair sharing counterparts while the increase in analysis times appears reasonable. In fact the additional precision (also when combined with nullity and classes) brings in some cases analysis time reductions. This seems to support that our contributions bring more precision at reasonable cost.

Acknowledgments

The authors would like to thank Samir Genaim for many useful comments to previous drafts of this document. Manuel Hermenegildo and Mario Méndez-Lojo are supported in part by the Prince of Asturias Chair in Information Science and Technology at UNM. This work was also funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of

Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the *PROMESAS* project.

References

1. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)
2. Alves-Foss, J. (ed.): Formal Syntax and Semantics of Java. LNCS, vol. 1523. Springer, Heidelberg (1999)
3. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Proc. of OOPSLA 1996, SIGPLAN Notices, October 1996, vol. 31(10), pp. 324–341 (1996)
4. Bueno, F., García de la Banda, M., Hermenegildo, M.: Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. ACM Transactions on Programming Languages and Systems 21(2), 189–238 (1999)
5. Burke, M.G., et al.: Carini, Jong-Deok Choi, and Michael Hind. In: Pingali, K.K., et al. (eds.) LCPC 1994. LNCS, vol. 892, pp. 234–250. Springer, Heidelberg (1995)
6. Codish, M., et al.: Improving Abstract Interpretations by Combining Domains. ACM Transactions on Programming Languages and Systems 17(1), 28–44 (1995)
7. Cortesi, A., et al.: Complementation in abstract interpretation. ACM Trans. Program. Lang. Syst. 19(1), 7–47 (1997)
8. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL 1977, pp. 238–252 (1977)
9. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Sixth ACM Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282 (1979)
10. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
11. Gosling, J., et al.: Java(TM) Language Specification, 3rd edn. Addison-Wesley Professional, Reading (2005)
12. Hill, P.M., Payet, E., Spoto, F.: Path-length analysis of object-oriented programs. In: Proc. EAAI 2006 (2006)
13. Hind, M., et al.: Interprocedural pointer alias analysis. ACM Trans. Program. Lang. Syst. 21(4), 848–894 (1999)
14. Jacobs, D., Langen, A.: Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In: 1989 North American Conference on Logic Programming, MIT Press, Cambridge (1989)
15. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing (with retrospective). In: McKinley, K.S. (ed.) Best of PLDI, pp. 473–489. ACM Press, New York (1992)
16. Méndez-Lojo, M., Hermenegildo, M.: Precise Set Sharing for Java-style Programs (and proofs). Technical Report CLIP2/2007.1, Technical University of Madrid (UPM), School of Computer Science, UPM (November 2007)
17. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007) (August 2007)

18. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In: ISSTA, pp. 1–11 (2002)
19. Muthukumar, K., Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In: 1989 North American Conference on Logic Programming, October 1989, pp. 166–189. MIT Press, Cambridge (1989)
20. Muthukumar, K., Hermenegildo, M.: Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In: 1991 International Conference on Logic Programming, June 1991, pp. 49–63. MIT Press, Cambridge (1991)
21. Navas, J., Méndez-Lojo, M., Hermenegildo, M.: An Efficient, Context and Path Sensitive Analysis Framework for Java Programs. In: 9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007 (July 2007)
22. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: OOPSLA, pp. 146–161 (1991)
23. Pollet, I.: Towards a generic framework for the abstract interpretation of Java. PhD thesis, Catholic University of Louvain, Dept. of Computer Science (2004)
24. Pollet, I., Le Charlier, B., Cortesi, A.: Distinctness and sharing domains for static analysis of java programs. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, Springer, Heidelberg (2001)
25. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL 1999 (1999)
26. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: SAS, pp. 320–335 (2005)
27. Søndergaard, H.: An application of abstract interpretation of logic programs: occur check reduction. In: Duijvestijn, A.J.W., Lockemann, P.C. (eds.) Trends in Information Processing Systems. LNCS, vol. 123, pp. 327–338. Springer, Heidelberg (1981)
28. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: PLDI, pp. 387–400 (2006)
29. Streckenbach, M., Snelting, G.: Points-to for java: A general framework and an empirical comparison. Technical report, University Passau (November 2000)
30. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, pp. 131–144. ACM Press, New York (2004)
31. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: PLDI, pp. 1–12 (1995)

Towards Execution Time Estimation in Abstract Machine-Based Languages *

E. Mera [1]
[1] Complutense University of Madrid
edison@fdi.ucm.es

P. Lopez-Garcia[2,3]
[2] IMDEA-Software
[3] CSIC
pedro.lopez.garcia@imdea.org

M. Carro[4], M. Hermenegildo[2,4,5]
[4] Technical U. of Madrid
[5] U. of New Mexico
{mcarro, herme}@fi.upm.es

Abstract

Abstract machines provide a certain separation between platform-dependent and platform-independent concerns in compilation. Many of the differences between architectures are encapsulated in the specific abstract machine implementation and the bytecode is left largely architecture independent. Taking advantage of this fact, we present a framework for estimating upper and lower bounds on the execution times of logic programs running on a bytecode-based abstract machine. Our approach includes a one-time, program-independent profiling stage which calculates constants or functions bounding the execution time of each abstract machine instruction. Then, a compile-time cost estimation phase, using the instruction timing information, infers expressions giving platform-dependent upper and lower bounds on actual execution time as functions of input data sizes for each program. Working at the abstract machine level makes it possible to take into account low-level issues in new architectures and platforms by just reexecuting the calibration stage instead of having to tailor the analysis for each architecture and platform. Applications of such predicted execution times include debugging/verification of time properties, certification of time properties in mobile code, granularity control in parallel/distributed computing, and resource-oriented specialization.

Categories and Subject Descriptors D.4.8 [*Performance*]: Modeling and prediction;
F.3.2 [*Semantics of Programming Languages*]: Program analysis;
D.1.6 [*Programming Techniques*]: Logic programming

General Terms Languages, performance

Keywords Execution Time Estimation, Cost Analysis, Profiling, Resource Awareness, Cost Models, Logic Programming.

* The authors have been partially supported by EU projects 215483 *S-Cube*, IST-15905 *MOBIUS*, Spanish projects ITEA2/PROFIT FIT-340005-2007-14 *ES_PASS*, ITEA/PROFIT FIT-350400-2006-44 *GGCC*, MEC TIN2005-09207-C03-01 *MERIT/COMVERS*, Comunidad de Madrid project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo is also partially funded by the Prince of Asturias Chair in Information Science and Technology.

PPDP'08, July 15–17, 2008, Valencia, Spain.

1. Introduction

Cost analysis has been studied for several declarative languages (7; 16; 11; 13). In logic programming previous work has focused on inferring upper (12; 11) or lower (13; 8) bounds on the cost of programs, where such bounds are *functions on the size (or values) of input data*. This approach captures well the fact that program execution cost in general depends on input data sizes. On the other hand the results of these analyses are given in terms of *execution steps*. While this measure has the advantage of being platform independent, it is not straightforward to translate such steps into execution time.

Estimation of *worst case execution times* (WCET) has received significant attention in the context of high-level imperative programming languages (24). In (18; 6) a portable WCET analysis for Java is proposed. However, the WCET approach only provides absolute upper bounds on execution time (i.e., bounds that do not depend on program input arguments) and often requires annotating loops manually.

Our objective is to infer automatically more precise bounds on execution times that are in general functions that depend on input data sizes. In (19) a static analysis was proposed in order to infer such platform-dependent time bounds in logic programs. This approach is based on a high-level analysis of certain syntactic characteristics of the program clause text (sizes of terms in heads, sizes of terms in bodies, number of arguments, etc.). Although promising experimental results were obtained, the predicted execution times were not very precise. In this paper we propose a new analysis which, in order to improve the accuracy of the time predictions, on one hand takes into account lower level factors and on the other makes the model richer by directly taking into account the inherently variable cost of certain low-level operations.

Regarding the choice of this lower level, rather than trying for example to model directly the characteristics of the physical processor, as in WCET, and given that most popular logic programming implementations are based on variations of the Warren abstract machine (WAM) (23; 1), we chose to model cost at the level of abstract machine instructions. Abstract machines have been used as a basic implementation technique in several programming paradigms (functional, logic, imperative, and object-oriented) (14) with the advantage that they provide an intermediate layer that separates to a certain extent the many low-level details of real (hardware) machines from the higher-level language, while at the same time making compilation easier. This property can be used to facilitate the design of our framework.

Within this setting, we present a new framework for the static estimation of execution times of programs. The basic ideas in our approach follow:

1. Measure the execution time of each of the instructions in a lower-level L_B (bytecode) language (or approximate it with a function if it depends on the value of an argument) in some specific abstract machine implementation when executed on a given processor / O.S.
2. Make the information regarding instruction execution time available to the timing analyzer. This is, in our proposal, done by means of *cost assertions* (written in a suitable assertion language) which are stored in a module accessible to the compiler/analyzer.
3. Given a concrete program P written in the source language L_H, compile it into L_B and record the relationship between P and its compiled counterpart.
4. Automatically analyze program P, taking into account the instruction execution time (determined in item 1 above) to infer a cost function C_P. This function is an expression which returns (bounds on) the actual execution time of P for different input data sizes for the given platform.

Points (1) and (2) are performed in a one-time profiling phase, independent from program P, while the rest are performed once for each P in the static (compile-time) cost analysis phase. We would like to point out that, in general, the basic ideas underlying our work can be applied to any language L_H as long as (i) cost estimation can be derived for programs written in L_H, (ii) the translation of L_H to some other (usually lower-level) language L_B is accessible, and (iii) the execution time of the instructions in L_B can be timed accurately enough. We will, however, focus herein on logic languages, so that we assume L_H to be a Prolog-like language and L_B some variant of the WAM bytecode.

The proposed framework has been implemented as part of the CiaoPP (17) system in such a way that any abstract machine properly instrumented can be analyzed. To the best of our knowledge, this is the first attempt at providing a timing analysis producing upper- and lower-bound time *functions* based on the cost of lower-level machine instructions.

2. Mappings Between Program Segments and Bytecodes

Let $OpSet = \{b_1, b_2, \ldots, b_n\}$ be the set of instructions of the abstract machine under consideration. We assume that each instruction is defined by a numeric identifier and its arity, i.e., $b_i \equiv f_i/n_i$, where f_i is the identifier and n_i the arity. Each program is compiled into a sequence of expressions of the form $f(a_1, a_2, \ldots, a_n)$ where f is the instruction name and the a_i's are its arguments. For conciseness, we will use I_i to refer to such expressions. The sequences of expressions into which a program is compiled are generally encoded using bytecodes. In the following we will often refer to sequences of abstract machine instructions or sequences of bytecodes simply as "bytecodes."

Let C be a clause H :- $L_1, \ldots, L_m$. Let $E(C)$ be a function that returns the sequence of bytecodes resulting from the compilation of clause C:

$$E(C) =< I_1, I_2, \ldots, I_p >$$

Let $E(C, H)$ be a function that maps the clause head H to the sequence of bytecodes in $E(C)$ starting from the beginning up to the first `call/execute` instruction or to the end of the sequence $E(C)$ if there are no more `call/execute` instructions (i.e., to the end of the bytecode sequence resulting from the compilation of clause C). Let $E(C, L_i)$ be the function that maps literal L_i of clause C to the sequence of bytecodes in $E(C)$ which start at the `call` bytecode instruction corresponding to this literal and up to the next `call/execute` instruction or to the end of the sequence $E(C)$ if there are no more `call/execute` instructions. If $\uplus$ represents the concatenation of sequences of bytecodes, then:

$$E(C) = E(C, H) \biguplus (\biguplus_{i=1}^{m} E(C, L_i))$$

Note that functions $E(C, H)$ and $E(C, L_i)$ do not necessarily return the bytecodes that one would normally associate to the clause head H and literal L_i respectively. Instead, the definition of those functions associates the instructions corresponding to argument preparation for a given call with the (success of the) *previous* call (or head). This is to cater for the fact that, in the context of backtracking, the WAM argument preparation occurs only one time per call to a literal, even if such call is retried more times before failing definitively. As a result, the cost of argument preparation for a given `call/execute` instruction needs to be associated with the previous literal to that `call/execute`, in order not to count it every time the call is retried.

Table 1 shows how predicate append/3 is compiled into bytecodes, and identifies the result of calling the $E(C, H)$ and $E(C, L_i)$ functions for each clause head and body literal. H^1 represents the head of the first clause (C_1), and H^2 and L_1^2 the head of the second (recursive) clause (C_2) and the first literal in such clause body (the only body literal).

$E(C_1, H^1)$	`append([], X, X).`	
	append/3/1:	try_me_else append/3/2 allocate get_constant([],A0) get_variable(V0,A1) get_value(V0,A2) deallocate proceed
$E(C_2, H^2)$	`append([X|Xs], Y, [X|Zs]) :-`	
	append/3/2:	trust_me allocate get_variable(V0,A0) set_variable(V1) set_variable(V2) set_variable(V3) get_list(V1,V3) set_variable(V4) unify_variable(V2,V4) unify_variable(V0,V3) set_variable(V5,A1) get_variable(V6,A2) set_variable(V7) set_variable(V8) get_list(V1,V8) set_variable(V9) unify_variable(V7,V9) unify_variable(V6,V8) put_value(V2,A0) put_value(V5,A1) put_value(V7,A2) deallocate
$E(C_2, L_1^2)$	`append(Xs, Y, Zs).`	
		execute append/3

Table 1. Sequences of bytecodes assigned to clause heads and body literals of the clauses C_1 and C_2 of predicate append by the functions $E(C, H)$ and $E(C, L)$.

3. Modeling the Execution Time of Instructions

We define a function $t(I)$ (the *timing model*), which takes a bytecode instruction I and returns another function which estimates the execution time for it depending on the input data sizes of the bytecode. This is similar to the approach described in (5), where, however, $t(I)$ was a constant.

In many cases we can assume that the time to execute a bytecode is constant. However there are some instructions for which this does not hold because their definitions involve loops. In many of these cases the timing model consists of an initial constant time t_0 plus another additional constant time t_{iter} to cater for the cost of each iteration, and a simple linear model can be used: $t_0 + n \times t_{iter}$. Consider for example the `unify_void` n instruction, which pushes n new unbound cells on the heap (1), and whose execution time is a linear function on n. In some other cases instructions have different execution times depending on the (fixed) values a given argument can take from some finite set. In such cases, execution time is an arbitrary function on the argument. Specific constants are assigned for each possible argument value by means of profiling (Section 5).

Since the cost of a given instruction is different when it succeeds and when it fails, we will have two costs for each instruction that can fail: one for the success case and another for the failure case. Finally, and besides lower-level factors such as cache behavior, there are some additional variable factors (such as, e.g., the length of dereferencing chains) which may affect execution times. These factors are in principle not impossible to cater for via a combination of static and dynamic analysis, but, given the additional complication involved, we will ignore them herein and explore what kind of precision of timing prediction can be achieved with this first level of approximation.

Another factor that we are not taking into account at this moment is garbage collection (GC). GC makes programs run slower, which, at profiling time, increases the (estimated) cost of every instruction. Therefore, turning it off at profile time (which gives a smaller estimation of instruction cost) is safe when finding out lower bounds: if the program whose execution time is to be predicted is run with GC turned on, then it would run slower w.r.t. an execution with GC turned off (as it was when profiling), and the estimated bounds will still be lower bounds, albeit more conservative. An inverse reasoning applies to upper bounds, and the technique herein presented is equally valid. However, for the sake of simplicity, we have taken all the measurements (both for profiling and executions to be predicted) with GC disconnected.

4. Static Cost Analysis

We now present the compile-time component of our combined framework: the static cost analysis. This analysis has been implemented and integrated in `CiaoPP` (17).

4.1 Overview of the Approach

Since the work done by a call to a recursive procedure often depends on the "size" of its input, knowing this size is a prerequisite to statically estimate such work. Our basic approach is as follows: given a call p, an expression $\Phi_p(n)$ is *statically* computed that (i) is relatively simple to evaluate, and (ii) it approximates $\mathtt{Time}_p(n)$, where $\mathtt{Time}_p(n)$ denotes the cost (in time units) of computing p for an input of size n on a given platform. Various measures are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. It is then evaluated at run-time, when the size of the input is known, yielding (upper or lower) bounds on the execution time required by the computation of the call on a given platform. In the following we will refer to the compile-time computed expressions $\Phi_p(n)$ as *cost functions*.

Certain program information (such as, for example, input/output modes and size metrics for predicate arguments) is first automatically inferred by other analyzers which are part of `CiaoPP` and then provided as input to the size and cost analysis. The techniques involved in inferring this information are beyond the scope of this paper —see, e.g., (17) and its references for some examples. Based on this information, our analysis first finds bounds on the size of input arguments to the calls in the body of the predicate being analyzed, relative to the sizes of the input arguments to this predicate, using the inferred metrics. The size of an output argument in a predicate call depends in general on the size of the input arguments in that call. For this reason, for each output argument we infer an expression which yields its size as a function of the input data sizes. To this end, and using the input-output argument information, data dependency graphs (namely the *argument dependency graph* and the *literal dependency graph*) are used to set up difference equations whose solution yields size relationships between input and output arguments of predicate calls. The *argument dependency graph* is a directed acyclic graph used to represent the data dependency between argument positions in a clause body (and between them and those in the clause head). The *literal dependency graph* is constructed from the *argument dependency graph* (grouping nodes) and represents the data dependencies between literals.

The information regarding argument sizes is then used to set up another set of difference equations whose solution provides bound functions on predicate calls (execution time). Both the size and cost difference equations must be solved by a difference equation solver. Although the operation of such solvers is beyond the scope of the paper, our implementation does provide a table-based solver which covers a reasonable set of difference equations such as first-order and higher-order linear difference equations in one variable with constant and polynomial coefficients,[1] divide and conquer difference equations, etc. In addition, the system allows the use of external solvers (such as, e.g., Purrs (4), Mathematica, Matlab, etc.) and is currently being extended to interface with other interesting solvers that have been recently developed (2). Note also that, since we are computing upper/lower bounds, it suffices to compute upper/lower bounds on the solution of a set of difference equations, rather than an exact solution. This allows obtaining an *approximate* closed form when the exact solution is not possible.

4.2 Estimating the Execution Time of Clauses and Predicates

Our cost analysis approach is based on that developed in (12; 11) (for estimation of upper bounds on resolution steps) and further extended in (13) (for lower bounds). More recently, in (19) the analysis was extended to work with *vectors* of cost components, with each component considering a known aspect that affects the total cost of the program. In these approaches the cost of a clause can be bounded by the cost of head unification together with the cost of each of its body literals. For simplicity, the discussion that follows is focused on the estimation of upper bounds. We refer the reader to (13) for details on lower-bounds cost analysis.

Consider a predicate defined by r clauses $\mathtt{C}_1, \ldots, \mathtt{C}_r$. We take into account that a given clause $\mathtt{C}_k$ will be tried only if clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{k-1}$ fail to yield a solution. Consider clause $\mathtt{C}_k$ defined as $\mathtt{H}^k$:- $\mathtt{L}^k_1, \ldots, \mathtt{L}^k_m$. Because of backtracking, the number of times a literal will be executed depends on the number of solutions of the previous literals. Assume that $\overline{n}$ is a vector such that each element corresponds to the size of an input argument to clause $\mathtt{C}_k$ and that each $\overline{n}_i, i = 1 \ldots m$, is a vector such that each element corresponds to the size of an input argument to literal $\mathtt{L}^k_i$. Assume also that $\tau(\mathtt{H}^k, \overline{\mathtt{n}})$ is the execution time needed to resolve the head $\mathtt{H}^k$ of

[1] Note that it is always possible to reduce a system of linear difference equations to a single linear difference equation in one variable.

the clause $\mathtt{C}_k$ with the literal being solved, $\mathtt{Sols}_{\mathtt{L}_j^k}$ is the number of solutions literal $\mathtt{L}_j^k$ can generate, and $\beta(\mathtt{L}_i^k, \overline{n}_i)$ the time needed to prepare the call to literal $\mathtt{L}_i^k$ in the body of the clause $\mathtt{C}_k$. Because of space constraints, we refer the reader to (11; 13) for details about the algorithms used to estimate the number of solutions that a literal can generate, and the sizes of input arguments. Then, an upper bound $\mathtt{Cost}_{\mathtt{C}^k}(\overline{n})$ on the cost of clause $\mathtt{C}^k$ (assuming all solutions are required) can be expressed as:

$$\mathtt{Cost}_{\mathtt{C}^k}(\overline{n}) \leq \quad \tau(\mathtt{H}^k, \overline{n}) + \sum_{i=1}^{m} (\prod_{j \prec i} \mathtt{Sols}_{\mathtt{L}_j^k}(\overline{n}_j))(\beta(\mathtt{L}_i^k, \overline{n}_i) + \mathtt{Cost}_{\mathtt{L}_i^k}(\overline{n}_i))$$

Here we use $j \prec i$ to denote that $\mathtt{L}_j^k$ precedes $\mathtt{L}_i^k$ in the literal dependency graph for the clause $\mathtt{C}_k$ (described in Section 4.1). We have that:

$$\tau(\mathtt{H}^k, \overline{n}) = \delta_k(\overline{n}) + \sum_{I \in E(\mathtt{C}^k, \mathtt{H}^k)} t(I)(\overline{n})$$

where $\delta_k(\overline{n})$ denotes the execution time necessary to determine that clauses $\mathtt{C}_1, \ldots, \mathtt{C}_{k-1}$ will not yield a solution and that $\mathtt{C}_k$ must be tried: the function δ_k obviously takes into account the type and cost of the indexing scheme being used in the underlying implementation. Also:

$$\beta(\mathtt{L}_i^k, \overline{n}_i) = \sum_{I \in E(\mathtt{C}, \mathtt{L}_i^k)} t(I)(\overline{n}_i), i = 1, \cdots, m$$

with $E(\mathtt{C}, \mathtt{L}_i^k)$ and $t(I)$ defined as in Sections 2 and 3 respectively.

A difference equation is set up for each recursive clause, whose solution (using as boundary conditions the execution times of non-recursive clauses) is a function that yields the execution time of a clause. The execution time of a predicate is then computed from the execution time of its defining clauses. Since the number of solutions which will be required from a predicate is generally not known in advance, a conservative upper bound on the execution time of a predicate can be obtained by assuming that all solutions are needed, and, thus, all clauses are executed and the execution time of the predicate will be the sum of the execution times of its defining clauses. When the clauses of a predicate are mutually exclusive, a more precise estimation of the execution time of such a deterministic predicate can be obtained as the maximum of the execution times of the clauses it is composed of.

Note that our approach allows defining via assertions the execution time of external predicates, which can then be used for modular composition. This includes also predicates for which the code is not available or which are even written in a programming language that is not supported by the analyzer. In addition, assertions also allow describing by hand the execution time of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. The description of the assertion language used is out of the scope of this paper, and we refer the reader to (21) for details.

5. Estimating Instruction Execution Times via Profiling

In this section we will see how data regarding the expected execution time of each instruction in the abstract machine (Section 3) can be accurately measured in a realistic environment.

5.1 Instruction Profiling

Profiling aims at calculating a function $t(I)$ for each bytecode instruction I. An approach is to instrument the WAM implementation so that time measures are taken and recorded at appropriate points in the execution (18). In practice, a number of issues have to be taken into account in order to obtain accurate enough measurements. These include the selection of the places where the instrumentation code will be inserted, how to minimize the effects of such instrumentation on the execution (not only execution time but also, e.g., cache behavior), and how to work around the complex instruction scheduling performed by modern processors, which may lead to large variance in the results, especially since we aim at measuring very small fragments of code.

```
while (op != END) { /* WAM emulation loop */
  ...
   record_profile_info (op); /* op is the current bytecode */

  switch(op) {
    ...
  }
  ...
  op = get_next_op();
}
```

Figure 1. A simple WAM emulation loop instrumented.

A first approximation is to add profiling-related calls in designated parts of the bytecode interpreter main loop. Figure 1 shows a piece of code illustrating this. The `record_profile_info(op)` operation records the start time for the bytecode `op`. The end time is processed when the next opcode is fetched. The data for each bytecode is maintained in memory during execution (and in raw form in order to impact execution as little as possible) and later saved to an external file.

A benchmark-based analysis is also proposed in (18), which describes how the instrumented code can be reused effectively on various platforms without modifying it, and how the execution time of a specific set of bytecodes can be measured.

However, the methods mentioned above have drawbacks. For example, the first one (instrumenting the main loop) depends on the existence of very precise, non-intrusive, low-overhead timing operations which, unfortunately, are not always available in all platforms. Portable O.S. calls, besides having a typically high associated overhead, are in general not accurate enough for our purposes. Even if a very fast timing operation is available (which is not the case in platforms such as mobile and embedded devices), its introduction may affect the behavior of the machine being analyzed if the abstract machine loop is very optimized. For example, if the new code changes register and variable allocation, program behavior will be affected in unforeseen ways.

We will, however, use an instrumented loop like that of Figure 1 to count the number of bytecodes executed in a calibration step.

5.2 Measuring Time Accurately

In order to do portable time measurements in platforms where high resolution timing is difficult or impossible to achieve, workarounds have to be used. The approach that we have followed is based on using synthetic benchmarks which on purpose repeatedly execute the instructions under estimation for a large enough time, and later divide the total execution time by the number of times the instructions were executed. A complication in this process is that it is in general not possible to run a single instruction repeatedly within the abstract machine, since the resulting sequence would not be legal and may "break" the abstract machine, run out of memory, etc. In general, more complex sequences of instructions must be constructed and repeated instead.

Therefore, the approach we have followed involves designing a set of *legal* programs which cover all the bytecode instructions,

Programs	Instructions	Trace
c1_5 :- c1_5_0.	00 : execute 01	00 : execute 01
c1_5_0 :- c1_5_1.	01 : execute 02	01 : execute 02
c1_5_1 :- c1_5_2.	02 : execute 03	02 : execute 03
c1_5_2 :- c1_5_3.	03 : execute 04	03 : execute 04
c1_5_3 :- c1_5_4.	04 : execute 05	04 : execute 05
c1_5_4 :- c1_5_5.	05 : execute 06	05 : execute 06
c1_5_5.	06 : proceed	06 : proceed
c1_0 :- c1_0_0.	01 : execute 02	01 : execute 02
c1_0_0.	02 : proceed	02 : proceed

Table 2. Programs used in order to get the execution time of the `execute` instruction.

relate the execution time of these programs with the individual instruction execution times with a system of equations, and solving such a system.

5.3 Getting Instruction Execution Time

We now discuss how to set up calibration programs in order to get the cost of bytecodes. In this section, and in order to simplify the discussion, we deal with those bytecodes whose execution time is bound by a constant. In the following section we extend our technique to manage instructions whose execution time is unbound.

Let $C_i, i = 0, 1, \ldots, n$ be a set of synthetic calibration programs, each of them returning the execution time of a block of code. Each C_i, which we will refer to as calibrator, is generated in such a way that it repeats such block a given number of times, say r. Let us assume, for example, that we want to calibrate the WAM instruction "`execute`" when it does not fail and that we want to repeat its execution 5 times (i.e., $r = 5$). Table 2 shows a set of programs which can be used to calibrate this WAM instruction. Columns **Instructions** and **Trace** show the WAM code as generated by the compiler and the sequence of instructions executed when running the program starting from the first clause respectively. In general, in our approach, rather than a concrete program, calibrators are program generation templates which take r as an input and return, e.g., the programs in Table 2 for that value of r. The actual calibration program includes an entry point which calls the programs in Table 2 and returns the value of the execution time of the `execute` instruction, subtracting the time spent in the entry calls (e.g., `c1_5` for Table 2). In this case the calibration time is easy to compute as the difference between the execution time of `c1_5` and `c1_0` divided by r. The result of the calibration should ideally be invariant with respect to r; in practice this is however not true due, among other factors, to timing imprecision. Thus, r needs to be determined for each case: it has to be a large enough value to ensure stability of the time measured by the calibrator for the particular platform and the method used to measure time, but not excessively large, as this would make calibration impractical.

In some cases we cannot isolate the behavior of only one bytecode. This is specially the case in the calibrators of instructions which alter the program flow, such as `call`, `proceed`, `trust_me`, `try_me_else`, `retry_me_else`, `allocate`, `deallocate`. It is also the case when measuring the cost of failure for any of the instructions which can fail (generally the `get_` and `unify_` instructions). All these instructions need to be always executed together with other bytecodes in order to make the calibration program legal. As a result, and due to interactions between the costs of the different instructions, the equations are not as easy to configure in all cases as the simple case for the `execute` instruction above.

As a simple example, the calibrator that returns the cost of `call` and the `proceed` instructions uses the programs in Table 3 (where we have turned off the optimization of register / variable allocation in the compiler for simplicity). In order to be able to separate the

Programs	Instructions	Trace
c2_5 :-	00 : allocate	00 : allocate
c_5,	01 : call 09	01 : call 09
c_5,	02 : call 09	09 : proceed
c_5,	03 : call 09	02 : call 09
c_5,	04 : call 09	09 : proceed
c_5,	05 : call 09	03 : call 09
c_5,	06 : call 09	09 : proceed
c_5.	07 : deallocate	04 : call 09
	08 : execute 09	09 : proceed
		05 : call 09
c_5.	09 : proceed	09 : proceed
		06 : call 09
		09 : proceed
		07 : deallocate
		08 : execute 09
		09 : proceed
c2_0 :-	00 : allocate	00 : allocate
c_0,	01 : call 04	01 : call 04
c_0.	02 : deallocate	04 : proceed
	03 : execute 04	02 : deallocate
		03 : execute 04
c_0.	04 : proceed	04 : proceed

Table 3. Programs used to get the execution time of the `call` and `proceed` instructions.

cost of `call` and `proceed` an idea might be to find a calibrator that isolates the cost of `proceed` by itself and subtract from the value given by the calibrator for `call` and `proceed` and obtain the cost of `call`. However, that is in general not possible since in all legal calibrators `proceed` and `call` must always appear combined with other bytecodes. In general we need to set up a system of equations in which the known values are the costs given by our calibrators and the unknown values are the costs of the individual bytecodes. Such equations can be configured automatically, by executing the calibration programs in a special version of the WAM with the bytecode dispatch loop instrumented as in Figure 1 so that the profiler keeps an account of the executed bytecodes.

Let c_i, $0 \leq i \leq n$, be the time calibrator C_i has returned, and let β_j, $0 \leq j \leq m$, $m \geq n$, be the cost of a bytecode B_j, distinguishing between the case of a fail or a success in the execution of such bytecode. In other words, $B_j \in I \times \{fail, success\}$, where I the set of all possible bytecodes and *fail* and *success* represent the failure or success of the execution of a bytecode. We can then set up the following system of equations:

$$\begin{array}{rcl} c_1 & = & a_{11}\beta_1 + a_{12}\beta_2 + \cdots + a_{1m}\beta_m \\ c_2 & = & a_{21}\beta_1 + a_{22}\beta_2 + \cdots + a_{2m}\beta_m \\ & \cdots & \\ c_n & = & a_{n1}\beta_1 + a_{n2}\beta_2 + \cdots + a_{nm}\beta_m \end{array} \quad (1)$$

which we can rewrite such using matrix notation:

$$C = AB \quad (2)$$

where $B = (\beta_i)$ is the vector of execution times for the bytecodes. In order to obtain B we ideally need to configure as many calibrators as bytecodes. Finding a solution to this system of equations requires, in principle, independence among the equations (i.e., there is no other linear independent equation but those in (1)), and to have as many equations as variables. However, that is not always possible due to dependencies between the number of times a bytecode is executed. For example, in the WAM under analysis, the following invariant holds:

PROPOSITION 1. *For any program, the number of times* `retry_me_else` *is called plus the number of times* `trust_me` *is called is equal to the number of failures.*

This holds since a failure always causes backtracking to the next choice point, which always implies executing either a `retry_me_else` or a `trust_me` instruction. As the coefficients a_{ij} in the equation above are precisely the number of times every bytecode is executed, it turns out that, for a given execution, some coefficients are dependent on some other coefficients, therefore breaking the initial independence assumption: the system of equations is underdetermined and it does not have a unique solution.

For this reason, since the coefficients a_{ij} where obtained by summarizing legal programs (i.e., the calibrators), and they will be affected by the linear dependency mentioned above, the undetermined system (2) will not have a unique solution. However, note that when several bytecodes in a block must be executed together (because of constraints in the WAM compilation and execution scheme) knowing the execution time of each of them in isolation is not needed: knowing the total execution time of the whole block is enough. This intuitive idea can be formalized and generalized with the following result:

PROPOSITION 2. *Given a set of n calibration programs C_i, that define n linear independent equations with β_i variables (corresponding to the m bytecodes, with both success and failure cases included), if we have that for all programs there exist $m-n$ linear independent relationships between the number of bytecodes that are always fulfilled, then the estimated execution time is invariant with respect to the choice of any arbitrary element of the solution set of such linear system.*

Proof : Let B be an arbitrary solution of $C = AB$. Let X be a vector which represents the number of times each bytecode has been executed for a given program. The estimated execution time is $E = X^T B$, i.e., the sum of the time for each bytecode multiplied by the number of times it has been executed.

By linear algebra, and considering that each calibrator defines a linear independent equation, we have that the range of A is n, and the kernel (or nullspace) of A is given by the set of all λ such that $A\lambda = 0$, a vector space of dimension $m - n$ (0 represents the null vector of dimension n). In other words, we have that:

$$C = AB = AB + 0 = AB + A\lambda = A(B + \lambda) \qquad (3)$$

Then, $B+\lambda$ is a solution of (2), and it is also a representative of the solution set of such equation system. What we should prove now is that $X^T(B + \lambda) = X^T B$, that is, canceling common terms and transposing the equations:

$$\lambda^T X = 0 \qquad (4)$$

On the other hand, we have a set of $m-n = k$ linear dependencies between the number of bytecodes executed of the form:

$$\begin{array}{rcl} 0 & = & v_{11}x_1 + v_{12}x_2 + \cdots + v_{1m}x_m \\ 0 & = & v_{21}x_1 + v_{22}x_2 + \cdots + v_{2m}x_m \\ & \cdots & \\ 0 & = & v_{k1}x_1 + v_{k2}x_2 + \cdots + v_{km}x_m \end{array}$$

Or, rewriting them using matrices:

$$0 = VX \qquad (5)$$

The result of multiplying an arbitrary vector d by V is a vector $\mu^T = (dV)$ and for the equation above, it follows that $\mu^T X = 0$.

But note that the rows of A were obtained executing a program that meets the linear dependencies too, that is, $\mu^T A^T = 0$. Transposing, we have:

$$A\mu = 0 \qquad (6)$$

For this reason, we can see that as λ, μ is a member of the kernel of A, and considering the uniqueness of the kernel of a matrix, and that μ is an element of a space of dimension $m - n$, we can choose μ such that $\lambda = \mu$, that is, we can express λ as the product $(dV)^T$, as result of basic theorems of linear algebra. Therefore, we have that:

$$\lambda^T X = \mu^T X = (dV)X = d(VX) = d(0) = 0 \qquad (7)$$

□

Then, the method we follow to select a representative solution B is simply to complete the equation systems with $m - n$ arbitrary equations in order to make them become determined. Such equations should be selected in such a way that the β_i values be positive, for example, by setting the cost to 0 as the time of the bytecodes that are faster, avoiding negative solutions.

5.4 Dealing with unbound instructions

We now consider the case of bytecode instructions whose execution time depends on the specific values that certain parameters can take at run time. In such cases the accuracy of our analysis can be increased by taking advantage of static term-size analysis and the addition of cost-related assertions for such instructions. Such assertions make it possible to introduce ad-hoc functions giving the size of the input parameters of the bytecode.

In fact, our system is able to deal with several metrics (e.g., value, length, size, depth, ...) as shown in (12; 11; 13), but for brevity, in the following paragraphs we will describe an example unifying lists.

Let us take, the instruction `unify_variable(V, W)` and let us assume that we want to calculate an upper bound for its execution time upon success and for the case where the two arguments to unify are lists of numbers. We assume that an upper bound to the execution time is proportional to the number of iterations necessary to scan the lists. The timing model for such instruction is thus $K_1 + K_2 * length(V)$, because if the instruction succeeds, the length of both V and W should be equal. The value of constants K_1 and K_2 is calculated by setting up two benchmarks which unify lists of different length l_1 and l_2. If the cost of `unify_variable` for these two list lengths is, respectively, B_1 and B_2, then we set up the following system of linear equations:

$$\begin{array}{rcl} B_1 & = & K_1 + K_2 \times l_1 \\ B_2 & = & K_1 + K_2 \times l_2 \end{array} \qquad (8)$$

Note that B_1 and B_2 can be added to the system of equations (2) to get its values in one step, and later, we solve K_1 and K_2 in the system of linear equations (6).

6. Experimental results

In order to evaluate the techniques presented so far we need to choose a concrete bytecode language and an implementation of its abstract machine to execute and profile with. As mentioned before, the de-facto target abstract machine for most Prolog compilers is the WAM (23; 1) or one of its derivatives. In order to evaluate the feasibility of the approach we have chosen a relatively simple WAM design, which is quite close to the original WAM definition. It is based on (9), but has been ported from Java to C/C++ to achieve similar performance of other Prolog systems. The use of a relatively simple abstract machine allows evaluating the technique while avoiding the many practical complications present in modern implementations, such as having complex instructions resulting from merging other, simpler ones, or specializations of instruction and argument combinations. This of course does not preclude the application of our technique to the more complex cases.

In our concrete abstract machine, we have considered 42 equations for 43 bytecodes, differentiating the success and failure cases.

As we have seen in Proposition **??**, there exists a linear relationship between the number of bytecodes that a program will call which can be stated as:

$$\begin{aligned} 0 = \quad & x_{30} + x_{38} - x_{13} - x_{15} - x_{17} - x_{22} - x_{41} \\ & - x_{43} - x_{49} - x_{50} - x_{51} - x_{52} - x_{53} \end{aligned}$$

where the x_i represent the number of times the bytecode tagged as β_i has been executed for any program being analyzed (see Tables 4 and 6).

By Proposition 1, we are free to select any arbitrary solution of the linear system. The proposed solution has been found by setting arbitrarily the cost of fail to zero. Then, our set of linear equations, discarding those whose calibrators are composed only with one bytecode, is as follows:

$$\begin{array}{lllll} 0 & = \beta_{13} & & c_{01} & = \beta_{01} + \beta_{07} \\ c_{20} & = \beta_{20} + \beta_{33} + \beta_{43} & & c_{09} & = \beta_{09} + \beta_{24} \\ c_{11} & = \beta_{01} + \beta_{11} + 2\beta_{28} + \beta_{30} & & c_{15} & = \beta_{15} + \beta_{38} \\ c_{46} & = \beta_{01} + 2\beta_{28} + \beta_{30} + \beta_{50} & & c_{17} & = \beta_{17} + \beta_{30} \\ c_{42} & = \beta_{01} + 2\beta_{27} + \beta_{30} + \beta_{52} & & c_{07} & = \beta_{07} + \beta_{24} \\ c_{22} & = \beta_{01} + \beta_{22} + \beta_{23} + \beta_{30} & & c_{29} & = \beta_{01} + \beta_{17} + \beta_{30} \\ c_{34} & = \beta_{01} + \beta_{23} + \beta_{30} + \beta_{35} & & c_{37} & = \beta_{17} + \beta_{38} \\ c_{36} & = \beta_{01} + 2\beta_{28} + \beta_{30} + \beta_{37} & & c_{38} & = \beta_{07} + \beta_{24} + \beta_{39} \\ c_{40} & = \beta_{01} + \beta_{23} + \beta_{30} + \beta_{41} & & c_{19} & = \beta_{19} + \beta_{33} \\ c_{43} & = \beta_{01} + \beta_{27} + \beta_{28} & & c_{13} & = \beta_{01} + \beta_{13} + \beta_{30} \\ & \quad + \beta_{30} + \beta_{49} & & & \\ c_{49} & = \beta_{01} + 2\beta_{19} + 2\beta_{27} & & c_{51} & = \beta_{01} + 2\beta_{20} \\ & \quad + \beta_{30} + 2\beta_{31} + 2\beta_{33} + \beta_{51} & & & \quad + \beta_{30} + 2\beta_{31} + \beta_{53} \end{array} \tag{9}$$

Solving this linear system we get:

$$\begin{array}{lcl} \beta_{01} & = & c_{29} - c_{17} \\ \beta_{07} & = & -c_{29} + c_{17} + c_{01} \\ \beta_{09} & = & -c_{29} + c_{17} + c_{09} - c_{07} + c_{01} \\ \beta_{11} & = & -2c_{27} - c_{13} + c_{11} \\ \beta_{13} & = & 0 \\ \beta_{15} & = & -c_{37} + c_{29} + c_{15} - c_{13} \\ \beta_{17} & = & c_{29} - c_{13} \\ \beta_{19} & = & c_{19} - c_{32} \\ \beta_{20} & = & -c_{44} - c_{32} + c_{20} \\ \beta_{22} & = & -c_{23} + c_{22} - c_{13} \\ \beta_{24} & = & c_{29} - c_{17} + c_{07} - c_{01} \\ \beta_{30} & = & -c_{29} + c_{17} + c_{13} \\ \beta_{35} & = & c_{34} - c_{23} - c_{13} \\ \beta_{37} & = & c_{36} - 2c_{27} - c_{13} \\ \beta_{38} & = & c_{37} - c_{29} + c_{13} \\ \beta_{39} & = & c_{38} - c_{07} \\ \beta_{41} & = & c_{40} - c_{23} - c_{13} \\ \beta_{49} & = & c_{43} - c_{27} - c_{26} - c_{13} \\ \beta_{50} & = & c_{46} - 2c_{27} - c_{13} \\ \beta_{51} & = & c_{49} - 2c_{30} - 2c_{26} - 2c_{19} - c_{13} \\ \beta_{52} & = & c_{42} - 2c_{26} - c_{13} \\ \beta_{53} & = & c_{51} + 2c_{44} + 2c_{32} - 2c_{30} - 2c_{20} - c_{13} \end{array} \tag{10}$$

The leftmost column of Tables 4 and 6 summarizes the calibration data for the instructions of our WAM implementation. For brevity, we actually only show those being used in the examples tested, although we have calibrated all of them. In the second column there is a tag that is the variable name in the linear equations system. In the examples we deal with a subset of Prolog which only has operations on integers, atoms, lists, and terms. Likewise, we obviate issues like modules or syntactic sugar which can be dealt with at the Prolog level. A few additional built-in predicates are required to have a minimal functionality including `write/1`, `consult/1`, etc. They are profiled separately and their timing is given to the system through assertions. This is also a valid solution in order to be able to analyze larger programs.

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
allocate	β_{01}	29	366	1055
arith_add	β_{02}	29	489	1438
arith_div	β_{03}	29	580	1541
arith_mod	β_{04}	29	641	1553
arith_mul	β_{05}	28	519	1468
arith_sub	β_{06}	28	519	1438
call	β_{07}	11	183	261
cut	β_{08}	13	183	581
deallocate	β_{09}	7	305	142
execute	β_{12}	15	152	574
get_constant_atom	β_{14}	38	518	1211
get_constant_int	β_{16}	28	396	1157
get_level	β_{18}	28	213	1054
get_list	β_{19}	20	275	763
get_struct	β_{20}	52	642	1766
get_value	β_{21}	43	488	1457
get_variable	β_{23}	43	549	1658
proceed	β_{24}	17	61	699
put_a_constant_atom	β_{25}	20	122	594
put_a_constant_int	β_{26}	20	122	506
put_constant_atom	β_{27}	37	274	1085
put_constant_int	β_{28}	37	274	997
put_value	β_{29}	21	183	910
retry_me_else	β_{30}	33	336	999
set_constant_atom	β_{31}	26	213	861
set_constant_int	β_{32}	25	183	767
set_variable	β_{33}	29	213	850
trust_me	β_{38}	29	336	973
try_me_else	β_{39}	30	457	1132
unequal	β_{40}	21	244	1021
unify_variable(nvar,var)	β_{42}	35	396	1309
unify_variable(var,nvar)	β_{43}	35	397	1309
unify_variable(int,int)	β_{44}	32	275	1179
unify_variable(atm,atm)	β_{46}	44	427	1413
unify_variable(str(1),str(1))	β_{47}	77	885	2560
unify_variable(list(1),list(1))	β_{45}	96	1068	3291
unify_variable(list(100),list(100))	β_{48}	4062	42511	217975

Table 4. Timing model for the WAM instructions. Cost of bytecodes when they succeed.

The experiments were made on the following representative platforms:

- Ultra**Sparc**-T1, 8 cores x 1GHz (4 threads per core), 8GB of RAM, SunOS 5.10.
- **Intel** Core Duo 1.66GHz, 2GB of RAM, Ubuntu Linux 7.04.
- Nokia **N810**. 400MHz processor, 128MB of RAM, Internet Tablet OS, Maemo Linux based OS2008 51.3

In order to reduce noise in the data because of spurious results, we have repeated each experiment 20 times and present the lowest results. In the calibration step 1000 repetitions were made (i.e., $r = 1000$). When possible, the tests were performed with the machines in single-user mode, stopping unnecessary processes. System tasks such as garbage collection, which, as mentioned before, is not considered in our model at the moment, were turned off.

Platform	Timing Model (ns)
Intel	$44 + 40.62 * length(X)$
N810	$427 + 425.11 * length(X)$
Sparc	$1413 + 2179.75 * length(X)$

Table 5. Timing model given by a linear function, for `unify_variable(X,Y)` when the arguments are lists of integers, and the instruction does not fail.

Bytecode	Tag	Intel (ns)	N810 (ns)	Sparc (ns)
fail	β_{13}	0	0	0
get_constant_atom	β_{15}	32	457	1256
get_constant_int	β_{17}	26	366	1169
get_value	β_{22}	25	244	1106
unequal	β_{41}	11	61	651
unify_variable	β_{43}	121	1065	3867
unify_variable(const1,const2) const1 $\neq$ const2	β_{49}	41	154	697
unify_variable(int,int)	β_{50}	122	1035	3830
unify_variable(list(1),list(1))	β_{51}	338	3227	12229
unify_variable(atm,atm)	β_{52}	127	1126	4282
unify_variable(str(1),str(1))	β_{53}	223	2381	9239

Table 6. Timing model for the WAM instructions. Cost of bytecodes when they fail.

Tables 4 and 6 show the timing model for this WAM and the architectures studied. In the benchmarks used the `is/2` instruction is compiled into basic operations over pairs of numbers. The table shows the corresponding instructions named `arith_*`. We also have separated the cost of the instructions `put_constant`, `get_constant` when they are called for an atom or an integer. Note however, that their cost is very similar in most cases, but this will still help to reduce errors in the estimation. For the `unify_variable` instruction we have also included calibrations for several cases depending on the type and size of the input arguments in order to increase precision. In other cases, as mentioned in 5.4, the execution time of this instruction is not bounded by any a-priori known constant. Since, as also shown in Section 5.4, in our implementation it is possible to use functions instead of constants as timing model for a given instruction, in this table we include in the calibrations two data points for the case when the arguments are lists of integers, and for lists of size (length) 1 and 100 (β_{45} and β_{48} in Table 4). The value for an empty list is the same as for unifying any two equal atoms, i.e., β_{46} in Table 4. Table 5 shows the resulting timing model for `unify_variable` using these values to fit our linear model for this instruction.

Using the timing model shown in Tables 4, 5, and 6, we have performed some experiments with a series of programs on the three platforms (Intel, N810, and Sparc) in order to assess the accuracy of our technique for estimating execution times. The results of these experiments are shown in Tables 8 (Intel), 9 (N810), and 10 (Sparc).

Column **Pr. No.** lists the program identifiers, whose association with the programs and the input data sizes used is shown in Table 7. Column **Cost App.** indicates the type of approximation of the automatically inferred cost functions which estimate execution times (as a function on input data size): upper bound (**U**), lower bound (**L**), or exact (**E**). Such cost functions are shown in column **Cost Function** for the three different platforms considered

No.	Program	Data size
1	append(+A,+,-)	x=length(A)=150
2	evalpol(+A,+X,-)	x=length(A)=100
3	fib(+N,-)	x=N=16
4	hanoi(+N,+,+,+,-)	x=N=8
5	nreverse(+L,-)	x=length(L)=83
6	palindro(+A,-)	x=length(A)=9
7	powset(+A,-)	x=length(A)=11
8	list_diff(+L,+D,-)	x=length(L)=65 y=length(D)=65
9	list_inters(+L,+D,-)	x=length(L)=65 y=length(D)=65
10	substitute(+A,+B,-)	x=term_size(A)=67 y=term_size(B)=80
11	derive(+E,+,-)	x=term_size(E)=75

Table 7. List of program examples used in the experimental assessment.

in our experiments. The variables x and y represent the sizes of the input arguments to the programs which are relevant for the inference of the cost functions. The type of approximation directly depends on the one used by the static analysis described in Section 4 for estimating the number of executed instructions (as a function on input data size). The value **E** means that the lower and upper bound cost functions are the same, and thus, since the analysis is safe, this means that the exact cost function was inferred. Using the cost functions shown in column **Cost Function**, and in order to assess their accuracy, we have also estimated execution times for particular input data for each program and compared them with the observed execution times. These execution times are shown in columns **Est.** and **Obs.** respectively. Column **D.** shows the relative harmonic difference between the estimated and the observed time [2]. The source of inaccuracies in the execution time estimations of our technique come mainly from two sources: the timing model (which gives the execution time estimation of bytecodes, as shown in Tables 4 and 6)) and the static analysis (described in Section 4, which estimates the number of times that the bytecodes are executed, depending on the input data size). Since we are interested in identifying the source(s) of inaccuracies, we have also introduced the column **Prf.** It shows the result of estimating execution times using the timing model and assuming that the static analysis was perfect and obtained a function which provides the *exact* number of times that the bytecodes are executed. This obviously represents the case in which all loss of accuracy must be assigned to the timing model. The "perfect" cost model is obtained from an actual execution by instrumenting the profiler so that it records the number of times each instruction is executed for the application and the particular input data. Column **Pr.D.** shows the relative harmonic difference between **Prf.** and the observed execution time **Obs.**

The upper part of Tables 8, 9, and 10, up to the double line corresponds to examples where an exact cost function for the number of executed bytecodes was automatically inferred by the static analysis (note that, as expected, the values **Est.** and **Prf.** are the same). We can see that with an exact static analysis, the estimated execution times **Est.** are quite precise, which in turn supports the accuracy of our timing model.

It is particularly interesting to compare these results with those which were obtained using a variety of higher-level models in (19). Table 11 provides the standard deviation of the four high-level models of (19) as well as that of the abstract machine-based model presented in this paper, for the Intel platform and our set of bench-

[2] $rel\_harmonic\_diff(x, y) = (x - y)(1/x + 1/y)/2$.

Pr. No.	Cost. App.	Intel (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$0.73x + 0.21$	110	110	113	-2.4	-2.4
2	E	$0.69x + 0.19$	69	69	71	-2.3	-2.3
3	E	$0.69 \cdot 1.6^x + 0.21(-0.62)^x - 0.72$	1525	1525	1576	-3.3	-3.3
4	E	$-0.0042 \cdot 2^x + 0.73x \cdot 2^x - 0.86$	1501	1501	1589	-5.7	-5.7
5	E	$0.37x^2 + 0.49x + 0.12$	2569	2569	2638	-2.7	-2.7
6	E	$0.36 \cdot 2^x + 0.37x \cdot 2^x - 0.24$	1875	1875	2027	-7.8	-7.8
7	E	$0.91 \cdot 2^x + 0.87x - 0.6$	1868	1868	1931	-3.3	-3.3
8	L	$0.66x + 0.2$	43	68	81	-67.2	-17.8
	U	$0.78xy + 1.7x + 0.4$	3414	3569	3640	-6.4	-2.0
9	L	$0.83x + 0.2$	54	79	91	-54.6	-14.8
	U	$0.78xy + 1.7x + 0.4$	3414	3694	4011	-16.2	-8.2
10	L	$2x$	135	142	124	8.6	13.7
	U	$1.4xy + 1.4y + 6.1x + 4.1$	7922	2937	2858	120.6	2.7
11	L	$2.9x$	216	138	111	72.3	22.5
	U	$3x + 3$	226	216	162	34.0	29.5

Table 8. Observed and estimated execution time with cost functions, Intel platform (microseconds).

Pr. No.	Cost. App.	N810 (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$7.8x + 2.7$	1169	1169	1037	12.0	12.0
2	E	$7.8x + 2.7$	786	786	641	20.6	20.6
3	E	$8.3 \cdot 1.6^x + 2.5(-0.62)^x - 8.4$	18333	18333	14496	23.7	23.7
4	E	$0.74 \cdot 2^x + 7.8x \cdot 2^x - 10$	16095	16095	16144	-0.3	-0.3
5	E	$3.9x^2 + 5.7x + 1.6$	27247	27247	28381	-4.1	-4.1
6	E	$4.4 \cdot 2^x + 3.9x \cdot 2^x - 2.9$	20167	20167	20416	-1.2	-1.2
7	E	$9.5 \cdot 2^x + 10x - 6$	19517	19517	19653	-0.7	-0.7
8	L	$7.3x + 2.8$	474	744	640	-30.4	15.1
	U	$8.2xy + 19x + 5.5$	35849	37162	29266	20.4	24.1
9	L	$8.7x + 2.8$	569	839	732	-25.4	13.7
	U	$8.2xy + 19x + 5.5$	35849	38076	29907	18.2	24.4
10	L	$21x$	1399	1475	1068	27.3	32.9
	U	$15xy + 15y + 64x + 43$	85893	30375	25543	153.3	17.4
11	L	$29x$	2190	1423	854	108.7	53.3
	U	$30x + 30$	2306	2193	1342	56.8	51.1

Table 9. Observed and estimated execution time with cost functions, Nokia N810 platform (microseconds).

Model		Deviation
High Level	1	51.17 %
	2	31.06 %
	3	21.48 %
	4	58.45 %
Abs. Machine		4.72 %

Table 11. Comparison between the higher level models and the abstract machine-based model, on the Intel platform.

marks. It can be observed that the results obtained with the abstract machine-based model are more than five times better on the same platform than those obtained using the higher-level models.

With the abstract machine-based model, and for this type of programs we believe that the remaining error comes simply from the accumulated loss of accuracy of the bytecode instruction profiling and expect that making the *timing* model more precise will increase precision even further.

The lower part of Tables 8, 9, and 10 shows programs for which there is no unique value for $\texttt{Time}_p(n)$, where $\texttt{Time}_p(n)$ (as described in Section 4.1) denotes the cost (in time units) of computing a call to program p for an input of size n on a given platform. The reason is that for such programs, the number of instructions executed does not only depend on the input data sizes, but also depends on other characteristics of the input data (e.g., their actual values). Thus, for a given data size, there are actual lower and upper bounds for the cost of the program calls. For this reason, the two observed execution times shown in column **Obs.** for each program have been obtained by running the program with the input data, of the size specified in Table 7, that yield the actual lower and upper bounds to the execution times for such size. In this case, the static analysis infers approximations to such actual lower and upper bound cost functions (**L** and **U** respectively). These predictions are understandably much less accurate in these cases than those in the first part of the table, but still reasonable. In any case, lower bounds and upper bounds tend to be reasonably smaller or bigger than the observed execution times respectively. In general, for the programs in the lower part of the tables with big (absolute) values for **D.**, the (absolute) value for **Pr.D.** is reasonably small. This means that, in those cases, most of the inaccuracy in the estimation of execution times (**Est.**) comes from the static analysis, which does not approximate actual lower and upper bound cost functions accurately enough, and that the timing model used for predicting

Pr. No.	Cost. App.	Sparc (μs) Cost Function	Est.	Prf.	Obs.	D. %	Pr.D. %
1	E	$26x + 7.4$	3906	3906	4670	-18.0	-18.0
2	E	$25x + 7.1$	2543	2543	2985	-16.1	-16.1
3	E	$26 \cdot 1.6^x + 7.8(-0.62)^x - 27$	56828	56828	59120	-4.0	-4.0
4	E	$1.2 \cdot 2^x + 26x \cdot 2^x - 33$	53504	53504	63156	-16.7	-16.7
5	E	$13x^2 + 17x + 4.3$	90973	90973	109849	-19.0	-19.0
6	E	$13 \cdot 2^x + 13x \cdot 2^x - 8.5$	66400	66400	78980	-17.4	-17.4
7	E	$32 \cdot 2^x + 32x - 22$	66224	66224	78151	-16.6	-16.6
8	L	$24x + 7.1$	1574	2458	2991	-68.7	-19.7
	U	$27xy + 62x + 14$	118269	123733	129951	-9.4	-4.9
9	L	$30x + 7.1$	1940	2824	3394	-58.9	-18.5
	U	$27xy + 62x + 14$	118269	127378	133703	-12.3	-4.8
10	L	$68x$	4545	4821	4634	-1.9	4.0
	U	$48xy + 48y + 207x + 140$	277175	101779	111829	103.8	-9.4
11	L	$95x$	7104	4628	4038	59.6	13.7
	U	$98x + 98$	7454	7147	6081	20.5	16.2

Table 10. Observed and estimated execution time with cost functions, Sparc platform (microseconds).

the execution time of bytecodes is reasonably precise. Thus, we believe that using a better static analysis for inferring cost functions which take into account other characteristics of the input data, besides their sizes, would significantly improve the predictions. In any case, there is always a reasonable slack in the precision of the estimations due to the timing measurements and the timing model.

7. Conclusions and Future Work

We have developed a framework for estimating upper and lower bounds on the execution times of logic programs running on a bytecode-based abstract machine. We have shown that working at the abstract machine level allows taking into account low-level issues without having to tailor the analysis for each architecture and platform, and allows obtaining more accurate estimates than with previous approaches, including comparatively accurate upper and lower bound estimations of execution time.

Although the framework has been presented in the context of logic programs, we believe the technique can easily be applied to other languages. This adaptation of the approach, while certainly not trivial, to some extent would actually imply some simplification, since backtracking does not need to be taken into account. For example, analyses have been recently developed for Java bytecode (3) which infer the number of execution steps using similar techniques to those used in logic programming (12; 11; 13). Such analyses could be adapted, following the techniques presented herein, to take into account the bytecode timing information and would then be able to estimate actual execution time for Java programs. Appropriate cost models for Java bytecode are already being developed in (22).

We believe that the more accurate execution time estimates that can be obtained with our technique can be very useful in several contexts including parallelism, compilation, real-time applications, pervasive systems, etc. More concretely, increased timing precision can improve the effectiveness of resource/granularity control in parallel/distributed computing. This belief is based on previous experimental results, where it appeared that, even if improved precision in timing estimates is not essential, it does yield increased speedups. Also, the inferred cost functions can be used to develop automatic program optimization techniques. For example, they can be used for performing self-tuning specialization which compares statically the estimated execution time of different specialized versions (10).

Given that our experimental results are encouraging with respect to actually being able to find more accurate upper and lower bounds to program execution times, the approach may eventually also be used for verification (or falsification) of timing constraints, as in, for example, real-time systems, which was not possible in an accurate way with previous approaches. In fact, our approach (which can be adapted to take also into account destructive assignment, as in (20)) can potentially be used to solve a common problem in current WCET static analysis, where only constant WCET bounds (i.e., non dependent on input data sizes) are inferred. These bounds are not always appropriate since the WCET of a given program often depends on several input parameters, and using an absolute bound, covering all possible situations (i.e., all possible values or sizes of input), produces only a very gross over approximation (15). Substituting the estimated costs of the bytecodes by the actual worst-case costs of the instructions and using our approach, the WCET is expressed as a cost function parameterized by the size or values of input arguments, providing tighter WCET approximations. On the other hand, WCET work has produced more accurate (but, unfortunately, non-freely available) timing models which take into account many low-level parameters (such as cache behavior, pipeline state, etc.) which we have abstracted away in our work. It is clear that a combination of both techniques might be very useful in practice.

References

[1] H. Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Proc. of Static Analysis Symposium (SAS)*, LNCS. Springer-Verlag, July 2008. To appear.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In R. D. Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[4] R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.

[5] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[6] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, Washington, DC, USA*, Apr. 2002.

[7] R. Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[8] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[9] S. Buettcher. Warren's Abstract Machine - A Java Implementation. `http://www.stefan.buettcher.org/cs/wam/index.html`.

[10] S.-J. Craig and M. Leuschel. Self-tuning resource aware specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[11] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[12] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[13] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[14] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[15] A. Ermedahl, J. Gustafsson, and B. Lisper. Experiences from Industrial WCET Analysis Case Studies. In R. Wilhelm, editor, *Proc. Fifth International Workshop on Worst-Case Execution Time (WCET) Analysis*, Palma de Mallorca, July 2005.

[16] G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *PEPM*. ACM Press, 2002.

[17] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[18] E. Y.-S. Hu, A. J. Wellings, and G. Bernat. Deriving java virtual machine timing models for portable worst-case execution time analysis. In *On The Move to Meaningful Internet Systems 2003: OTM 2003Workshops*, volume 2889 of *LNCS*, pages 411–424. Springer, October 2003.

[19] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.

[20] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable Resource Usage Analysis for Java Bytecode. Technical report, University of New Mexico, Department of Computer Science, UNM, January 2008. Submitted for publication.

[21] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

[22] G. Román-Díez and G. Puebla. Java bytecode timing cost models. Technical Report CLIP12/2007.0, Technical University of Madrid, School of Computer Science, UPM, December 2007.

[23] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[24] R. Wilhelm. Timing analysis and timing predictability. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium, FMCO 2004, Leiden, The Netherlands, November 2 - 5, 2004, Revised Lectures*, volume 3657 of *Lecture Notes in Computer Science*, pages 317–323. Springer, 2004.

Customizable Resource Usage Analysis for Java Bytecode

Jorge Navas,[1] Mario Méndez-Lojo,[1] Manuel V. Hermenegildo[1,2]

[1] Dept. of Computer Science, University of New Mexico (USA)
[2] Dept. of Computer Science, Tech. U. of Madrid (Spain) and IMDEA-Software

Abstract. Automatic cost analysis of programs has been traditionally studied in terms of a number of concrete, predefined *resources* such as execution steps, time, or memory. However, the increasing relevance of analysis applications such as static debugging and/or certification of user-level properties (including for mobile code) makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, bytes sent or received by an application, number of files left open, number of SMSs sent or received, number of accesses to a database, money spent, energy consumption, etc. We present a fully automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of *application programmer-definable* resources. In our context, a resource is defined by programmer-provided annotations which state the basic consumption that certain program elements make of that resource. From these definitions our analysis derives functions which return an upper bound on the usage that the whole program (and individual blocks) make of that resource for any given set of input data sizes. The analysis proposed is independent of the particular resource. We also present some experimental results from a prototype implementation of the approach covering an ample set of interesting resources.

1 Introduction

The usefulness of analyses which can infer information about the costs of computations is widely recognized since such information is useful in a large number of applications including performance debugging, verification, and resource-oriented specialization. The kinds of costs which have received most attention so far are related to execution steps as well as, sometimes, execution time or memory (see, e.g., [21, 28, 29, 16, 8, 17, 32] for functional languages, [30, 7, 15, 34] for imperative languages, and [13, 12, 14, 26] for logic languages). These and other types of cost analyses have been used in the context of applications such as granularity control in parallel and distributed computing (e.g., [23]), resource-oriented specialization (e.g., [10, 27]), or, more recently, certification of the resources used by mobile code (e.g., [11, 4, 9, 3, 18]). Specially in these more recent applications, the properties of interest are often higher-level, user-oriented, and application-dependent rather than (or, rather, in addition to) the predefined, more traditional costs such as steps, time, or memory. Regarding the object of certification, in the case of mobile code the certification and checking process is often performed at the bytecode level [22], since, in addition to other reasons of syntactic convenience, bytecode is what is most often available at the receiving (checker) end.

We propose a fully automated framework which infers upper bounds on the usage that a Java bytecode program makes of *application programmer-definable* resources. Examples of such programmer-definable resources are bytes sent or received by an application over a socket, number of files left open, number of SMSs sent or received, number of accesses to a database, number of licenses consumed, monetary units spent, energy consumed, disk space used, and of course, execution steps (or bytecode instructions), time, or memory. In our context, resources are defined by programmers by means of *annotations*. The annotations defining each resource must provide for some user-selected elements corresponding to the bytecode program being analyzed (classes, methods, variables, etc.), a value that describes the cost of that element for that particular resource. These values can be constants or, more generally, functions of the input data sizes. The objective of our analysis is then to statically derive from these elementary costs an upper bound on the amount of those resources that the program as a whole (as well as individual blocks) will consume or provide.

Our approach builds on the work of [13, 12] for logic programs, where cost functions are inferred by solving recurrence equations derived from the syntactic structure of the program. Also, most previous work deals only with concrete, traditional resources (e.g., execution steps, time, or memory). The analysis of [26] is parametric but it is designed for Prolog and works at the source code level, and thus cannot be applied to Java bytecode due to particularities like virtual method invocation, unstructured control flow, assignment, the fact that statements are low-level bytecode instructions, the absence of backtracking (which has a significant impact on the method used in [26]), etc. More importantly, the presentation of [26] is descriptive in contrast to the concrete algorithm provided herein. In [1], a cost analysis is described that does deal with Java bytecode and is capable of deriving cost relations which are functions of input data sizes. However, while the approach proposed can conceptually be adapted to infer different resources, for each analysis developed the measured resource is fixed and changes in the implementation are needed to develop analyses for other resources. In contrast, our approach allows the application programmer to define the resources through annotations in the Java source, and without changing the analyzer in any way. In addition, the presentation in [1] is again descriptive, while herein we provide a concrete, memo table-based analysis algorithm, as well as implementation results.

2 Overview of the Approach

We start by illustrating the overall approach through a working example. The Java program in Fig. 1 emulates the process of sending text messages within a cell phone. The source code is provided here just for clarity, since the analyzer works directly on the corresponding bytecode. The phone (class `CellPhone)` receives a list of packets (`SmsPacket`), each one containing a single SMS, encodes them (`Encoder`), and sends them through a stream (`Stream`). There are two types of encoding: `TrimEncoder`, which eliminates any leading and trailing white spaces, and `UnicodeEncoder`, which converts any special character into its Unicode($\backslash uxxxx$) equivalent. The length of the SMS which the cell phone ultimately sends through the stream depends on the size of the encoded message.

A *resource* is a fundamental component in our approach. A resource is a user-defined notion which associates a basic cost function with some user-selected elements (class, method, statement) in the program. This is expressed by adding Java annotations to the code. The

```
import java.net.URLEncoder;
public class CellPhone {
 SmsPacket sendSms(SmsPacket smsPk,
                   Encoder enc,
                   Stream stm) {
  if (smsPk != null) {
    String newSms = enc.format(smsPk.sms);
    stm.send(newSms);
    smsPk.next=sendSms(smsPk.next,enc,stm);
    smsPk.sms = newSms;
  }
  return smsPk;
 }
}
class SmsPacket{
  String sms;
  SmsPacket next;
}
```

```
interface Encoder{
  String format(String data);
}
class TrimEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=size(s)")
  public String format(String s){
    return s.trim();
  }
}
class UnicodeEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=6*size(s)")
  public String format(String s){
    return URLEncoder.encode(s);
  }
}
abstract class Stream{
  @Cost({"cents","2*size(data)"})
  native void send(String data);
}
```

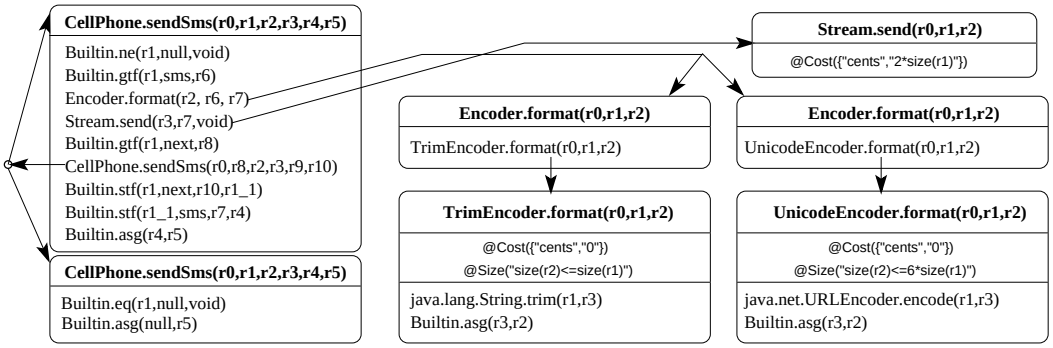


Fig. 1. Motivating example: Java source code and Control Flow Graph

objective of the analysis is to approximate the usage that the program makes of the resource. In the example, the resource is the cost in cents of a dollar for sending the list of text messages, since we will assume for simplicity that the carrier charges are proportional (2 cents/character) to the number of characters sent. This domain knowledge is reflected by the user in the method that is ultimately responsible for the communication (`Stream.send`), by adding the annotation `@Cost({"cents","2*size(data)"})`. Similarly, the formatting of an SMS done in any implementation of `Encoder.format` is free, as indicated by the `@Cost({"cents","0")})` annotation. The analysis understands these resource usage expressions and uses them to infer a safe upper bound on the total usage of the program.

Step 1: Constructing the Control Flow Graph. In the first step, the analysis translates the Java bytecode into an intermediate representation building a Control Flow Graph (CFG). Edges in the CFG connect *block methods* and describe the possible flows originated from conditional jumps, exception handling, virtual invocations, etc. A (simplified) version of the CFG corresponding to our code example is also shown in Fig. 1.

The original `sendSms` method has been compiled into two block methods that share the same signature: class where declared, name (`CellPhone.sendSms`), and number and type of the formal parameters. The bottom-most box represents the base case, in which we re-

turn null, here represented as an assignment of `null` to the return variable r_5; the sibling corresponds to the recursive case. The virtual invocation of `format` has been transformed into a static call to a block method named `Encoder.format`. There are two block methods which are compatible in signature with that invocation, and which serve as proxies for the intermediate representations of the interface implementations in `TrimEncoder.format` and `UnicodeEncoder.format`. Note that the resource-related annotations have been carried through the CFG and are thus available to the analysis.

Step 2: Inference of Data Dependencies and Size Relationships. The algorithm infers in this phase *size relationships* between the input and the output formal parameters of every block method. For now, we can assume that size of (the contents of) a variable is the maximum number of pointers we need to traverse, starting at the variable, until `null` is found. The following equations are inferred by the analysis for the two `CellPhone.sendSms` block methods :

$$\begin{aligned}\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) &\leq 0\\ \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) &\leq 7 \times s_{r_1} - 6 + \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3})\end{aligned}$$

The size of the returned value r_5 is independent of the sizes of the input parameters *this*, *enc*, and *stm* (s_{r_0}, s_{r_2} and s_{r_3} respectively) but not of the size s_{r_1} of the list of text messages *smsPk* (r_1 in the graph). Such size relationships are computed based on *dependency graphs*, which represent data dependencies between variables in a block, and user annotations if available. In the example in Fig. 1, the user indicates that the formatting in `UnicodeEncoder` results in strings that are at most six times longer than the ones received as input `@Size("size(ret)<=6*size(s)")`, while the trimming in `TrimEncoder` returns strings that are equal or shorter than the input (`@Size("size(ret)<=size(s)")`). The equation system shown above is approximated by a recurrence solver included in our analysis in order to obtain the closed form solution $\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 3.5 \times s^2_{r_1} - 2.5 \times s_{r_1}$.

Step 3: Resource Usage Analysis. In the this phase, the analysis uses the CFG, the data dependencies, and the size relationships inferred in previous steps to infer a resource usage equation for each block method in the CFG (possibly simplifying such equations) and obtain closed form solutions (in general, approximated –upper bounds). Therefore, the objective of the resource analysis is to statically derive safe upper bounds on the amount of resources that each of the block methods in the CFG consumes or provides. The result given by our analysis for the monetary cost of sending the messages (`CellPhone.sendSms`) is

$$\begin{aligned}Cost_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) &\leq 0\\ Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) &\leq 12 \times s_{r_1} - 12 + Cost_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3})\end{aligned}$$

i.e., the cost is proportional to the size of the message list (`smsPk` in the source, r_1 in the CFG). Again, this equation system is solved by a recurrence solver, resulting in the closed formula $Cost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 6 \times s^2_{r_1} - 6 \times s_{r_1}$.

3 Intermediate program representation

Analysis of a Java bytecode program normally requires its translation into an intermediate representation that is easier to manipulate. In particular, our decompilation (assisted by the Soot [31] tool) involves elimination of stack variables, conversion to three-address statements, static single assignment (SSA) transformation, and generation of a Control Flow Graph

(CFG) that is ultimately the subject of analysis. The decompilation process is an evolution of the work presented in [25], which has been successfully used as the basis for other (non resource-related) analyses [24]. Our ultimate objective is to support the full Java language but the current transformation has some limitations: it does not yet support reflection, threads, or runtime exceptions. The following grammar describes the intermediate representation; some of the elements in the tuples are named so we can refer to them as *node*.name.

$$
\begin{array}{lll}
CFG & ::= & Block^{+} \\
Block & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{fpars}{:}Id^{+},\mathsf{annot}{:}expr^{*},\mathsf{body}{:}Stmt^{*}) \\
Sig & ::= & (\mathsf{class}{:}Type,\mathsf{name}{:}Id,\mathsf{pars}{:}Type^{+}) \\
Stmt & ::= & (\mathsf{id}{:}\mathbb{N},\mathsf{sig}{:}Sig,\mathsf{apars}{:}(Id|Ct)^{+}) \\
Var & ::= & (\mathsf{name}{:}Id,\mathsf{type}{:}Type)
\end{array}
$$

The Control Flow Graph is composed of *block methods*. A block method is similar to a Java method, with some particularities: a) if the program flow reaches it, every statement in it will be executed, i.e, it contains no branching; b) its signature might not be unique: the CFG might contain several block methods in the same class sharing the same name and formal parameter types; c) it always includes as formal parameters the returned value *ret* and, unless it is static, the instance self-reference *this*; d) for every formal parameter (*input* formal parameter) of the original Java method that might be modified, there is an extra formal parameter in the block method that contains its final version in the SSA transformation (*output* formal parameter); e) every statement in a block method is an invocation, including builtins (assignment `asg`, field dereference `gtf`, etc.), which are understood as block methods of the class `Builtin`.

As mentioned before, there is no branching within a block method. Instead, each conditional `if` *cond* $stmt_1$ `else` $stmt_2$ in the original program is replaced with an invocation and two block methods which uniquely match its signature: the first block corresponds to the $stmt_1$ branch, and the second one to $stmt_2$. To respect the semantics of the language, we decorate the first block method with the result of decompiling *cond*, while we attach $\overline{cond}$ to its sibling. A similar approach is used in virtual invocations, for which we introduce as many block methods in the graph as possible receivers of the call were in the original program. A set of block methods with the same signature *sig* can be retrieved by the function $getBlocks(CFG, sig)$.

User specifications are written using the annotation system introduced in Java 1.5 which, unlike JML specifications, has the very useful characteristic of being preserved in the bytecode. Annotations are carried over to our CFG representation, as can be seen in Fig. 1.

Example 1 We now focus our attention on the two block methods in Fig. 1, which are the result of (de)compiling the `CellPhone.sendSms` method. Input formal parameters r_0, r_1, r_2, r_3 correspond to $this, smsPk, enc$, and stm, respectively. In the case of r_1, the contents of its fields *next* and *sms* are altered by invoking the `stf` and accessed by invoking the `gtf` (abbreviation for `setfield` and `getfield`, respectively) builtin block methods. The output formal parameter r_4 contains the final state of r_1 after those modifications. The value returned by the block methods is contained in r_5. Space reasons prevent us from showing any type information in the CFG in Fig 1. In the case of `Encoder.format`, for example, we say that there are two blocks with the same signature because they are both defined in class `Encoder`, have the same name (`format`) and list of types of formal parameters {`Encoder,String,String`}.

```
resourceAnalysis(CFG, res) {
 CFG←classAnalysis(CFG)
 mt←initialize(CFG)
 dg←dataDependencyAnalysis(CFG, mt)
 for (SCC:SCCs)
   //in reverse topological order
   mt←genSizeEqs(SCC, mt, CFG, dg)
   mt←genResourceUsageEqs(SCC, res, mt, CFG)
 return mt
}
```

```
normalize(Eqs) {
 for (size relation p ≤ e1:Eqs)
   do
     if (expression s appears in e1
         and s ≤ e2 ∈Eqs)
       replace ocurrences of s in e1 with e2
   while there is change
 return Eqs
}
```

Fig. 2. Generic resource analysis algorithm and normalization.

4 The resource usage analysis framework

We now describe our framework for inferring upper bounds on the usage that the Java bytecode program makes of a set of resources defined by the application programmer, as described before. The algorithm in Fig 2 takes as input a Control Flow Graph in the format described in the previous section, including the user annotations that assign elementary costs to certain graph elements for a particular resource. The user also indicates the set of resources to be tracked by the analysis. Without loss of generality we assume for conciseness in our presentation a single resource.

A preliminary step in our approach is a class hierarchy analysis [5, 24], aimed at simplifying the CFG and therefore improving overall precision. Then, another analysis is performed over the CFG to extract data dependencies, as described below. The next step is the decomposition of the CFG into its strongly-connected components. After these steps, two different analyses are run separately on each strongly connected component: a) the size analysis, which estimates parameter size relationships for each statement and output formal parameters as a function of the input formal parameter sizes (Sec. 4.1); and b) the actual resource analysis, which computes the resource usage of each block method in terms also of the input data sizes (Sec. 4.2). Each phase is dependent on the previous one.

The *data dependency analysis* is a dataflow analysis that yields *position dependency graph*s for the block methods within a strongly connected component. Each graph $G = (V, E)$ represents data dependencies between positions corresponding to statements in the same block method, including its formal parameters. Vertexes in V denote positions, and edges $(s_1, s_2) \in E$ denote that s_2 is dependent on s_1 (s_1 is a *predecessor* of s_2). We will assume a `predec` function that takes a position dependency graph, a statement, and a parameter position and returns its nearest predecessor in the graph. Fig. 3 shows the position dependency graph of the `TrimEncoder.format` block method.

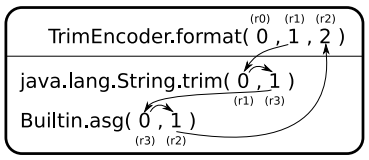


Fig. 3.

4.1 Size analysis

We now show our algorithm for estimating parameter size relations based on the data dependency analysis, inspired by the original ideas of [13, 12]. Also, we provide a concrete algorithm for performing the analysis, rather than the more descriptive presentations of the

related work discussed previously. Our goal is to represent input and output size relationships for each statement as a function defined in terms of the formal parameter sizes. Unless otherwise stated, whenever we refer to a parameter we mean its position.

The size of an input is defined in terms of measures. By *measure* we mean a function that, given a data structure, returns a number. Our method is parametric on measures, which can be defined by the user and attached via annotations to parameters or classes. For concreteness, we have defined herein two measures, int for integer variables, and the *longest path-length* [1] ref for reference variables. The longest path-length of a variable is the cardinality of the longest chain of pointers than can be followed from it. More complex measures can be defined to handle other data types such as cyclic structures, arrays, etc. The set of measures will be denoted by $\mathcal{M}$.

The size analysis algorithm is given in pseudo-code in Fig. 4; its main steps are:

1. Assign an upper bound to the size of every parameter position of all statements, including formal parameters, for all the block methods with the same signature (genSigSize).
2. For a given signature, take the set of size inequations returned by (1) and rename each size relation in terms of the sizes of input formal parameters (normalize).
3. Repeat the first step for every signature in the same strongly-connected component (genSizeEqs).
4. Simplify size relationships by resolving mutually recursive functions, and find closed form solutions for the output formal parameters (genSizeEqs).

Intermediate results are cashed in a memo table mt, which for every parameter position stores measures, sizes, and resource usage expressions defined in the $\mathcal{L}$ language:

$$\begin{array}{lll} \langle expr \rangle & ::= & \langle expr \rangle \langle bin\_op \rangle \langle expr \rangle \mid (\sum \mid \prod) \langle expr \rangle \\ & \mid & \langle expr \rangle^{\langle expr \rangle} \mid log_{num} \langle expr \rangle \mid -\langle expr \rangle \\ & \mid & \langle expr \rangle ! \mid \infty \mid \text{num} \\ & \mid & \text{size}([\langle measure \rangle,]\text{arg(r num)}) \\ \langle bin\_op \rangle & ::= & + \mid - \mid \times \mid / \mid \% \\ \langle measure \rangle & ::= & \textbf{int} \mid \textbf{ref} \mid \ldots \end{array}$$

The size of the parameter at position i in statement $stmt$, under measure m, is referred to as $\texttt{size}(m, stmt, i)$. We consider a parameter position to be *input* if it is bound to some data when the statement is invoked. Otherwise, it is considered an *output parameter position*. In the case of input parameter and output formal parameter positions, an upper bound on that size is returned by getSize (Fig. 4). The upper bound can be a concrete value when there is a constant in the referred position, i.e., when the val function returns a non-infinite value:

Definition 1. *The concrete size value for a parameter position under a particular measure is returned by* $\texttt{val} : \mathcal{M} \times \mathcal{S}tmt \times \mathbb{N} \rightarrow \mathcal{L}$*, which evaluates the* syntactic *content of the actual parameter in that position:*

$$\texttt{val}(m, stmt, i) = \begin{cases} n \text{ if } stmt.\mathsf{apars}_i \text{ is an integer } n \text{ and } m{=}\textbf{int} \\ 0 \text{ if } stmt.\mathsf{apars}_i \text{ is } \texttt{null} \text{ and } m{=}\textbf{ref} \\ \infty \text{ otherwise} \end{cases}$$

If the content of that input parameter position is a variable, the algorithm searches the data dependency graph for its immediate predecessor. Since the intermediate representation is in SSA form, the only possible scenarios are that either there is a unique predecessor

```
genSizeEqs(SCC,mt,CFG,dg) {
 Eqs← ∅^|SCC|
 for (sig: SCC)
   Eqs[sig]←genSigSize(sig,mt,SCC,CFG,dg)
 Sols←recEqsSolver(simplifyEqs(Eqs))
 for (sig:SCC)
   insert(mt,size,sig,Sols[sig])
 return mt
}

genSigSize(sig,mt,SCC,CFG,dg) {
 Eqs← ∅
 BMS←getBlocks(CFG,sig)
 for (bm:BMs)
   Eqs←Eqs ∪ genBlockSize(bm,mt,SCC,dg)
 return normalize(Eqs)
}

genBlockSize(bm,mt,SCC,dg) {
 Eqs← ∅
 for (stmt:bm.body)
   I←stmt input parameter positions
   Eqs←Eqs ∪ genInSize(stmt,I,mt,dg)
   Eqs←Eqs ∪ genOutSize(stmt,mt,SCC)
 K← bm output formal parameter positions
 Eqs←Eqs ∪ genInSize(bm,K,mt,dg)
 return Eqs
}

genInSize(elem,Pos,mt,dg) {
 Eqs← ∅
 for (pos:Pos)
   m←lookup(mt,measure,elem.sig,pos)
   s←getSize(m,elem.id,pos,dg)
   Eqs←Eqs ∪ {size(m,elem.id,pos)≤s}
 return Eqs
}

genOutSize(stmt,mt,SCC) {
 {i_1,...,i_l} ← stmt input positions
 sig←stmt.sig
 {m_{i_1},...,m_{i_l}} ←{lookup(mt,measure,sig,i_1),...,
                         lookup(mt,measure,sig,i_l)}
 {s_{i_1},...,s_{i_l}} ← {size(m_{i_1},stmt.id,i_1),...,
                          size(m_{i_l},stmt.id,i_l)
 Eqs← ∅
 O← stmt output parameter positions
 for (o:O)
   m_o ←lookup(mt,measure,sig,o)
   if (sig∉SCC)
     Size_user ← A^o_sig(s_{i_1},...,s_{i_l})
     Size_alg' ←max(lookup(mt,size,sig,o))
     Size_alg ←Size_alg'(s_{i_1},...,s_{i_l})
     Size_o ←min(Size_user,Size_alg)
   else
     Size_o ← Size^o_sig(m_o,s_{i_1},...,s_{i_l})
   Eqs←Eqs ∪ {size(m_o,stmt.id,o)≤ Size_o}
 return Eqs
}

getSize(m,id,pos,dg) {
 result←val(m,id,i)
 if (result≠∞)
   return result
 else
   if (∃(elem,pos_p)∈ predec(dg,id,pos))
     m_p ←lookup(mt,measure,elem.sig,pos_p)
     if (m=m_p)
       return size(m_p,elem.id,pos_p)
 return ∞
}
```

Fig. 4. The size analysis algorithm

whose size is assigned to that input parameter position, or there is none, causing the input parameter size to be unbounded (∞).

Consider now an output parameter position within a block method, case covered in genOutSize (Fig. 4). If the output parameter position corresponds to a non-recursive invoke statement, either a size relationship function has already been computed recursively (since the analysis traverses each strongly-connected component in reverse topological order), or it is provided by the user through size annotations. In the first case, the size function of the output parameter position can be retrieved from the memo table by using the lookup operation, taking the maximum in case of several size relationship functions, and then passing the input parameter size relationships to this function to evaluate it. In the second scenario, the size function of the output parameter position is provided by the user through size annotations, denoted by the $\mathcal{A}$ function in the algorithm. In both cases, it will able to return an explicit size relation function.

8

Example 2 We have already shown in the `CellPhone` example how a class can be annotated. The `Builtin` class includes the assignment method `asg`, annotated as follows:

```
public class Builtin {
  @Size{"size(ret)<=size(o)"}
  public static native Object asg(Object o);
  // ...rest of annotated builtins
}
```

which results in equation $\mathcal{A}^1_{\mathsf{asg}}(\mathtt{ref}, \mathtt{size}(\mathtt{ref}, asg, 0)) \leq \mathtt{size}(\mathtt{ref}, asg, 0)$.

If the output parameter position corresponds to a recursive invoke statement, the size relationships between the output and input parameters are built as a symbolic size function. Since the input parameter size relations have already been computed, we can establish each output parameter position size as a function described in terms of the input parameter sizes.

At this point, the algorithm has defined size relations for all parameter positions within a block method. However, those relations are either constants or given in terms of the immediate predecessor in the dependency graph. The algorithm rewrites the equation system such that we obtain an equivalent system in which only formal parameter positions are involved. This process, called *normalization*, is shown in Fig. 2

After normalization, the analysis repeats the same process for all block methods in the same strongly-connected component (SCC). Once every component has been processed, the analysis further simplifies the equations in order to resolve mutually recursive calls among block methods within the same SCC in the simplifyEqs procedure.

In the final step, the analysis submits the simplified system to a recurrence equation solver (recEqsSolver, called from genSizeEqs) in order to obtain approximated upper-bound closed forms. The interesting subject of how the equations are solved is beyond the scope of this paper (see, e.g., [33]). Our implementation does provide a dedicated implementation (an evolution of the solver of the Caslog system [12]) which covers a reasonable set of recurrence equations such as first-order and higher-order linear recurrence equations in one variable with constant and polynomial coefficients, divide and conquer recurrence equations, etc. In addition, the system has interfaces to external solvers such as Purrs [6] or Mathematica.

Example 3 We now illustrate the definitions and algorithm with an example of how the size relations are inferred for the two `CellPhone.sendSms` block methods (Fig. 1), using the ref measure for reference variables. We will refer to the k-th occurrence of a statement $stmt$ in a block method as $stmt_k$, and denote `CellPhone.sendSms`, `Encoder.format`, and `Stream.send` by `sendSms`, `format`, and `send` respectively. Finally, we will refer to the size of the input formal parameter position i, corresponding to variable r_i, as s_{r_i}.

The main steps in the process are listed in Fig. 5. The first block of rows contains the most relevant size parameter relationship equations for the recursive block method, while the second block of rows corresponds to the base case. These size parameter relationship equations are constructed by the analysis by first following the algorithm in Fig. 4, and then normalizing them (expressing them in terms of the input formal parameter sizes s_{r_i}). Also, in the first block of rows we observe that the algorithm has returned $6 \times \mathtt{size}(\mathtt{ref}, format, 1)$ as upper bound for the size of the formatted string, $\mathsf{max}(\mathsf{lookup}(mt, \mathtt{size}, format, 2))$. The result is the maximum of the two upper bounds given by the user for the two implementations for `Encoder.format` since `TrimEncoder.format` eliminates any leading and trailing white

Size parameter relationship equations (normalized)

$$
\begin{array}{ll}
\texttt{size}(\texttt{ref}, ne, 0) & \leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, ne, 1) & \leq \texttt{val}(\texttt{ref}, ne, 1) \leq 0 \\
\texttt{size}(\texttt{ref}, gtf_1, 0) & \leq \texttt{size}(\texttt{ref}, ne, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, gtf_1, 2) & \leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_1, 0), \_) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, format, 1) & \leq \texttt{size}(\texttt{ref}, gtf_1, 2) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, format, 2) & \leq \texttt{max}(\texttt{lookup}(mt, \texttt{size}, format, 2))(\texttt{size}(\texttt{ref}, format, 2)) \\
 & \leq \texttt{max}(s_{r1}, 6 \times s_{r1})(s_{r_1} - 1) \\
 & \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, send, 1) & \leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, gtf_2, 0) & \leq \texttt{size}(\texttt{ref}, gtf_1, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, gtf_2, 2) & \leq \mathcal{A}^2_{gtf}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf_2, 0), \_) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, sendSms, 1) & \leq \texttt{size}(\texttt{ref}, gtf_2, 2) \leq s_{r1} - 1 \\
\texttt{size}(\texttt{ref}, sendSms, 5) & \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, \_, \texttt{size}(\texttt{ref}, sendSms, 1), \_, \_) \\
 & \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_1, 0) & \leq \texttt{size}(\texttt{ref}, gtf_2, 0) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, stf_1, 2) & \leq \texttt{size}(\texttt{ref}, sendSms, 5) \leq \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_1, 3) & \leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_1, 0), \_, \texttt{size}(\texttt{ref}, stf_1, 2)) \\
 & \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_2, 0) & \leq \texttt{size}(\texttt{ref}, stf_1, 3) \leq s_{r1} + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, stf_2, 2) & \leq \texttt{size}(\texttt{ref}, format, 2) \leq 6 \times (s_{r1} - 1) \\
\texttt{size}(\texttt{ref}, stf_2, 3) & \leq \mathcal{A}^3_{stf}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf_2, 0), \_, \texttt{size}(\texttt{ref}, stf_2, 2)) \\
 & \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, asg, 0) & \leq \texttt{size}(\texttt{ref}, stf_2, 3) \\
 & \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) \\
\texttt{size}(\texttt{ref}, asg, 1) & \leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \\
 & \leq 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3})
\end{array}
$$

$$
\begin{array}{ll}
\texttt{size}(\texttt{ref}, eq, 0) & \leq \texttt{size}(\texttt{ref}, sendSms, 1) \leq s_{r1} \\
\texttt{size}(\texttt{ref}, eq, 1) & \leq \texttt{val}(\texttt{ref}, eq, 1) \leq 0 \\
\texttt{size}(\texttt{ref}, asg, 0) & \leq \texttt{val}(\texttt{ref}, asg, 0) \leq 0 \\
\texttt{size}(\texttt{ref}, asg, 1) & \leq \mathcal{A}^1_{asg}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \leq 0
\end{array}
$$

Output parameter size functions for builtins (provided through annotations)

$$\mathcal{A}^2_{\texttt{gtf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, gtf, 0), \_) \leq \texttt{size}(\texttt{ref}, gtf, 0) - 1$$
$$\mathcal{A}^1_{\texttt{asg}}(\texttt{ref}, \texttt{size}(\texttt{ref}, asg, 0)) \leq \texttt{size}(\texttt{ref}, asg, 0)$$
$$\mathcal{A}^3_{\texttt{stf}}(\texttt{ref}, \texttt{size}(\texttt{ref}, stf, 0), \_, \texttt{size}(\texttt{ref}, stf, 2)) \leq \texttt{size}(\texttt{ref}, stf, 0) + \texttt{size}(\texttt{ref}, stf, 2)$$

Simplified size equations and closed form solution

$$\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \begin{cases} 0 & \text{if } s_{r1} = 0 \\ 7 \times s_{r1} - 6 + \mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1} - 1, s_{r2}, s_{r3}) & \text{if } s_{r1} > 0 \end{cases}$$

$$\mathcal{S}ize^{r5}_{sendSms}(\texttt{ref}, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq 3.5 \times s^2_{r1} - 2.5 \times s_{r1}$$

Fig. 5. Size equations example

spaces (thus the output is at most as bigger as the input), whereas `UnicodeEncoder.format` converts any special character into its Unicode equivalent (thus the output is at most six times the size of the input), a safe upper bound for the output parameter position size is given by the second annotation.

```
genResourceUsageEqs(SCC, res, mt, CFG) {
  Eqs ← ∅^|SCC|
  for (sig : SCC)
    Eqs[sig] ← genSigRU(sig, res, mt, SCC, CFG)
  Sols ← recEqsSolver(simplifyEqs(Eqs))
  for (sig : SCC)
    insert(mt, cost, max(Sols[sig]))
  return mt
}

genSigRU(sig, res, mt, SCC, CFG) {
  Eqs ← ∅
  BMs ← getBlocks(CFG, sig)
  for (bm : BMs)
    body ← bm.body
    Cost_body ← 0
    for (stmt : body)
      Cost_stmt ← genStmtRU(stmt, res, mt, SCC)
      Cost_body ← Cost_body + Cost_stmt
    Cost_bm ← genBlockRU(bm, res, mt)
    Eqs ← Eqs ∪ {Cost_bm ≤ Cost_body}
}

genStmtRU(stmt, res, mt, SCC) {
  {i_1, ..., i_k} ← stmt input parameter positions
  {s_i1, ..., s_ik} ←
      {max(lookup(mt, size, stmt.sig, i_1)), ...,
       max(lookup(mt, size, stmt.sig, i_k))}
  if (stmt.sig ∉ SCC)
    Cost_user ← A_stmt.sig(res, s_i1, ..., s_ik)
    Cost_alg' ← lookup(mt, cost, res, stmt.sig)
    Cost_alg ← Cost_alg'(s_i1, ..., s_ik)
    return min(Cost_alg, Cost_user)
  else return Cost(stmt.sig, res, s_i1, ..., s_ik)
}

genBlockRU(bm, res, mt) {
  {i_1, ..., i_l} ← bm input formal parameter positions
  {s_i1, ..., s_il} ←
      {lookup(mt, size, bm.id, i_1), ...,
       lookup(mt, size, bm.id, i_l)
  return Cost(bm.id, res, s_i1, ..., s_il)
}
```

Fig. 6. The resource usage analysis algorithm

In the particular case of builtins and methods for which we do not have the code, size relationships are not computed but rather taken from the user `@Size` annotations. These functions are illustrated in the third block of rows. Finally, in the fourth block of rows we show the recurrence equations built for the output parameter sizes in the block method and in the final row the closed form solution obtained.

4.2 Resource usage analysis

The core of our framework is the resource usage analysis, whose pseudo code is shown in Fig 6. It takes a strongly-connected component of the CFG, including the set of annotations which provide the application programmer-provided resources and cost functions, and calculates an resource usage function which is an upper bound on the usage made by the program of those resources. The algorithm manipulates the same memo table described in Sec. 4.1 in order to avoid recomputations and access the size relationships already inferred.

The algorithm is structured in a very similar way to the size analysis (which also allows us to draw from it to keep the explanation within space limits): for each element of the strongly-connected component the algorithm will construct an equation for each block method that shares the same signature representing the resource usage of that block. To do this, the algorithm will visit each invoke statement. There are three possible scenarios, covered by the genStmtRU function. If the signatures of caller and callee(s) belong to the same strongly-connected component, we are analyzing a recursive invoke statement. Then, we add to the body resource usage a symbolic resource usage function, in an analogous fashion to the case of output parameters in recursive invocations during the size analysis.

The other scenarios occur when the invoke statement is non-recursive. Either a resource usage function $Cost_{alg}$ for the callee has been previously computed, or there is a user anno-

tation $Cost_{usr}$ that matches the given signature, or both. In the latter case, the minimum between these two functions is chosen (i.e., the most precise safe upper bound assigned by the analysis to the resource usage of the non-recursive invoke statement).

Example 4 The call (sixth statement) in the upper-most `CellPhone.sendSms` block method matches the signature of the block method itself and thus it is recursive. The first four parameter positions are of input type. The upper-bound expression returned by genStmtRU is $\underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})$. Note that the input size relationships were already normalized during the size analysis. Now consider the invocation of `Stream.send`. The resource usage expression for the statement is defined by the function $\mathcal{A}_{send}(\$, \_, 6\times(s_{r1}-1))$ since the input parameter at position one is at most six times the size of the second input formal parameter, as calculated by the size analysis in Fig. 5. Note also that there is a resource annotation `@Cost({"cents","2*size(r1)"})` attached to the block method describing the behavior of $\mathcal{A}_{send}$ and yielding the expression $Cost_{user} = 12\times(s_{r1}-1)$. On the other hand, the absence of any callee code to analyze –the original method is native– results in $Cost_{alg} = \infty$. Then, the upper bound obtained by the analysis for the statement is $\min(Cost_{alg}, Cost_{user}) = Cost_{user}$.

At this point, the analysis has built a resource usage function (denoted by $Cost_{body}$) that reflects the resource usage of the statements within the block. Finally, it yields a resource usage equation of the form $Cost_{block} \leq Cost_{body}$ where $Cost_{block}$ is again a symbolic resource usage function built by replacing each input formal parameter position with its size relations in that block method. These resource usage equations are simplified by calling simplifyEqs and, finally, they are solved calling recEqsSolver, both already defined in Sec. 4.1. This process yields an (in general, approximate, but always safe) closed form upper bound on the resource usage of the block methods in each strongly-connected component. Note that given a signature the analysis constructs a closed form solution for every block method that shares that signature. These solutions approximate the resource usage consumed in or provided by each block method. In order to compute the total resource usage of the signature the analysis returns the maximum of these solutions yielding a safe global upper bound.

Example 5 The resource usage equations generated by our algorithm for the two `sendSms` block methods and the "$" resource (monetary cost of sending the SMSs) are listed in Fig. 7. The computation is partially based on the size relations in Fig. 5. The resource usage of each block method is calculated by building an equation such that the left part is a symbolic function constructed by replacing each parameter position with its size (i.e., $\underset{sendSms}{Cost}(\$, s_{r0}, s_{r1}, s_{r2}, s_{r3})$ and $\underset{sendSms}{Cost}(\$, s_{r0}, 0, s_{r2}, s_{r3})$), and the rest of the equation consists of adding the resource usage of the invoke statements in the block method. These are calculated by computing the minimum between the resource usage function inferred by the analysis and the function provided by the user. The equations corresponding to the recursive and non-recursive block methods are in the first and second row, respectively. They can be simplified (third row) and expressed in closed form (fourth row), obtaining a final upper bound for the charge incurred by sending the list of text messages of $6 \times s_{r1}^2 - 6 \times s_{r1}$.

5 Experimental results

We have completed an implementation of our framework, and tested it for a representative set of benchmarks and resources. Our experimental results are summarized in Table 1. Column

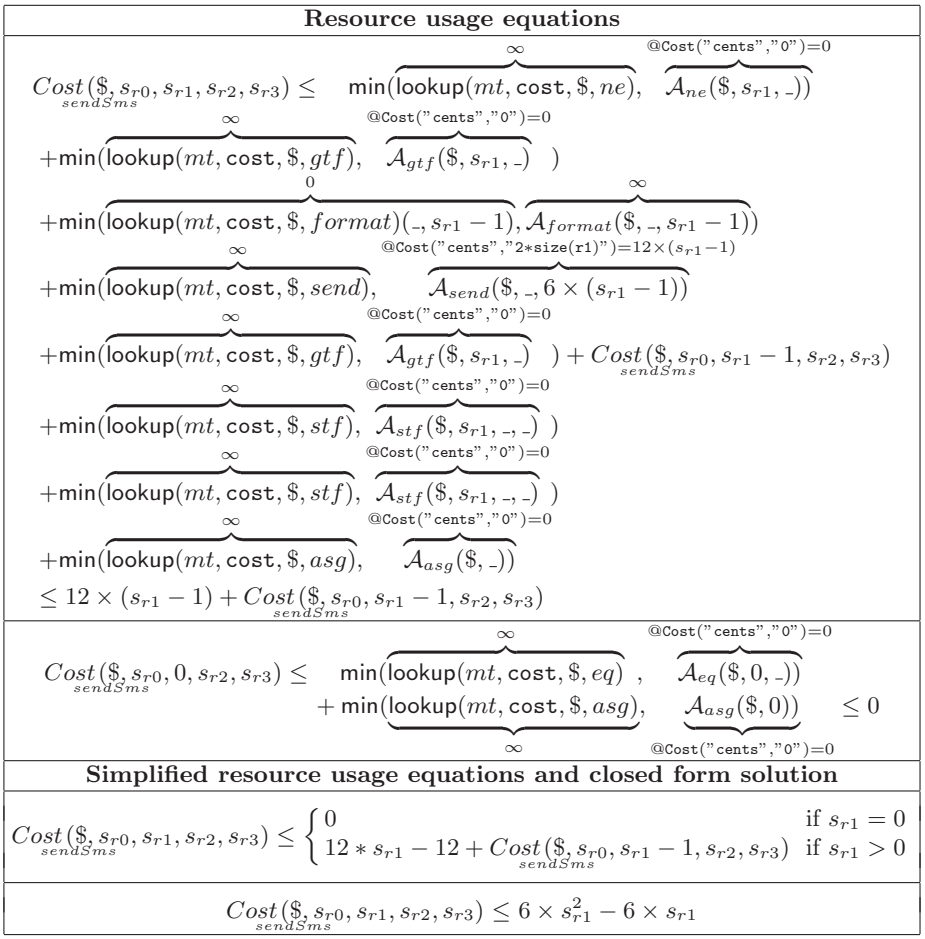


Fig. 7. Resource equations example

Program provides the name of the main class to be analyzed. Column **Resource(s)** shows the resource(s) defined and tracked. Column t_s shows the time (in milliseconds) required by the size analysis to construct the size relations (including the data dependency analysis and class hierarchy analysis) and obtain the closed form. Column t_r lists the time taken to build the resource usage expressions for all method blocks and obtain their closed form solutions. t provides the total times for the whole analysis process. Finally, column **Resource Usage Func.** provides the upper bound functions inferred for the resource usage. For space reasons, we only show the most important (asymptotic) component of these functions, but the analysis yields concrete functions with constants.

Regarding the benchmarks we have covered a reasonable set of data-structures used in object-oriented programming and also standard Java libraries used in real applications. We have also covered an ample set of application-dependent resources which we believe can be relevant in those applications. In particular, not only have we represented high-level resources such as cost of SMS, bytes received (including a coarse measure of bandwidth, as

Program	Resource(s)	t_s	t_r	t	Resource Usage Func.	
BST	Heap usage	250	22	367	$O(2^n)$	$n \equiv$ tree depth
CellPhone	SMS monetary cost	271	17	386	$O(n^2)$	$n \equiv$ packets length
Client	Bytes received and	391	38	527	$O(n)$	$n \equiv$ stream length
	bandwidth required				$O(1)$	—
Dhrystone	Energy consumption	602	47	759	$O(n)$	$n \equiv$ int value
Divbytwo	Stack usage	142	13	219	$O(log_2(n))$	$n \equiv$ int value
Files	Files left open and	508	53	649	$O(n)$	$n \equiv$ number of files
	Data stored				$O(n \times m)$	$m \equiv$ stream length
Join	DB accesses	334	19	460	$O(n \times m)$	$n, m \equiv$ records in tables
Screen	Screen width	388	38	536	$O(n)$	$n \equiv$ stream length

Table 1. Times of different phases of the resource analysis and resource usage functions.

a ratio of data per program step), and files left open, but also other low-level (i.e., bytecode level) resources such as stack usage or energy consumption. The resource usage functions obtained can be used for several purposes. In program *Files* (a fragment characteristic of operating system kernel code) we kept track of the number of file descriptors left open. The data inferred for this resource can be clearly useful, e.g., for debugging: the resource usage function inferred in this case ($O(n)$) denotes that the programmer did not close $O(n)$ file descriptors previously opened. In program *Join* (a database transaction which carries out accesses to different tables) we decided to measure the number of accesses to such external tables. This information can be used, e.g., for resource-oriented specialization in order to perform optimized checkpoints in transactional systems. The rest of the benchmarks include other definitions of resources which are also typically useful for verifying application-specific properties: *BST* (a generic binary search tree, used in [2] where a heap space analysis for Java bytecode is presented), *CellPhone* (extended version of program in Figure 1), *Client* (a socket-based client application), *Dhrystone* (a modified version of a program from [20] where a general framework is defined for estimating the energy consumption of embedded JVM applications; the complete table with the energy consumption costs that we used can be found there), *DivByTwo* (a simple arithmetic operation), and *Screen* (a MIDP application for a cellphone, where the analysis is used to make sure that message lines do not exceed the phone screen width). The benchmarks also cover a good range of complexity functions ($O(1), O(log(n), O(n), O(n^2) \ldots, O(2^n), \ldots$) and different types of structural recursion such as simple, indirect, and mutual. The code for these benchmarks and a demonstrator are available at `http://www.cs.unm.edu/~jorge/RUA`.

6 Conclusions

We have presented a fully-automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. Our analysis derives a vector of functions, one for each defined resource. Each of these functions returns, for each given set of input data sizes, an upper bound on the usage that the whole program (and each individual method) make of the corresponding resource. Important novel aspects of our approach are the fact that it allows the application programmer to define the resources to be tracked by writing simple resource descriptions via source-level annotations, as well as the fact that we have provided a concrete analysis algorithm and report on an

implementation. The current results show that the proposed analysis can obtain non-trivial bounds on a wide range of interesting resources in reasonable time. Another important aspect of our work, because of its impact on the scalability, precision, and automation of the analysis, is that our approach allows using the annotations also for a number of other purposes such as stating the resource usage of external methods, which is instrumental in allowing modular composition and thus scalability. In addition, our annotations allow stating the resource usage of any method for which the automatic analysis infers a value that is not accurate enough to prevent inaccuracies in the automatic inference from propagating. Annotations are also used by the size and resource usage analysis to express their output. Finally, the annotation language can also be used to state specifications related to resource usage, which can then be proved or disproved based on the results of analysis following, e.g., the scheme of [19] thus finding bugs or verifying (the resource usage of) the program.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.
2. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.
3. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.
4. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.
5. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *Proc. of OOPSLA'96, SIGPLAN Notices*, 31(10):324–341, October 1996.
6. R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.
7. I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case execution-time analysis. In *5th IEEE Int'l. Symp. on Object-oriented Real-time Distributed Computing*, Apr. 2002.
8. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.
9. Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming (ESOP)*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.
10. S.J. Craig and M. Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *Proc. of PPDP'05*, pages 23–34. ACM Press, 2005.
11. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.
12. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.
13. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. PLDI'90*, pages 174–188. ACM, June 1990.
14. S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*. MIT Press, 1997.
15. J. Eisinger, I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proc. of DDECS*. IEEE Computer Society, 2006.

16. G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM Press, 2002.
17. B. Grobauer. Cost recurrences for DML programs. In *Int'l. Conf. on Functional Programming*, pages 253–264, 2001.
18. M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.
19. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
20. Sébastien Lafond and Johan Lilius. Energy consumption analysis for two embedded java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.
21. D. Le Metayer. ACE: An Automatic Complexity Evaluator. *TOPLAS*, 10(2), 1988.
22. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
23. P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21:715–734, 1996.
24. M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.
25. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.
26. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP*, LNCS, 2007.
27. G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of PPDP'06*, pages 261–271. ACM Press, 2006.
28. M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.
29. D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.
30. Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In *Perspectives Workshop: Design of Systems with Predictable Behaviour*, 2004.
31. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.
32. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.
33. H. S. Wilf. *Algorithms and Complexity*. A.K. Peters Ltd, 2002.
34. R. Wilhelm. Timing analysis and timing predictability. In *Proc. FMCO*, LNCS. Springer-Verlag, 2004.