University of California
Santa Barbara

The ZARF Architecture for Recursive Functions

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Joseph Earl McMahan

Committee in charge:

Professor Timothy Sherwood, Chair
Professor Chandra Krintz
Professor Ben Hardekopf
Professor Yuan Xie

June 2019

The Dissertation of Joseph Earl McMahan is approved.

Professor Chandra Krintz

Professor Ben Hardekopf

Professor Yuan Xie

Professor Timothy Sherwood, Committee Chair

June 2019

The ZARF Architecture for Recursive Functions

Dedicated to every errant embedded system and every erroneous C program

Acknowledgements

First, I have to acknowledge the collaborators that made this research possible. I am but a humble architect, and though I could design a crazy CPU at the onset of the project, I had no notion of how to prove a property of anything, or what types really where, or what a monad does. Zarf never would have gotten off the ground without the programming languages side of things, motivating it, shaping it, and giving it life.

Jared Roesch deserves half the credit for the original Zarf ISA design, though at the time we were calling it the "h-machine." We worked through so many problems together in a short span — what instructions should look like, what the capabilities of software should be, how much influence Haskell should have — and at the end of only a few months, we had a mostly-broken CPU and a mostly-broken compiler. Our first paper was rejected. Those were good times.

Jared would leave for the University of Washington and the Zarf hardware and ISA would be redesigned, but I still think of him as the co-creator of the platform. Any success Zarf eventually finds is due in part to him.

Once we began to actually develop applications and try to start proving things, Lawton Nichols took over the PL side of things. He gave Zarf its first formal semantics, and did the first proofs on the platform. He moved on to bigger and better projects, but he helped Zarf get off the ground.

Michael Christensen stepped in, and though he had never used Coq before, or written a typesystem or language semantics, his ability to learn and pick things up on his own made him cover ground fast. Most of the formalisms in this thesis are the product of his labor. He's the co-author on all the Zarf papers to-date, and they wouldn't exist without him and his exceptional work ethic.

Of course, it would be remiss of me to not thank and acknowledge my family and friends for the support they've given me in this seemingly endless series of grad school years. Thanks belongs especially to my boyfriend, Jonathan Lee, who is always there after a long day of failed papers to remind me that there's a world outside of grad school and a life waiting to be lived.

The last and largest acknowledgement goes to my advisor, Tim Sherwood — a professor on the fringe, who gives radical ideas a home. From his willingness to let new students join projects, to the freedom he gives in what we work on, to the support and hands-on advising he's not afraid to administer, Tim is not only a top-notch researcher and professor, but a good human too. And that deserves to be acknowledged.

Curriculum Vitæ
Joseph Earl McMahan

Education

2019	Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara.
2018	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2013	B.A. in Physics, Princeton University.

Publications

Bouncer: Static Program Analysis in Hardware
Joseph McMahan, Michael Christensen, Kyle Dewey, Ben Hardekopf, and Timothy Sherwood. *International Symposium on Computer Architecture* (ISCA), June 2019. Phoenix, AZ.

Safer Program Behavior Sharing Through Trace Wringing
Deeksha Dangawl, Weilong Cui, **Joseph McMahan**, and Timothy Sherwood. *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), April 2019. Providence, RI.

Information Leakage in Arbiter Protocols
Nestan Tsiskaridze, Lucas Bang, **Joseph McMahan**, Tevfik Bultan, and Timothy Sherwood. *International Symposium for Verification and Analysis* (ATVA), October 2018. Los Angeles, CA.

Hiding Intermitten Information Leakage with Architectural Support for Blinking
Alric Althoff, **Joseph McMahan**, Luis Vega, Scott Davidson, Timothy Sherwood, Michael Taylor, and Ryan Kastner. *International Symposium on Computer Architecture* (ISCA), June 2018. Los Angeles, CA.

Charm: A Language for Closed-form High-level Architecture Modeling
Weilong Cui, Yongshan Ding, Deeksha Dangwal, Adam Holmes, **Joseph McMahan**, Ali Javadi-Abhari, Geroge Tzimpragos, Frederic Chong, Timothy Sherwood. *International Symposium on Computer Architecture* (ISCA), June 2018. Los Angeles, CA.

An Architecture for Analysis
Joseph McMahan, Michael Christensen, Lawton Nichols, Jared Roesch, Sung-Yee Guo, Ben Hardekopf, and Timothy Sherwood. *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences* (IEEE Micro - top pick), May-June 2018.

A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation
John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, **Joseph McMahan**, and Timothy Sherwood. *Proceedings of the International Conference on Field-Programmable Logic and Applications* (FPL), September 2017. Ghent, Belgium.

Challenging On-Chip SRAM Security with Boot-State Statistics
Joseph McMahan, Weilong Cui, Liang Xia, Jeff Heckey, Frederic T. Chong, and Timothy Sherwood. *IEEE International Symposium on Hardware Oriented Security and Trust* (HOST), May 2017. McLean, VA.

An Architecture Supporting Formal and Compositional Binary Analysis
Joseph McMahan, Michael Christensen, Lawton Nichols, Jared Roesch, Sung-Yee Guo, Ben Hardekopf, and Timothy Sherwood. *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), April 2017. Xi'an, China.

Abstract

The ZARF Architecture for Recursive Functions

by

Joseph Earl McMahan

For highly critical workloads, the legitimate fear of catastrophic failure leads to both highly conservative design practices and excessive assurance costs. One import part of the problem is that modern machines, while providing impressive performance and efficiency, are difficult to reason about formally. We explore the microarchitectural support needed to create a machine with a compact and well defined semantics, lowering the difficulty of sound and compositional reasoning across the hardware/software interface. Specifically, we explore implementation options for a machine organization devoid of programmer-visible memory, registers, or state update, built instead around function primitives. The resulting machine can be precisely and mathematically described in a brief set of semantics, which we quantitatively and qualitatively demonstrate is amenable to software proofs at the binary level.

As time continues, we become increasingly dependent on computational devices for all facets of our lives — including our health, well-being, and safety. Many of these devices live "in the wild," in resource-constrained and/or embedded environments, without access to large software stacks and heavy language run-times. At the same, increasing trends in heterogeneity in computer architecture gives the opportunity for new cores in system-on-chips (SoC's) that provide support for increasing critical workloads. We propose an implementation and provide an evaluation of such a device, the Zarf Architecture for Recursive Functions (Zarf), providing a interface of reduced *semantic* complexity at the ISA level, giving designers a platform amenable to reasoning and static analysis. The described prototype is comparable to normal embedded systems in size and resource usage, but it is far easier to reason about programs

according to analysis. This can serve both resource-constrained devices, providing a new hardware platform, and resource-rich SoC's, serving as a small, trusted co-processor that can handle critical workloads in the larger ecosystem.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded devices are ubiquitous. These small computers run the encryption in our mobile devices, help drive our cars, keep our planes in the air, and monitor our heart rates and glucose levels. Tolerance for error varies across different types of devices, with the most critical systems having virtually no room for failure. Development in this region is at the unfortunate intersection of needing low-level implementations that can run on embedded hardware and needing high-level guarantees from languages that support functional correctness and verified properties.

The critical nature of these systems — automotive, medical, cryptographic, avionic — is at odds with the increasing complexity of embedded software overall: even simple devices can easily include an HTTP server for monitoring purposes. Traditional processor interfaces are inherently global and stateful, making the task of isolating and verifying critical application subsets a significant challenge. Architectural extensions have been proposed that enhance the power, performance, and functionality of systems, but no modern architecture has been designed with formal program analysis as a core motivating principle.

Software modeling and verification has found success on top of large language stacks on which one can reason about code at an appropriately high level — however, this approach is unsuited to the world of embedded and critical systems, which often cannot handle large language runtimes, nor can they tolerate excessively large trusted computing bases (TCB) for modeling and proof frameworks. High-level, functional languages offer a remarkable ability to reason about the behavior of programs, but for low-level embedded systems reasoning must be done at the assembly level to give a full picture of the code that will actually execute. At a high level, in a language designed for verification, reasoning typically requires relying on a language run-time that can be prohibitive for resource-constrained or real-time embedded systems, and/or require the assumption that thousands of lines of untrusted code in the language stack are correct.

One might propose that an alternative to an untrusted language stack is to use a verified compiler. Working in a high-level source language and then, in a verified way, translating to executable machine code is a great and useful work-flow. However, though a verified compiler might assign a semantic model to a subset of an ISA, guaranteeing that source program semantics are preserved in compilation, it does not aid in program-specific reasoning or analysis — i.e., whether or not you use a verified compiler, reasoning about C is still subject to reasoning about pointers, memory mutation, and countless imperative, effectful behaviors. Perhaps more importantly, the task of *developing* the verified compiler in the first place is a huge, daunting endeavor. A platform that supports reasoning at the machine level aids in both binary analysis *and* verified compiler development.

Formal semantics has proven to be a powerful tool in both the theory and practice of programming languages, enabling precise mathematical reasoning across myriad languages — from a pure functional λ-calculus, to higher level languages such as C, Java, and even Javascript. The formal semantics of entire instruction sets have also been examined with many thousands of lines of high order logic [1]. However, while modern architectures may be mathematically modeled instruction-by-instruction, *composibility* and (more generally) *semantic simplicity* are much harder to come by. We instead explore the ramifications of designing a machine with the first-order constraint of being highly amenable to reasoning using existing techniques. To this end, we take inspiration from pure functional programming languages in creating a machine interface that allows for *equational reasoning*; that is, an executable program in our ISA can be reasoned about algebraically in much the same way as a Haskell or Coq program.

The mismatch between the semantics of the typical machines and the semantics useful for formal specification has a serious cost. As Ahmdal, Blaauw, and Brooks note, "The real value of an information system is properly measured by answers-per-month, not bits-per-microsecond," going on to explain "The former criterion required specific advances to increase throughput for a given internal speed, to shorten turnaround time for a given throughput, and to make the whole complex of machines and programming systems easier to use" [2]. If one is seeking a full system with provable mathematical properties (for controlling embedded systems or as a sub-component of a traditional system), an evaluation must consider both the performance of the system and the ease with which such proofs can be obtained. In fact, we would argue, in *many* critical embedded systems (e.g. automobiles, drones, medical devices, etc.) the engineering cost of a full verification of critical sub-components, such as a crypto algorithm or safety monitor, dwarfs the incremental cost of a couple hundred thousands gates or a few microwatts. If there is a chance to apply a limited amount of extra resources to make provability easier, it is a research topic worth exploring.

For most systems, reasoning at the *binary* level provides an otherwise unparalleled view on program behavior that is matched only by the difficulty of reasoning at such a low level of abstraction. Previous attempts to develop formal models of ISAs typically model only a tiny subset of instructions and behaviors [3], make simplifying assumptions about what programs can do [4], or are outright unwieldy for high-level proofs [1]. Current costs reflect this difficulty: the cost of assurance alone (ignoring development) for highly-critical systems (EAL 6+) now exceeds $1,000 per line of code, while the strictest flight control software is supposedly

$10,000 per line [5].

The problem of creating and verifying correct embedded systems has been attempted with many horizontal approaches — verified compilers, verified operating systems, verified assembly constructs — but these do not necessarily operate in tandem, nor does each level operate without caveats. For embedded critical systems, the property of full-system verification remains elusive. Working across the stack, for a vertical solution, requires integrating hardware, software, and analysis techniques together. By combining a novel architecture of compact and precise semantics with existing analysis methodologies, we can formally verify end-to-end critical behaviors of full software systems.

To this end, we propose a system where the critical code can execute at the assembly level in a way that is very similar to the underlying computational model that proof and reasoning systems are already built upon. Under such a mode of computation, properties such as isolation, composition, and correctness can be reasoned about incrementally, rather than monolithically. By using a new architecture designed for software verification, we can avoid the problem inherent to reasoning about traditional architectures. The absence of global, implicit, or mutable state from the ISA upwards means that components can be composed without unanticipated interference. Having the notion of a function built-in to the hardware provides a convenient basis for modeling program behavior. The compact and mathematical semantics means that reasoning can be done without relying on large language stacks or compilers: we are able to successfully model programs at the binary level without resorting to incomplete or approximate models of the ISA.

Of course, most ISAs already do present a simplified machine model to the software: fetch a single instruction, execute that instruction (modifying the machine state accordingly), and repeat. However, this contract is inherently *stateful* in nature. Even in RISC machines (which seek to minimize instruction complexity) the continuous mutation of the machine's state such as memory, registers, flags, etc. creates implicit dependencies between disparate places in the code, requiring one to reason about the entire program, rather than independent pieces modularly.

In contrast, a pure functional instruction set would use side-effect free functions for all operations, never exposing internal micro-architectural state such as memory and registers to *any* level of software. Local reasoning about behavior becomes easier, as every dependency becomes well-defined and explicit.

The idea of a stateless, pure functional machine is perhaps counter-intuitive at first — the notion of mutable state is baked into the way we *think* about ISAs at a fundamental level. Architectural registers take on new values as a program executes; memories are updated byte-wise and word-wise; flags are set; program counters are saved and restored; stack pointers are incremented and decremented; etc. In fact, it isn't clear at the outset whether a pure functional machine can be designed that is practical and effective for executing real programs. However, as we will show, it is indeed possible to define a semantically simple architecture whose interface is made up entirely of functional pieces amenable to equational reasoning. Furthermore, we will demonstrate that it is possible to realize such a design in a small and practical hardware implementation, on which real programs can be written and executed.

In summary, by leveraging principles from pure functional programming, it is possible to create a usable computer architecture whose instruction set architecture is mathematically well-defined. The architecture can support formal analysis at the binary level, allow for hardware-based type checking, and can be implemented with both lazy and strict evaluation semantics.

1.1 Permissions and Attributions

1. The original draft of Zarf was the result of extensive collaboration with Jared Roesch. The first draft of the ISA and hardware have since been replaced, but he helped lay an important foundation.

2. The original semantics of Zarf were crafted by Lawton Nichols. They do not appear here, but his aid in further drafts was invaluable.

3. The big-step operational semantics in Figures 3.2 and 3.3 in Section 3.3 were developed mostly by Michael Christensen, and have previously appeared in [6].

4. The Zarf typesystem of Chapter 7 was the result of collaboration with Michael Christensen. Namely, the static semantics of Figure 7.1 was the product of his labor. It previously appeared in [7].

5. The fuzzing methodology of testing discussed in Section 7.5 was the result of work done by Kyle Dewey, and previously appeared in [7].

6. The integrity type system mentioned in Section 6.3, and covered more extensively in Appendix B, was completed in collaboration with Michael Christensen and is reproduced here with permission.

Chapter 2

Background and Related Work

2.1 Verification

Our security type system draws from the work done on the Secure Lambda (SLam) calculus by Heintze and Riecke [8] and its further development by Abadi et al. in their Core Calculus of Dependency [9]. It also draws inspiration from Volpano [10] et al., who created a type system for secure information flow for an imperative block-structured language. By showing that their type system is sound, they show the absence of flow from high-security data to lower-security output, or similarly, that low-security data does not affect the integrity of higher-security data. Other seminal work on secure information flow via the formulation of a type system include Denning [11], Goguen [12], Pottier's information flow for ML [13], and Sabelfeld and Myers's survey on language-based information flow security [14].

Productive, expressive high-level languages that are also purely functional are excellent source platforms for Zarf. Even languages like Haskell, though, can have occasional weaknesses that can lead to runtime type-errors. Subsets such as Safe Haskell [15] shore up these loopholes, and provide extensions for sandboxing arbitrary untrusted code. Zarfprovides isolation guarantees at the ISA level and does not require runtimes, but relies on languages like

Safe Haskell for source code development.

Previous work on ISA-level verification has often involved either simplified or incomplete models of the architecture. These can be in the form of new "idealized" assembly-like languages: Yu et al. [16] use Coq to apply Hoare-style reasoning for assembly programs written in a simple RISC-like language. They also provide a certified memory management library for the machine they describe. Chlipala presents Bedrock, a framework that facilitates the implementation and verification of low-level programs [17], but limits available memory structures.

Verification has also been done for subsets of existing machines. For example, a "substantial" subset of the Motorola MC68020 interface is modeled and used to mechanically prove the correctness of quicksort, GCD, and binary search [18]; other examples include a formalization of the SPARC instruction set, including some of the more complex properties, such as branch delay slots [19]; and subsets of x86 [20]. One of the biggest efforts to date has been a formal model of the ARMv7 ISA using a monadic specification in HOL4 [1]. Moore developed Piton, a high level assembly language and a verified compiler for the FM8502 microprocessor [21], which complemented the verification work done on the FM8502 implementation [22]. These are large efforts because of the difficulty in reasoning about imperative systems. At higher levels of abstraction, entire journal issues have been devoted to works on Java bytecode verification [23].

In addition to proofs on machine code for existing machines, it is also possible to define new assembly abstractions that carry useful information. Typed assembly as an intermediate representation was previously identified as a method for Proof-Carrying Code [24], where machine-checked proofs guarantee properties of a program [25]. Typed assemblies and intermediate representations have seen extensive use in the verification community [26, 17, 27, 28] and have been extended with dependent types [29], allowing for more expressive programs and proofs at the assembly level.

SeL4, a fully verified OS kernel [30], required several person-years of verification work.

Verification is done at higher levels of abstraction to make the problem tractable, like modeling memory behavior at the C level [31].

Verified compilers are a popular topic in the verification community [32, 33, 34, 35], the most well-known example being CompCert [36], a verified C compiler. Verified compilers are usually equipped with a proof of semantics preservation, demonstrating that for every output program, the semantics match those of the corresponding input program. A verified compiler does not provide tools for, nor simplify the process of doing, program-specific reasoning. One needs a secondary tool-chain for reasoning about source programs, such as the Verified Software Toolchain (VST) [37] for CompCert. These frameworks often have a great cost, mandating the use of sophisticated program logics, such as higher-order separation logic in VST, in order to fully reason about possible program behaviors. Further, in many systems, it's possible that not all source code is available; without being able to reason about binary programs, guarantees made on a piece of the source program (and preserved by the verified compiler) may be violated by other components. Extensions to support combining the output of verified compilers, such as separate compilation and linking, are still an active research area [38, 39]. As work on verified compilers requires a semantic model of the ISA, it is complemented by our work, which gives complete and formal semantics for an ISA.

Previous work at the intersection of verification and biological systems has attempted to improve device reliability through modeling efforts. This includes work that formulates real-time automata models of the heart for device testing [40], formal models of pacing systems in Z notation [41], quantitative and automated checking of the interaction of heart-pacemaker automata to verify pacemaker properties [42], and semi-formal verification by combining platform-dependent and independent model checking to exhaustively check the state space of an embedded system [43]. Our work is complemented by verification works such as these that refine device specification by taking into account device-environment interactions.

2.2 Architecture

The SECD Machine [44] is an abstract machine for evaluating arithmetic expressions based in the lambda calculus, designed in 1963 as a target for functional language compilers. It describes the concept of “state” (consisting of a Stack, Environment, Control, and Dump) and transitions between states during said evaluation. Interpreters for SECD run on standard, imperative hardware. Hardware implementations of the SECD Machine have been produced [45], which explore the implementation of SECD at the RTL and transistor level, but present the same high-level interface. The SECD hardware provides an abstract-machine semantics, indicating how the machine state changes with each instruction. Our verification layer makes machine components fully transparent, presenting a higher-level small-step operational semantics, where instructions affect an abstract environment, and a big-step semantics, which immediately reduces each operation to a value. These latter two versions of the semantics are more compact, precise, and useful for typical program-level reasoning.

The SKI Reduction Machine [46] was a hardware platform whose machine code was specially designed to do reductions on simple combinators, this being the basis of computation. Like our verification layer, it was garbage-collected and its language was purely applicative. The goal was to create a machine with a fast, simple, and complete ISA. The choice to use the “simpler” SKI model means that machine instructions are a step removed from the typically function-based, mathematical methods of reasoning about programs. Our functional ISA, while also simple and complete, chooses somewhat more robust instructions based on function application; though the implementation is more complicated, modern hardware resources can easily handle the resulting state machine, giving a simple ISA that is sufficiently high-level for program reasoning.

The most famous work on hardware support for functional programming was on Lisp Machines [47, 48, 49]. Lisp machines provided a specialized instruction set and data format to efficiently implement the most common list operations used in functional programming. For example, Knight [49] describes a machine with instructions for Lisp primitives such as CAR and CADR, and also for complex operations like CALL and MOVE. While these these machines partially inspired this work, Lisp Machines are not directly applicable to the problem at hand. Side-effects on global state at the ISA level are critical to the operation of these machines, and while fast function calls are supported, the stepwise register-memory-update model common to more traditional ISAs is still a foundation of these Lisp Machine ISAs. In fact, several commercial Lisp Machine efforts attempted to capitalize on this fact by building Lisp Machines as a thin translation layer on top of other processors.

Flicker also dealt with architectural support for a smaller TCB in the presence of untrusted, imperative code, but did so with architectural extensions that could create small, independent, trusted bubbles within untrusted code [50]. Previous works such as NoHype [51] dealt with raising the level of abstraction of the ISA and factoring software responsibilities into the hardware. Our verification layer shares some of these characteristics, but deals with verification instead of virtualization, as well as being a complete, self-contained, functional ISA.

Previous work has explored the security vulnerabilities present in many embedded medical devices, as well as zero-power defenses against them [52, 53, 54]. The focus of our work is analysis and correctness properties, and we do not deal with security.

2.3 Typed Assembly

When dealing with typed assembly, the most prominent works are TAL [55] and its extensions TALx86 [56], DTAL [57], STAL [58], and TALT [59]. In TAL, they demonstrate the ability to safely convert high-level languages based on System F (e.g. ML) into a typed target assembly language, maintaining type information through the entire compilation process. Their target typed assembly provides several high-level abstractions like integers, tuples, and code labels, as well as type constructors for building new abstractions.

TALx86 is a version of IA32, extending TAL to handle additional basic type constructors (like records and sums), recursive types, arrays, and higher-order type constructors. They use dependent types to better support arrays; the size of an array becomes part of its type, and they introduce singleton types to track integer values of arbitrary registers or memory words. TAL provides a way to ensure that high-level properties like type- and memory-safety are preserved after compiler transformations and optimizations have taken place.

Unlike TAL, our type system was co-designed with hardware checking in mind — a distinction that greatly impacts the type system design. It allows for binary encoding of types and empowers the target machine, rather than the program authors, to decide if a program is malformed. TAL requires a complex, compile-time software typechecker, as opposed to our small, load-time hardware checker. Our type system operates on an actual machine binary and not an intermediate language.

The eventual target of TALx86 is untyped assembly code (assembled by their MASM assembler into x86). The types are not carried in the binary and are not visible to the device that ultimately runs the code. Though useful, a device cannot trust that the program it has been given has been vetted; therefore, bad binaries can still run on TAL's target machines.

In SAFE [60], the authors develop a machine design that dynamically tracks types at the hardware level. Using these types along with hardware tags assigned to each word, their system works to prove properties about information-flow control and non-interference. They claim that the generic architecture of their system could facilitate efforts related to memory and control-flow safety in further work.

There has also been important work in binary analysis, which seeks to recover information from arbitrary binaries to make sound and useful observations. For example, Code Surfer [61] is a tool that analyzes executables to observe run-time and memory usage patterns, to determine whether a binary may be malicious. Work on binary type reconstruction in particular seeks to recover type information from binaries. In one work [62], they recover high-level C types from binaries via a conservative inference-based algorithm. In Retypd [63], Noonan et al. develop a technique for inferring complex types from binaries, including polymorphic and recursive structures, as well as pointer, subtyping, and type qualifier information. Caballero et al. [64] provide a survey of the many approaches to binary type inference and reconstruction.

Static safety via on-card bytecode verification in a JavaCard [65] is an interesting line of work with a similar goal to our approach. However a hardware implementation can be verified non-bypassable in a way that is much harder to guarantee for software. The Java type system is known to both violate safety [66, 67] and be undecidable [68] which makes it a far more difficult target for static analysis and, we would argue, nearly impossible to implement in hardware directly.

At the intersection of hardware and functional programming, previous works have synthesized hardware directly from high-level Haskell programs [69], even incorporating pipelined dataflow parallelism [70]. Run-time solutions to help enforce memory management for C programs have been proposed at the software level [71], as well as in hardware-enforced implementations [72, 73]; these provide run-time, rather than static, checks.

Chapter 3

Interface and Instruction Set Architecture

3.1 Introduction

The following subsections describe the interface and construction of Zarf, including the reasons we take an approach much closer to the lambda calculus underlying most software proof techniques, how we capture this style of execution in an instruction set, the semantics for that instruction set, and more practical considerations such as I/O, errors, and ALU functions.

3.1.1 Design Goals

Normal, imperative architectures have been difficult to model, and the task of composing verified components is still an open problem [38, 39]. We identify the following features as undesirable and counterproductive to the goal of assembly-level verification:

1. Large amounts of global machine state (memory, stack, registers, etc.) directly accessible to instructions, all of which must be modeled and managed in every proof, and which inhibit modularity: state may be modified by code you haven't seen.

2. The mutable nature of machine state, which prevents abstraction and composition when

reasoning about functions or sets of instructions.

3. A large number of instructions and features: a complete model must incorporate all of them (e.g., fully modeling the behavior of the ARMv7 was 6,500 lines of HOL4 [1]).

4. Arbitrary control flow, which often requires complex and approximate analyses to soundly determine possible control flows [74].

5. Unenforced function call conventions, meaning one must prove that every function respects the convention.

6. Implicit instruction semantics, such as exceptions where “jump” becomes “jump and update registers on certain conditions.”

To avoid these traits, we design an interface that is small, explicit in all arguments, and completely free of state manipulation and side effects — with the exception of I/O, which is necessary for programs to be useful. Without explicit state to reference (memory and registers), standard imperative operations become impossible, and we must raise the level of abstraction. Instead of imperative instructions acting as the building blocks of a program, our basic unit is the *function*. This is a major departure from a typical imperative assembly, where the notion of a “function” is a higher-level construct consisting of a label, control flow operations, and a calling convention enforced by the compiler — but which has no definition in the machine itself. By bringing the definition of functions to the ISA level, they become not just callable “methods” that serve to separate out independent routines, but are actually strict functions in the mathematical sense: they have no side effects, never mutate state, and simply map inputs to outputs. This change allows us to attach precise and formal semantics to the ISA operations.

3.1.2 Description and Semantics

Zarf's functional ISA is effectively an **a)** untyped, **b)** lambda-lifted, **c)** administrative normal form (ANF) lambda calculus. Those limitations are a result of the implementation being done in real hardware: **a)** to avoid the complexity of a hardware type checker, the assembly is untyped; **b)** because every function must live somewhere in the global instruction memory, only top-level declarations of functions are allowed (lambda-lifted); **c)** because the instruction words are fixed-width with a static number of operands, nested expressions are not allowed and every sub-expression must be bound to its own variable (ANF). One bit is attached to values at runtime to distinguish primitive integers from function objects; this prevents malformed code from placing the machine in an invalid state. Instructions use De Bruijn indices[1] to refer to data elements; since the references are localized and cannot refer to any global state, together with immutability it enforces referential transparency.

All words in the machine are 32-bits. Each binary program starts with a magic word, a word-length integer *N* stating how many functions are contained in the program, and then a sequence of *N* functions. Each function starts with an informational word that lets the machine know the "fingerprint" of the function (including the number of arguments expected and how many locals will be used) and a word-length integer *M* to specify that the body of the function is *M* words long. The remaining *M* words of the function are then composed entirely of the individual instructions of the machine.

Each function, as it is loaded, is given a unique and sequential identifier starting at 0x100. These function identifiers are the only globally visible state in the system and serve as both a kind of name and a kind of pointer back to the code. Other functions can refer to, test, and apply arguments to function identifiers. There are two varieties of function identifiers: those that refer to full functions that contain a body of code, and "constructors," which have no body

[1] This is effectively using the stack offset of each variable in the local frame; variables are placed on the "stack" automatically and static numbering is easily determined.

at all. Constructors are essentially stub functions and cannot be executed. However, just like other functions, you can apply arguments to them. These special function identifiers thus can serve as a "name" for software data types, where arguments are the composed data elements. (In more formal terms, you can use our constructors to implement algebraic data types.)

The words defining the body of a function are built out of just three instructions: `let`, `case`, and `result`, which we will describe below. Unlike RISC instructions, `let` and `case` can be multiple words long (depending on the number of arguments and branches, respectively). However, unlike most CISC instructions, each piece of the variable length instruction is also word-aligned and trivial to decode.

Zarf has no programmer-visible registers or memory addresses, but instructions will still need to reference particular data elements. Instructions can refer to data by its source and index, where the source is one of a predefined set — e.g., *local* and *arg*, which serve a purpose similar to the stack on a traditional machine. The *local* and *arg* indices might be analogous to stack offsets, while the actual addresses themselves are never visible.

3.1.3 Built-In Functions, I/O, and Errors

ALU operations are, for the most part, already purely mathematical functions — they just map inputs to an output. Zarf's functional ISA is built around the notion of function calls, so no new mechanism or instructions are needed to use the hardware ALU. Invoking a hardware "add" is the same as invoking a program-supplied function. In our prototype, function indices less than 256 (0x100) are reserved for hardware operations; the first program-supplied function, main, is 0x100, with functions numbered up from there. During evaluation, if the machine encounters a function with an index less than 0x100, it knows to invoke the ALU instead of jumping to a space in instruction memory.

The only two functions with side-effects in the system, input and output, are also primitive functions. The input function takes one argument (a port number) and returns a single word from that port; the output function takes two arguments, a port and a value, and writes its result to the port, returning the value written. Since data dependencies are never violated in function evaluation, software can ensure I/O operations always occur in the right order even in a pure functional environment by introducing artificial data dependencies; this is the principle underlying the I/O monad [75, 76], used prominently in languages like Haskell.

In a purely functional system there are no side effects, and thus no notion of an "exception". For program-defined functions, this just requires that every branch of every case return a value (that value could be a program-defined error). However, some invalid conditions resulting from a malformed program can still occur at runtime. To respect the purely functional system, these must cause a non-effectful result that is still distinguishable from valid results. Our solution is to define a "runtime error constructor" in the space of reserved functions. Every function, both hardware- and software-defined, can potentially return an instance of the error constructor. The ISA semantics are undefined in these error cases, because it's very easy to avoid — compiling from any Hindley-Milner typechecked language will guarantee the absence of runtime type errors [77, 78].

3.2 ISA

The `let` instruction applies a function to arguments and assigns it a *local* identifier, which are sequential numbers that begin at 0 for each function. The first word in the `let` instruction indicates a function identifier or closure object and the number of argument words that follow. Each argument word consists of a source and an index, indicating where the argument value should be pulled from and which value to pull. Note that unlike a function "call", `let` does not immediately change the control flow or force evaluation of arguments; rather it creates a new structure in memory (closure) tying the code (function identifier) to the data (arguments), which, when finally needed, can actually be evaluated (using lazy evaluation semantics). Additionally, the `let` instruction allows partial application, meaning that new functions (but not function identifiers) can be dynamically produced by applying a function identifier to some, but not all, of its arguments.

The `case` instruction provides pattern-matching for control flow. It takes a value, then makes a set of equality comparisons, one for each "pattern" provided. The first word of the case instruction indicates a piece of data to evaluate. As we need an actual value, this is the point in execution that forces evaluation of structures created with `let` — however, it is evaluated only enough to get a value with which comparisons can be made; specifically, until it results in either an integer or a constructor object[2]. The first word of the instruction is followed by additional words encoding patterns against which to match the argument. A `pattern_literal` argument contains both an integer value to match against and a number of words *n* to skip if the match fails. If the case value exactly equals the literal value in the instruction word, then execution continues with the next instruction. If not equal, *n* instruction words are skipped, which brings execution to the next branch of the case. The `pattern_cons` takes a similar argument, but the integer value indicates a function identifier to match against. The match succeeds if and only if 1) the case instruction was attempting to match a constructor, not an integer, and 2) the constructor is the same as specified in the `pattern_cons`. Skips are handled in the same way. Finally, a matching `pattern_else` is required for every `case` which will be executed when no other matches are found (and demarcates the end of the `case` instruction encoding). Case/pattern sequences not adhering to the encoding described are malformed and invalid — e.g., you cannot skip to the middle of a branch, or have a case without an else branch.

The `result` instructions are a single word, indicating a single piece of data that the current function should yield. Every branch of every function must terminate with a result instruction

[2] More precisely, evaluation of that argument will always produces a result in Weak Head-Normal Form (WHNF), but never a lambda abstraction.

(disallowing re-convergent branches means the simple pattern-skip mechanism is all that is necessary for control flow). Functions that do not produce a value do not make sense in an environment without side effects, and so are disallowed. After a `result`, control flow passes to the `case` instruction where the function result was required.

As mentioned above, since we have abandoned the ability to refer to machine memory explicitly, we must provide a new mechanism for a function to refer to data. We use a system of scoped references which are relative to (and available only to) the current execution context. That way, a function does not refer to a set of argument registers — it simply refers to argument indices, and the hardware maps these to actual values (whether in registers or memory is an implementation detail obfuscated from the binary program). The order of `let` instructions in a function determines the numbering of the local variables. References to local objects is done via these implicit indices. The implementation of such a system just requires enumerating the possible categories of data a function may refer to (e.g. argument vs. locally-defined variable), and then supporting references to each via scoped indices. When the assembly-level code is mapped to binary, the names are replaced with the source value (the category of data being accessed) and an index (which piece of data to use). In this way, the machine enforces referential transparency: functions can only use their arguments and locals, so the same function called with the same arguments always evaluates to the same result.

Possible sources (*src*) for an instruction are arguments, locals, literals, casefields, and itables. Each *src* has an accompanying *index*, which has a different meaning depending on the source.

1. **Arguments** are passed in to a function; all arguments must be present for the function to be executed. Depending on the implementation, arguments may be stored in the heap (in a closure) or on the stack. Accessing argument *n* will fetch arg *n* from the list, wherever it is stored.

2. **Locals** are created locally within the body of a function. Every function application (via a `let` instruction) implicitly creates a new local variable. A number scheme, similar to De Bruijn indices, is used: starting with 0, each `let` instruction allocates the next available "slot" in this scope. So the `let` instructions will allocate local 0, then local 1, then local 2, etc. The *index* given refers to which local to take.

3. **Literals** are used to inject integer literals into a program. For this *src* type, the *index* gives the actual value of the integer to be used. In an Argument Word, this limits the integer values to those that fit in 29 bits; inside of a `let` instruction, the limit is 16 bits.

4. **Casefields** refer to the fields of the most recently matched constructor. It only has meaning inside of a `case` statement that matches against a constructor; otherwise, it is semantically invalid. The *index* tells which field of the matched constructor to use, so, for example, if we match against `Cons` (the list constructor), `casefield 0` will get the head of the list and `casefield 1` will get the tail.

5. **Itables**, short for Info Tables, refer to the static function signatures that make up part of a binary program. Every function has an itable (Info Table), which contains the information in the Function Header specified in the binary (is constructor, arity, and number locals), as well as the address of the entry point (used internally by the execution hardware, but inaccessible to software). For the itable *src*, the *index* gives the itable number of the function you want to apply on. It is only valid to use the itable *src* inside of a `let` instruction; you cannot pass an itable as a argument. Each program's Main function is always itable 0x100, with program functions goin up from there; the space of functions below 0x100 is reserved for hardware operations.

$$x \in \mathit{Variable} \quad n \in \mathbb{Z} \quad \mathit{fn}, \mathit{cn} \in \mathit{Name} \quad \oplus \in \mathit{PrimOp}$$

$$
\begin{aligned}
p \in \mathit{Program} &::= \overrightarrow{\mathit{decl}}\ \textbf{fun}\ \mathrm{main}\ \textbf{=}\ e \\
\mathit{decl} \in \mathit{Declaration} &::= \mathit{cons} \mid \mathit{func} \\
\mathit{cons} \in \mathit{Constructor} &::= \textbf{con}\ \mathit{cn}\ \vec{x} \\
\mathit{func} \in \mathit{Function} &::= \textbf{fun}\ \mathit{fn}\ \vec{x}\ \textbf{=}\ e \\
e \in \mathit{Expression} &::= \mathit{let} \mid \mathit{case} \mid \mathit{res} \\
\mathit{let} \in \mathit{Let} &::= \textbf{let}\ x\ \textbf{=}\ \mathit{id}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \\
\mathit{case} \in \mathit{Case} &::= \textbf{case}\ \mathit{arg}\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e \\
\mathit{res} \in \mathit{Result} &::= \textbf{result}\ \mathit{arg} \\
\mathit{br} \in \mathit{Branch} &::= \mathit{cn}\ \vec{x}\ \Rightarrow\ e \mid\ n\ \Rightarrow\ e \\
\mathit{id} \in \mathit{Identifier} &::= x \mid \mathit{fn} \mid \mathit{cn} \mid \oplus \\
\mathit{arg} \in \mathit{Argument} &::= n \mid x
\end{aligned}
$$

Figure 3.1: The Abstract Syntax for Zarf's functional ISA. A program is a set of function and constructor declarations, where functions are composed solely of `let`, `case`, and `result` expressions, and constructors are tuples with unique names. Case expressions contain branches and serve as the mechanism for both control flow and deconstruction of constructor forms. An arrow over any metavariable (e.g. $\vec{x}$) signifies a list of zero or more elements. $\oplus$ refers to a function that is implemented in hardware (such as ALU operations); though the execution of the function invokes a hardware unit instead of a piece of software, the functional interface is identical to program-defined functions.

3.3 Big Step Semantics

For our ISA to be useful for formal reasoning, we must formalize what the instructions actually "mean." For different languages and systems, this normally manifests as a set of *semantics* describing the system, on top of which proofs and reasoning can be done. Semantics come in many different forms; they can be axiomatic, where actions are expressed as logical predicates on program state; they can be denotational, where the semantics are built up with denotations of mathematical objects; they can be operational, where we describe the meaning of different operations of the system.

Though each category has its uses, we will concern ourselves with the last: operational semantics. Specifically, we will outline in this section a big-step operational semantics, which details *the result* of each operation in the system — as opposed to a small-step operational semantics, which would explain *how* each operation is executed. For example, given the expression $2 + 2 + 3 \times 4$, a big-step style would compute the answer "all at once," yielding 16, while a small-step style would reduce it to $2 + 2 + 12$, which reduces to $4 + 12$, which reduces to 16 [79].

Small- and big-step operational semantics are each useful for reasoning about program effects, results, and behavior. Clearly, small-step is more useful when we care about how answers are derived, while big-step has the possibility of being more concise if we care only about what expressions reduce to. So, for example, in proving that a program is correct, we care that it always produces a correct answer, but may not necessarily care about all the intermediate steps; in contrast, if we wanted to reason about the efficiency of the program, the intermediate steps are crucial.

In Figures 3.2 and 3.3, we detail with a set of induction rules precisely what the instructions in a Zarf program "do," formally, with a big-step operational semantics. Figure 3.1 shows the abstract syntax, which will help in deciphering the operational semantics by describing how a program is composed. Below, we detail in words what Figures 3.1, 3.2, and 3.3 mean[3]. It is important to note that the abstract syntax and semantics are **untyped**. Later, in section 7, we will add types at the ISA level; even then, however, the types can be compiled away and the program can be de-sugared down to the level described in this section.

3.3.1 Abstract Syntax

The first line of Figure 3.1 details the atoms of the system: x is a variable, n is a natural number, fn and cn (function and constructor names) are names, and $\oplus$ is a primitive operation. From there, we proceed down the figure. A program is made up of a series of 0 or more declarations, along with a `main` function, which is defined by an expression. Each declaration is either a constructor or a function; a constructor has a name and a series of 0 or more variables (the fields of the constructor). A function has a name, a series of 0 or more arguments, and an expression that defines it. An expression can be a *let*, *case*, or *res* operation.

A *let* instruction takes a variable x to bind the result to, an identifier, a series of 0 or more arguments, and an expression for which the binding is in scope[4]. This expression in the tail contains the computations for the remainder of the function. A *case* instruction takes a single argument on which to case, a series of 0 or more branches, and an else branch. A *res* (result) instruction simply indicates a single argument to return.

The branches *br* of a case statement take two forms: the first $\left(cn\overrightarrow{x} \Rightarrow e\right)$ indicates the name of a constructor *cn*, a series of variables to which the fields of the matching constructor are bound, and an expression to evaluate if the branch is a match. The other form $(n \Rightarrow e)$

[3] These explanations are provided so that readers can penetrate the occasionally abstruse, though precise, mathematical notation by moving through the figures piece and piece and reading explanations. Readers familiar with the notation, or those who do not feel a need to understand the semantic explanations in depth, should feel free to skip to the next section (Section 3.5).

[4] For ease of expression, the abstract syntax describes a recursive structure of programs, where each *let* instruction contains the remainder of the function in its tail. The actual Zarf binaries are not recursive, but are operationally equivalent.

$$c \in \mathit{Constructor} = \mathit{Name} \times \overrightarrow{\mathit{Value}} \quad \mathit{clo} \in \mathit{Closure} = (\lambda\vec{x}.e) \times \overrightarrow{\mathit{Value}}$$

$$v \in \mathit{Value} = \mathbb{Z} \uplus \mathit{Constructor} \uplus \mathit{Closure} \quad \rho \in \mathit{Env} = \mathit{Variable} \to \mathit{Value}$$

$$\frac{\vdash e \Downarrow v}{\vdash \overrightarrow{\mathit{decl}}\ \textbf{fun}\ \text{main}\ \textbf{=}\ e \Downarrow v}\ (\textsc{program}) \qquad \frac{v = \rho(\mathit{arg})}{\rho \vdash \textbf{result}\ \mathit{arg} \Downarrow v}\ (\textsc{result})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{\mathit{arg}}) \quad v_2 = \texttt{applyCn(}\mathit{cn}\texttt{,}\vec{v}_1\texttt{)} \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \mathit{cn}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \Downarrow v_3}\ (\textsc{let-con})$$

$$\frac{\begin{array}{c} \mathit{fn} \notin \{\textbf{getint}, \textbf{putint}\} \quad \textbf{fun}\ \mathit{fn}\ \vec{x}_2\ \textbf{=}\ e_2 \in \overrightarrow{\mathit{decl}} \quad \vec{v}_1 = \rho(\overrightarrow{\mathit{arg}}) \\ v_2\ =\ \texttt{applyFn(}(\lambda\vec{x}_2.e_2, [])\texttt{,}\vec{v}_1\texttt{,}\rho\texttt{)} \quad \rho[x_1 \mapsto v_2]\ \vdash\ e_1\ \Downarrow\ v_3 \end{array}}{\rho \vdash \textbf{let}\ x_1\ \textbf{=}\ \mathit{fn}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e_1 \Downarrow v_3}\ (\textsc{let-fun})$$

$$\frac{v_1 = \rho(x_2) \quad \vec{v}_2 = \rho(\overrightarrow{\mathit{arg}}) \quad v_3 = \texttt{applyFn(}v_1\texttt{,}\vec{v}_2\texttt{,}\rho\texttt{)} \quad \rho[x_1 \mapsto v_3] \vdash e \Downarrow v_4}{\rho \vdash \textbf{let}\ x_1\ \textbf{=}\ x_2\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \Downarrow v_4}\ (\textsc{let-var})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{\mathit{arg}}) \quad v_2 = \texttt{applyPrim(}\oplus, \vec{v}_1\texttt{)} \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \oplus\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e \Downarrow v_3}\ (\textsc{let-prim})$$

$$\frac{n_2 \text{ is input from port } n_1 \quad \rho[x \mapsto n_2] \vdash e \Downarrow v}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \textbf{getint}\ n_1\ \textbf{in}\ e \Downarrow v}\ (\textsc{getint})$$

$$\frac{n_2 = \rho(\mathit{arg}) \quad \rho[x \mapsto n_2] \vdash e \Downarrow v}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \textbf{putint}\ n_1\ \mathit{arg}\ \textbf{in}\ e \Downarrow v}\ (\textsc{putint})$$

Figure 3.2: Big-Step Semantics for Zarf's functional ISA (PART 1, continued in Figure 3.3).

indicates an integer value *n* against which to match, and an expression to evaluate if the value was a match.

The final two lines describe what makes up an identifier (*id*), which can be a variable, a function or constructor name, or a primitive operation, and what makes up an argument (*arg*), which can be an integer or a variable.

$$\frac{(cn, \vec{v}_1) = \rho(arg) \quad (cn\ \vec{x}\ \Rightarrow\ e_1) \in \overrightarrow{br} \quad \rho[\vec{x} \mapsto \vec{v}_1] \vdash e_1 \Downarrow v_2}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\textsc{case-con})$$

$$\frac{n = \rho(arg) \quad (n\ \Rightarrow\ e_1) \in \overrightarrow{br} \quad \rho \vdash e_1 \Downarrow v}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v}\ (\textsc{case-lit})$$

$$\frac{(cn, \overrightarrow{v_1}) = \rho(arg) \quad (cn\ \vec{x}\ \Rightarrow\ e_1) \notin \overrightarrow{br} \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\textsc{case-else}1)$$

$$\frac{n = \rho(arg) \quad (n\ \Rightarrow\ e_1) \notin \overrightarrow{br} \quad \rho \vdash e_2 \Downarrow v_1}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_1}\ (\textsc{case-else}2)$$

Figure 3.3: Big-Step Semantics for Zarf's functional ISA (PART 2, continued from Figure 3.2).

3.3.2 Big-Step Semantic Rules

The big step semantics in Figures 3.2 and 3.3 is a ternary relation on an environment, which maps variables to values; a let, case, or result expression; and the value to which this expression evaluates. An induction rule has a name uniquely identifying it, a premise (the text above the line), and a conclusion (the text below the line). The text below the line is effectively what one would "match against" in deciding how to continue execution — by taking the expressions of the program, one at a time, and matching against the text in the induction rules, following the indicated operations, one can actually evaluate the program.

The beginning of Figure 3.2 contains some top-matter explanations for abbreviations used in the semantics: c indicates a constructor, which is a name and a series of values (fields); clo is a closure, which is a function and a series of values (arguments); v is a value, which can be a number, a constructor, or a closure; finally, ρ is the environment, which maps vairables to values. The end of Figure 3.3 describes some helper-functions that are used; they are taken as given for now, but will be explained later.

The first rule, PROGRAM, is the top-level evaluation rule. We match against it when we have

$$\mathtt{applyFn}((\lambda\vec{x}_1.e,\vec{v}_1)\mathtt{,}\vec{v}_2\mathtt{,}\rho) = \begin{cases} v & \text{if } |\vec{v}_2| = 0,\ |\vec{v}_1| = |\vec{x}_1|, \text{ and } \rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow v \\ (\lambda\vec{x}_1.e,\vec{v}_1) & \text{if } |\vec{v}_2| = 0 \text{ and } |\vec{v}_1| < |\vec{x}_1| \\ \mathtt{applyFn}((\lambda\vec{x}_1.e,\vec{v}_1 :+ \mathtt{hd}(\vec{v}_2))\mathtt{,}\mathtt{tl}(\vec{v}_2)\mathtt{,}\rho) & \text{if } |\vec{v}_2| > 0 \text{ and } |\vec{v}_1| < |\vec{x}_1| \\ \mathtt{applyFn}((\lambda\vec{x}_2.e',\vec{v}_3)\mathtt{,}\vec{v}_2\mathtt{,}\rho) & \text{if } |\vec{v}_2| > 0,\ |\vec{x}_1| = |\vec{v}_1|, \text{ and } \rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow (\lambda\vec{x}_2.e',\vec{v}_3) \end{cases}$$

$$\mathtt{applyCn}(cn\ \mathtt{,}\vec{v}\ \mathtt{)} = \begin{cases} (cn,\vec{v}) & \text{if } (\textbf{con}\ cn\ \vec{x}) \in \overrightarrow{decl} \text{ and } |\vec{v}| = |\vec{x}| \\ (\lambda\vec{x}.\textbf{let}\ c\ \textbf{=}\ cn\ \vec{x}\ \textbf{in result}\ c,\vec{v}) & \text{if } (\textbf{con}\ cn\ \vec{x}) \in \overrightarrow{decl} \text{ and } |\vec{v}| < |\vec{x}| \end{cases}$$

$$\rho(arg) = \begin{cases} n & \text{if } arg = n \\ v & \text{if } arg = x \text{ and } (x \mapsto v) \in \rho \end{cases}$$

$$\mathtt{applyPrim}(\oplus,\vec{v}_1\mathtt{)} = \begin{cases} v & \text{if } |\vec{v}_1| = \mathtt{arity}(\oplus) \text{ and } v = \mathtt{eval}(\oplus,\vec{v}_1) \\ (\lambda\vec{x}_1.\textbf{let}\ x_2\ \textbf{=}\ \oplus\ \vec{x}_1\ \textbf{in result}\ x_2,\vec{v}_1) & \text{if } |\vec{v}_1| < \mathtt{arity}(\oplus) \text{ and } |\vec{x}_1| = \mathtt{arity}(\oplus) \end{cases}$$

Figure 3.4: Helper functions for the Zarf big-step semantics (Figures 3.2 and 3.3).

a series of function declarations and a main function, which is an expression e. The rule says that, using our other induction rules, we can reduce e to a value v, and that is what the program reduces to.

RESULT is matched when we have a **result** expression to evaluate. We are returning *arg*; the premise says that v is equal to whatever *arg* maps to in the environment (ρ). The expression reduces to *nu* — so, in English, we look up whatever we're returning and simply yield the value it maps to.

LET-CON is used for **let** expressions applying on a constructor (*cn*). We use the notation $\rho[x \mapsto v]$ to mean that we return an updated copy of the environment with x mapped to v. The premises state that:

1. $\vec{v}_1 = \rho(\overrightarrow{arg})$ — the sequence of arguments is looked up in the environment, producing a new sequence $\overrightarrow{v_1}$;

2. $v_2 = \texttt{applyCn}(cn, \vec{v}_1)$ — the helper function `applyCn` is applied to the constructor and $\overrightarrow{v_1}$, producing v_2;[5]

3. $\rho[x \mapsto v_2] \vdash e \Downarrow v_3$ — we create a copied version of the environment, where the name x now maps to v_2: this new environment will allow us (via the induction rules) to reduce the remaining expression e to a value v_3.

The instruction, including the remaining expression, reduces to v_3. In effect, we have built a value for the constructor with the indicated arguments, bound it to a name, and this will allow us to proceed with executing the remaining program.

LET-FUN is used for **let** expressions applying on a function — specifically, a function that is not a constructor (i.e., has a body e_2) and has a program defined declaration in $\overrightarrow{decl}$. The

[5] Applying a helper function which accepts one argument to a list of arguments is shorthand for mapping that helper function over the list.

premises tell us how the instruction, along with its tail expression, can be reduced to a value v_3. The premises state that:

1. $fn \notin \{\textbf{getint}, \textbf{putint}\}$ — the function fn (the one we're applying on) is specifically not an I/O function, i.e., not **getint** or **putint** (these are handled with separate induction rules);

2. $\textbf{fun}\, fn\ \vec{x}_2\ = e_2 \in \overrightarrow{decl}$ — the function fn is in our set of declarations; it takes a series of arguments $\overrightarrow{x_2}$ and has the body expression e_2;

3. $\vec{v}_1 = \rho(\overrightarrow{arg})$ — the sequence of arguments is looked up in the environment, producing a new sequence $\overrightarrow{v_1}$;

4. $v_2 = \texttt{applyFn(}(\lambda\vec{x}_2.e_2, [])\texttt{,}\vec{v}_1\texttt{,}\rho\texttt{)}$ — the helper function `applyFn` is applied to fn, the value list $\overrightarrow{v_1}$, and the environment ρ,[6] producing v_2;

5. $\rho[x_1 \mapsto v_2] \vdash e_1 \Downarrow v_3$ — we create a copied version of the environment, where the name x_1 now maps to v_2: this new environment will allow us (via the induction rules) to reduce the remaining expression e_1 to a value v_3.

As with the prior rule, the instruction and its tail expression reduce to v_3. We mapped the indicated arguments to their values, fed them to the indicated function (via the helper function `applyFn`), and assigned the resulting value to indicated name; this allows us to proceed with executing the remaining program.

LET-VAR is used for **let** expressions applying on a variable — i.e., an existing closure in the system. x_1 refers to the name to which to bind the result, while x_2 is the name of the closure on which to apply. The premises state that:

1. $v_1 = \rho(x_2)$ — the name of the closure x_2 is looked up in the environment, producing the value v_1;

[6] The exact syntax of the application, containing a lambda expression and empty list, can seem confusing — but is set up to make the definition of `applyFn` as simple as possible. In short, this helper function will feed arguments to the indicated function one at a time, handling the more complex cases of under- and over-saturation.

2. $\vec{v}_2 = \rho(\overrightarrow{arg})$ — the sequence of arguments is looked up in the environment, producing the sequence of values $\overrightarrow{v_2}$;

3. $v_3 = \texttt{applyFn(}v_1\texttt{,}\vec{v}_2\texttt{,}\rho\texttt{)}$ — the helper function `applyFn` is applied to the closure v_1, the list of argument values $\overrightarrow{v_2}$, and the environment ρ, producing the value v_3;

4. $\rho[x_1 \mapsto v_3] \vdash e \Downarrow v_4$ — we create a copied version of the environment, where the name x_1 maps to the value v_3: this new environment will allow us (via the induction rules) to reduce the remaining expression e to a value v_4.

As before, the instruction and its tail expression reduce to a value v_4. We proceeded the same as with the prior rule, but had the additional step of looking up the closure that the name x_2 referred to.

LET-PRIM is used for **let** expressions applying on a primitive (built-in) operation. The special operator $\oplus$ (an identifier in the abstract syntax) represents the primitive operation. The premises state that:

1. $\vec{v}_1 = \rho(\overrightarrow{arg})$ — the arguments $\overrightarrow{arg}$ passed to the primitive operation are looked up in the environment, producing the sequence of values $\overrightarrow{v_1}$;

2. $v_2 = \texttt{applyPrim(}\oplus, \vec{v}_1\texttt{)}$ — the helper function `applyPrim` is applied to the operator $\oplus$ (add, subtract, less than, etc.) and the value sequence $\overrightarrow{v_1}$, producing the new value v_2;

3. $\rho[x \mapsto v_2] \vdash e \Downarrow v_3$ — we create a copied version of the environment, where the name x maps to the value v_2: this new environment will allow us (via the induction rules) to reduce the remaining expression e to a value v_3.

This rule proceeded similarly to the other `LET` rules, but handles the special case of using a built-in function, which cannot be found in $\overrightarrow{decl}$, nor is a closure in ρ), but rather a function executed by the hardware itself with no program definition.

GETINT reads an integer from the specified input port, binds the result to the indicated variable name, and proceeds with execution. PUTINT performs the reverse: it looks up the supplied argument in the environment, writes the value to the specified port (a side effect not included in the semantic rule), and then binds that same value to the indicated variable name.

CASE-CON (in the beginning of Figure 3.3) handles **case** instructions that match against constructors. It has the conclusion $\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2$, where *arg* is the variable being cased on (which could be an integer, a local variable, or an argument), $\overrightarrow{br}$ is the sequence of branches supplied against which the architecture will try to match. Each branch is composed of a constructor name or integer head, which is what the cased value is compared against, along with an expression to execute if the match succeeds. The final piece of the **case** instruction is the **else** clause, which supplies an expression to execute in the case there is no matching branch (handled in another semantic rule). The premises of the rule state that:

1. $(cn, \vec{v}_1) = \rho(arg)$ — we look up the supplied argument, and it maps to a constructor name *cn* and a sequence of values $\overrightarrow{v_1}$, which are the values inside of the constructor object (its fields);

2. $(cn\ \vec{x}\ \Rightarrow\ e_1) \in \overrightarrow{br}$ — there is a branch in the program's supplied sequence of branches whose head has the same constructor name *cn*, it supplies a series of names $\overrightarrow{x}$ to which to bind the field values, and it has the expression e_1;

3. $\rho[\vec{x} \mapsto \vec{v}_1] \vdash e_1 \Downarrow v_2$ — we create a copied version of the environment, where the names given in $\overrightarrow{x}$ now map to the values from the dynamic object $\overrightarrow{v_1}$: this new environment will allow us to evaluate the expression e_1 from the matching branch to a value v_2.

The complexity in the rule lies in mapping the supplied field names in the program to the dynamic values of the matched constructor at runtime. Also, note that we specifically continued evaluation with e_1, the expression from the matching branch only.

CASE-LIT is also for **case** instructions, but for those that match against literal (integer) values. Note that, compared to the previous rule, there is no longer a constructor name cn, or field names or values $\overrightarrow{x}$ and $\overrightarrow{v_1}$. Intead ,we have the simple integer value n. The premises of the rule state that:

1. $n = \rho(arg)$ — we look up the supplied argument in the environment, and it maps to an integer value n;

2. $(n \;\Rightarrow\; e_1) \in \overrightarrow{br}$ — there is a branch in the program's supplied sequence of branches whose head has the same integer n, and it has the expression e_1;

3. $\rho \vdash e_1 \Downarrow v$ — using the environment ρ, we can reduce the expression e_1 from the matching branch to a value v;[7]

The final two rules, CASE-ELSE1 and CASE-ELSE2 correspond to CASE-CON and CASE-LIT (respectively) for the situation where there is not matching branch supplied in $\overrightarrow{br}$. For CASE-ELSE1, note the only differences with `CASE-CON` are $\in$ becoming $\notin$, no longer needing to bind any field names with values (in the else branch, there is no matching constructor, so there are no names to bind to), and evaluating e_2 instead of e_1. For CASE-ELSE2, the only differences with CASE-LIT are $\in$ becoming $\notin$ and and evaluating e_2 instead of e_1.

3.3.3 Big-Step Helper Functions

Here we detail the machinations of the four helper functions in the bottom half of Figure 3.3.

The first of these, `applyFn`, handles function application. It takes as arguments 1) a lambda

[7] Here, we do not need to make a copy of the environment because there is no new name to bind something to, as with **let** instructions binding a call to a name and **case** instructions that match on constructors binding field names to values. The result that was matched against, which was originally supplied in the variable *arg*, does not need a new binding, because *arg* will still be in scope inside of e_1.

(function) expression along with a sequence of values,[8] 2) a sequence of values to feed to the expression, and 3) an environment. `applyFn` is divided into four cases, detailed below ($|\vec{x}|$ means length of the list $\vec{x}$).

1. If $|\vec{v}_2| = 0, |\vec{v}_1| = |\vec{x}_1|$, and $\rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow v$ — we have no arguments to feed to the closure ($|\vec{v}_2| = 0$), the closure contains exactly the expected number of arguments ($|\vec{v}_1| = |\vec{x}_1|$), and mapping the names of the expected arguments $\vec{x}_1$ to the values in the closure $\vec{v}_1$ will allow us to reduce the function's expression e to a value v. In this case, we simply evaluate e and return v.

2. If $|\vec{v}_2| = 0$ and $|\vec{v}_1| < |\vec{x}_1|$ — we have no arguments to feed to the closure ($|\vec{v}_2| = 0$), and the number of arguments already in the closure is less than the expected number ($|\vec{v}_1| < |\vec{x}_1|$). We simply return the closure as-is, since we don't yet have enough arguments to evaluate it.

3. If $|\vec{v}_2| > 0$ and $|\vec{v}_1| < |\vec{x}_1|$ — we have a nonzero number of new arguments to feed to the closure ($|\vec{v}_2| > 0$) and the number of arguments already in the closure are less than the number needed for evaluation ($|\vec{v}_1| < |\vec{x}_1|$). We recursively use `applyFn` on a modified closure where we have copied the first argument from the supplied argument list $\vec{v}_2$ to the tail of the argument list inside the closure $\vec{v}_1$.[9]

4. If $|\vec{v}_2| > 0, |\vec{x}_1| = |\vec{v}_1|$, and $\rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow (\lambda\vec{x}_2.e', \vec{v}_3)$ — we have a nonzero number of arguments to feed to the closure ($|\vec{v}_2| > 0$), but the closure has exactly the number of arguments needed for evaluation ($|\vec{x}_1| = |\vec{v}_1|$). By mapping the arguments $\vec{x}_1$ to the values $\vec{v}_1$ in the environment, we can reduce the expression e in the closure to a new closure

[8] This pairing of a function with a sequence of values is, effectively, a "closure." In programming languages, different definitions can be used where a closure contains an entire environment; throughout this thesis, "closure" is used to specifically mean "function plus some number of arguments." (That number can be 0.)

[9] This clause in Figure 3.3 uses operators that merit definition here. $\vec{x} :+ y$ appends y to the end of $\vec{x}$, creating a new list. `hd` and `tl` have the usual definitions of head and tail, taking the first entry from a list or everything after the first entry (respectively).

$(\lambda\vec{x}_2.e', \vec{v}_3)$. We recursively use `applyFn` on this new closure, passing in the arguments $\vec{v}_2$ we were handed.

Because arguments are only ever copied one at a time, we never have to deal with the over-saturated case where there are too many arguments in the closure. Despite this, it may feel like the set of four cases in `applyFn` are incomplete; they are, but this incompleteness is by design to disallow invalid behaviors. In the fourth case, it may feel like we made an assumption that the first closure, when evaluated, yields another closure to consume the argument list we have. But remember, that was a *premise* of the `applyFn` case: if it didn't evaluate to another closure, we couldn't apply that case. This highlights that only *semantically well-formed* programs have definitions here. If a program is erroneous — for example, by trying to pass extra arguments to an expression that does not return a closure to consume them — then there is no valid step in these semantics for it to follow, and execution would halt.[10]

`applyCn` handles constructor application. Because there are fewer cases to deal with (constructors cannot be over-saturated, for example), there are fewer branches in the helper function. It takes as arguments the name of the constructor cn and a sequence of values $\vec{v}$. It has two cases, detailed below.

1. If $(\textbf{con}\ cn\ \vec{x}) \in \overrightarrow{decl}$ and $|\vec{v}| = |\vec{x}|$ — the constructor cn is in our program's list of declarations $\vec{decl}$, with the sequence of fields $\vec{x}$, and the sequence of values being fed to the constructor are of the same length as the field sequence (it's receiving the expected number of arguments, $|\vec{v}| = |\vec{x}|$). In this case, we simply build the constructor by returning the constructor name cn with the given arguments $\vec{v}$ for the fields: $(cn, \vec{v})$.

2. if $(\textbf{con}\ cn\ \vec{x}) \in \overrightarrow{decl}$ and $|\vec{v}| < |\vec{x}|$ — the constructor cn is in our program's list of declarations $\vec{decl}$, with the sequence of fields $\vec{x}$, and there are too few arguments being fed to the

[10] When "executing" the semantics, execution would halt in this case, but in the actual machine, to be more error-tolerant, a special "error closure" is made and returned as the result of the offending operation.

constructor ($|\vec{v}| < |\vec{x}|$). In this case, we build a new closure with the arguments $\vec{v}$ already in it. When additional arguments are applied to the closure, `LET-VAR` will be invoked, which will call `applyFn`, which will handle copying the additional arguments in to the argument list. If this fully saturates the closure, `applyFn` will attempt to evaluate it — this would invoke `LET-CON`, which would call `applyCn`, which (because it now has the correct number of arguments) can simply take the first case and build the constructor.

Note that both cases require that the constructor is defined in the program.

It may seem strange that, in the second case of `applyCn`, we appeared to dynamically create new syntax by introducing new instructions that weren't originally in the program. However, we can treat each constructor as having a predefined body that evaluates to that constructor application (the new syntax), meaning that the transformation occurs only once — and thus we can do it statically.

The helper function for ρ is relatively simple. The first case states that if the argument passed to ρ is already evaluated (i.e., is an integer), simply return that number. The second case states that if the given argument is a name that maps to some value v, return v.

The final helper function is `applyPrim`. It is used to apply arguments to a primitive operation. It requires a different helper function from `applyFn` because primitive operations have no program-defined expression — they are executed by the hardware. Here, we use the special function `eval` to stand in for hardware evaluation; it returns the value of applying a primitive operation to arguments. `applyPrim` takes two arguments: the primitive operation $\oplus$, and a sequence of values $\vec{v}_1$ to feed to the operation (arguments). The first case states that if the right number of arguments are applied ($|\vec{v}_1| = arity(\oplus)$), and the operator applied to the arguments yields the value v ($v = eval(\oplus, \vec{v}_1)$), we simply return v. The second case is for when too few arguments are applied ($|\vec{v}_1| < arity(\oplus)$). Here, we create a new closure that expects the correct number of arguments and wrap up the given arguments in the closure. When it is fed the remaining arguments, it can be evaluated with the other semantics rules and helper functions. Here, as with `applyCn`, it appears that we are dynamically creating new syntax, but in fact it is merely a predefined body that can be added statically for each primitive operation.

3.4 Small Step Semantics

Big-step semantics can be useful by being high-level and concise. However, for some applications, a small-step semantics is required. Most notably, later (in Chapter 7) we will add a type system to the ISA. For the type system to be useful, we must prove that it is sound. This reduces to two inductive proofs: one of progress, meaning that at every step there is always an applicable semantic rule to take one more step, and preservation, meaning that in taking every step we do not violate type safety. Taken together, these two principles mean that a program will stay type safe as it runs and will always be able to keep running, until it halts or loops indefinitely. To do these proofs, however, we need to reason about each *step* of computation, and for that we require a small-step operational semantics, which is given in Figures 3.5 and 3.6.

The small-step semantics is longer and more detailed than the big-step semantics; for example, it exposes a continuation stack to manage function applications and excess arguments. The extra state and steps required for the small-step version are what makes it more useful in certain contexts. The inference rules more closely resemble how the hardware runtime actually executes instructions.[11] However, some details are still abstracted away — for example, rather than a stack and heap as in a lower level, we just have a unified environment Γ.

As with the big-step semantics, the following explanations are in place for readers that would like each inference rule explained in words. If one has no trouble reading Figures 3.5

[11] The implementation at the hardware level is still vastly more complex, because it encounters all the normal pitfalls of building a hardware system. However, abstracting away many of the details, you are left with an interface that looks just like the small-step semantics.

$$f \in Function = Name \quad \Gamma \in Environment = Variable \rightarrow Value$$

$$v \in Value = Z \uplus Constructor \uplus Closure$$

$$con \in Constructor = \text{constructor}(f, \overrightarrow{fields}) \quad clo \in Closure = \text{closure}(f, \overrightarrow{arg})$$

$$K \in ContinuationStack = \overrightarrow{k} \quad k \in K = \text{letK}(\Gamma, exp, name) \mid \text{argsK}(\overrightarrow{arg})$$

$$\Sigma \in ItableMap = f \rightarrow \overrightarrow{name}, exp$$

$$\frac{\Sigma[f] = \overrightarrow{var}, e' \quad |\overrightarrow{arg}| = |\overrightarrow{var}| \quad \overrightarrow{arg}' = \Gamma[\overrightarrow{arg}] \quad \Gamma' = [var_i \mapsto arg'_i \mid i \in 0..n-1]}{K, \Gamma, \textbf{let } x = f\ \overrightarrow{arg} \textbf{ in } e \rightarrow \text{letK}(\Gamma, e, x) :: K, \Gamma', e'}\ \textsc{let-sat}$$

$$\frac{\Sigma[f] = \overrightarrow{var}, e' \quad |\overrightarrow{arg}| < |\overrightarrow{var}| \quad \overrightarrow{arg}' = \Gamma[\overrightarrow{arg}] \quad \Gamma' = \Gamma[x \mapsto \text{closure}(f, \overrightarrow{arg}')]}{K, \Gamma, \textbf{let } x = f\ \overrightarrow{arg} \textbf{ in } e \rightarrow K, \Gamma', e}\ \textsc{let-undersat}$$

$$\frac{\begin{array}{c}\Sigma[f] = \overrightarrow{var}, e' \quad |\overrightarrow{arg}| > |\overrightarrow{var}| \quad n = |\overrightarrow{var}| \quad m = |\overrightarrow{arg}| \quad \overrightarrow{arg}' = \Gamma[\overrightarrow{arg}] \\ k = \text{argsK}(arg'_n..arg'_{m-1}) \quad \Gamma' = [var_i \mapsto arg_i \mid i \in 0..n-1]\end{array}}{K, \Gamma, \textbf{let } x = f\ \overrightarrow{arg} \textbf{ in } e \rightarrow k :: \text{letK}(\Gamma, e, x) :: K, \Gamma', e'}\ \textsc{let-oversat}$$

$$\frac{\begin{array}{c}\Gamma[id] = \text{closure}(f, \overrightarrow{arg_1}) \quad \Sigma[f] = \overrightarrow{var}, e' \quad \overrightarrow{arg} = \overrightarrow{arg_1}{+}{+}\overrightarrow{arg_2} \quad |\overrightarrow{arg}| = |\overrightarrow{var}| \\ \overrightarrow{arg}' = \Gamma[\overrightarrow{arg}] \quad \Gamma' = [var_i \mapsto arg'_i \mid i \in 0..n-1]\end{array}}{K, \Gamma, \textbf{let } x = id\ \overrightarrow{arg_2} \textbf{ in } e \rightarrow \text{letK}(\Gamma, e, x) :: K, \Gamma', e'}\ \textsc{letclo-sat}$$

$$\frac{\begin{array}{c}\Gamma[id] = \text{closure}(f, \overrightarrow{arg_1}) \quad \Sigma[f] = \overrightarrow{var}, e' \quad \overrightarrow{arg} = \overrightarrow{arg_1}{+}{+}\overrightarrow{arg_2} \\ |\overrightarrow{arg}| < |\overrightarrow{var}| \quad \overrightarrow{arg}' = \Gamma[\overrightarrow{arg}] \quad \Gamma' = \Gamma[x \mapsto \text{closure}(f, \overrightarrow{arg}')]\end{array}}{K, \Gamma, \textbf{let } x = id\ \overrightarrow{arg_2} \textbf{ in } e \rightarrow K, \Gamma', e}\ \textsc{letclo-undersat}$$

$$\frac{\begin{array}{c}\Gamma[id] = \text{closure}(f, \overrightarrow{arg_1}) \quad \Sigma[f] = \overrightarrow{var}, e' \quad \overrightarrow{arg} = \overrightarrow{arg_1}{+}{+}\overrightarrow{arg_2} \\ |\overrightarrow{arg}| > |\overrightarrow{var}| \quad n = |\overrightarrow{var}| \quad m = |\overrightarrow{arg}| \quad \overrightarrow{arg}' = \Gamma[\overrightarrow{arg}] \quad k = \text{argsK}(arg'_n..arg'_{m-1}) \\ \Gamma' = [var_i \mapsto arg_i \mid i \in 0..n-1]\end{array}}{K, \Gamma, \textbf{let } x = id\ \overrightarrow{arg_2} \textbf{ in } e \rightarrow k :: \text{letK}(\Gamma, e, x) :: K, \Gamma', e'}\ \textsc{letclo-oversat}$$

Figure 3.5: Zarf ISA small step semantics (with strict evaluation) PART 1, continued in Figure 3.6.

$$\frac{\Sigma[op] = \cdot, \oplus \quad |\overrightarrow{arg}| = \texttt{primArity}(\oplus) \quad \Gamma' = \Gamma[x \mapsto \texttt{primOp}(\oplus, \overrightarrow{arg})]}{K, \Gamma, \textbf{let}\ x\ = op\ \overrightarrow{arg}\ \textbf{in}\ e \rightarrow K, \Gamma', e}\ \textsc{let-prim}$$

$$\frac{\Sigma[f] = \overrightarrow{field}, \cdot \quad |\overrightarrow{arg}| = |\overrightarrow{field}| \quad \Gamma' = \Gamma[x \mapsto \text{constructor}(f, \overrightarrow{arg})]}{K, \Gamma, \textbf{let}\ x\ = f\ \overrightarrow{arg}\ \textbf{in}\ e \rightarrow K, \Gamma', e}\ \textsc{let-con}$$

$$\frac{\begin{array}{c}\Gamma[id] = \text{closure}(op, \overrightarrow{arg_1}) \quad \Sigma[op] = \cdot, \oplus \quad \overrightarrow{arg} = \overrightarrow{arg_1}{+\!+}\overrightarrow{arg_2} \\ |\overrightarrow{arg}| = \texttt{primArity}(\oplus) \quad \Gamma' = \Gamma[x \mapsto \texttt{primOp}(\oplus, \overrightarrow{arg})]\end{array}}{K, \Gamma, \textbf{let}\ x\ = id\ \overrightarrow{arg_2}\ \textbf{in}\ e \rightarrow K, \Gamma', e}\ \textsc{letclo-prim}$$

$$\frac{\begin{array}{c}\Gamma[id] = \text{closure}(f, \overrightarrow{arg_1}) \quad \Sigma[f] = \overrightarrow{field}, \cdot \quad \overrightarrow{arg} = \overrightarrow{arg_1}{+\!+}\overrightarrow{arg_2} \\ |\overrightarrow{arg}| = |\overrightarrow{field}| \quad \Gamma' = \Gamma[x \mapsto \text{constructor}(f, \overrightarrow{arg})]\end{array}}{K, \Gamma, \textbf{let}\ x\ = id\ \overrightarrow{arg_2}\ \textbf{in}\ e \rightarrow K, \Gamma', e}\ \textsc{letclo-con}$$

$$\frac{\Gamma' = \Gamma[x \mapsto n]}{K, \Gamma, \textbf{let}\ x\ = \ n\ \textbf{in}\ e \rightarrow K, \Gamma', e}\ \textsc{let-primint}$$

$$\frac{\Gamma[x] = v \quad v \neq \text{closure}(\cdot, \cdot) \quad K = \text{letK}(\Gamma', e', y) :: K' \quad \Gamma'' = \Gamma'[y \mapsto v]}{K, \Gamma, \textbf{result}\ x\ \rightarrow K', \Gamma'', e'}\ \textsc{result-var}$$

$$\frac{\Gamma[x] = \text{closure}(f, \overrightarrow{arg}) \quad K = \text{argsK}(\overrightarrow{var}) :: K'}{K, \Gamma, \textbf{result}\ x\ \rightarrow K', [], \texttt{let}\ z = f\ \overrightarrow{arg}{+\!+}\overrightarrow{var}\ \textbf{in}\ \texttt{result}\ z}\ \textsc{result-clo}$$

$$\frac{\begin{array}{c}\Gamma[x] = \text{constructor}(\text{f}, \overrightarrow{arg}) \quad f\ \overrightarrow{name} \Rightarrow e \in \overrightarrow{br} \\ \Gamma' = \Gamma[name_i \mapsto arg_i \mid i \in 0..|\overrightarrow{arg}| - 1]\end{array}}{K, \Gamma, \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br} \rightarrow K, \Gamma', e}\ \textsc{case-pat}$$

$$\frac{\Gamma[x] = n \quad n \Rightarrow e \in \overrightarrow{br}}{K, \Gamma, \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e' \rightarrow K, \Gamma, e}\ \textsc{case-lit} \qquad \frac{\Gamma[x] = n \quad n \Rightarrow \cdot \notin \overrightarrow{br}}{K, \Gamma, \textbf{case}\ x\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e' \rightarrow K, \Gamma, e'}\ \textsc{case-lit-else}$$

$$\frac{}{\cdot, \cdot, \textbf{fun}\ \text{main} = e \rightarrow [], [], e}\ \textsc{program}$$

Figure 3.6: Zarf ISA small step semantics (with strict evaluation) PART 2, continued from Figure 3.5.

and 3.6, or does not need to understand them fully, then one should feel free to skip to the next section.

The abstract syntax is the same as with the big-step semantics, so we do not repeat it here (refer to Figure 3.1 if necessary). The relevant abstract domains are listed at the top of Figure 3.5. A function f is just a unique name. An environment Γ maps variables to values. A value can be an integer, a constructor, or a closure. A constructor is an object containing a function (name) and a sequence of fields (values). A closure is a similar object, also containing a function and a sequence of arguments (values); the reason we have both objects is that they serve different purposes and we want to distinguish between them at runtime. The continuation stack K is a sequence of continuations; each continuation k can either be a let continuation (letK), storing an environment, an expression, and a name, or an argument continuation (argsK), storing a sequence of arguments. Let continuations are used to save state when a function call takes place as part of a `let` instruction; it saves the previous environment that will be resumed, the expression to resume, and the name of the variable to which the result of the function call should be bound. An argument continuation is used when a function is over-applied; it just stores the excess arguments, which will be required later when an undersaturated closure is returned. Finally, we have our itable map Σ, which is initialized with static information about all the functions in the program. It maps a function (name) to a series of names (the arguments) and an expression for the body of the function.

Each of the induction rules is a relation from a continuation stack K, an environment Γ, and an expression e to a new continuation stack, environment, and expression. I.e., each rule performs $K, \Gamma, e \to K', \Gamma', e'$. How the stack and environment change and how the next expression is chosen is the heart of how each rule is executed.

There are a total of ten rules for `let` applications. There are five situations to consider: a saturated application, undersaturated, oversaturated, primitive application, and closure application; then, for each case, we must consider a “static” application on a function f, and a "dynamic" application on a closure object. The difference between the two is that the static version applies on a program-defined function, and the arguments passed in are the only arguments to consider, while the dynamic version applies on a closure object (which has a pointer to a program-defined function), and we must consider both the arguments already in the closure and the arguments being passed in in the instruction.

Having already gone through the big-step semantics in detail, this explanation will be less explicit and more abbreviated, while still trying to fully explain each inference rule.

The first rule, LET-SAT, is for static `let` applications that have exactly the right number of arguments. First, we look up the function f we are applying on in the itable map Σ; this gives us a sequences are variables $\overrightarrow{var}$ and an expression e'. We require that the number of arguments given is exactly the number expected: $|\overrightarrow{arg}| = |\overrightarrow{var}|$; this is the condition for saturated application. We look up the indicated arguments in our environment with $\overrightarrow{arg}' = \Gamma[\overrightarrow{arg}]$. This is actually a shorthand: firstly, we use a vector notation to mean looking up each argument individually and building a new vector. Secondly, if an argument is an integer, that value is returned directly from Γ rather than attempting to look it up (only names map to values in Γ). This behavior is true for Γ everywhere it is used in these semantics. We create a new environment where each of the variable names from the function's declaration are mapped to the value in the new argument sequence. Our new continuation stack has a letK pushed on it to save state, we use the newly-created environment Γ', and we move on to execute e' (the expression making up the body of the applied function).

LET-UNDERSAT handles static `let` applications where too few arguments are passed in. The first three premises are as expected: we look up f in the itable map, require that too few arguments are present, and map the argument sequence to a value sequence using the environment. The only new premise is the creation of a closure containing f and the new argument sequence, which we bind to the name x in our new environment. Since there were too few arguments passed in, we cannot jump and begin executing this call; instead, we just make the closure and move on with the next expression *e*.

LET-OVERSAT handles static `let` applications where too many arguments are passed in. For this to be well-formed, the function must return an undersaturated closure that expects additional arguments, which will consume the extra arguments not initially used. So the key principle here is that, in addition to performing a normal `let` application, saving state, and jumping elsewhere, we must also save an argument continuation with the excess args so that they're waiting for us when we get back. We declare a variable n to be the number of arguments expected by the function, and a variable m to be the number that were passed in. We look up all the arguments in Γ, as with the other `let` rules, but then we split in how we deal with the new sequence. The first n arguments are put into a new environment, where the expected argument name maps to each value (this is the same mapping we did in LET-SAT). Arguments n through $m-1$ are saved in an argument continuation, which is put on the continuation stack *after* the let continuation. This ensures the arguments are seen before the final result of the `let` is required and execution resumes.

Next we examine LET-PRIM, which handles static `let` application on a primitive function where the expected number of arguments are given. We look up the op in the itable map, which instead of a body, yields a primitive operation $\oplus$. This operation is fed to our helper functions to facilitate execution of a primitive function. The helper function `primArity` takes the operation $\oplus$ and returns the number of expected arguments (which is two for most primitive functions, but one for others). We build a new environment where x maps to the result of another helper function `primOp`; this function takes the operation and the arguments and actually executes the primitive function according to which operation it is. We do not define a formal semantics for the primitive operations, because most have their standard mathematical definitions. The only two that may require further elaboration are the I/O operataions, `getint` and `putint`. `getint` takes one argument (a port number) and, when executed, reads a primitive integer from the requested port number and returns it. `putint` takes a port number and a primitive integer and, when executed, outputs the integer on the requested port.

Last in line for our static `let` inference rules is LET-CON, which handles building a constructor when the constructor function is given exactly the right number of arguments. Its definition is rather simple: we look up f to find the number of expected fields, make sure we have that number, then build a constructor in the environment (mapped from x) that holds the indicated constructor function f and the passed arguments $\overrightarrow{arg}$.

In the case where one applies on a primitive function or a constructor but gives too many or too few arguments, the correct inference rule to apply would be LET-OVERSAT and LET-UNDERSAT, respectively.

The dynamic, closure application version of each of the previous rules is very similar. Each changes in the same way, so we cover all five at once. Since *id* now refers to a closure, not a function, we must look the closure in Γ. This gives us our f and a sequence of arguments. We look up f in Σ, as with the standard rules, and concatenate the arguments in the closure with the arguments in the instruction. The remaining premises are the same for each rule; the closure merely provides another degree of indirection and some additional arguments that must be considered. This covers the rules LETCLO-SAT, LETCLO-UNDERSAT, LETCLO-OVERSAT, LETCLO-PRIM, and LETCLO-CON.

Next we look at the two possible inference rules for `result`. A `result` instruction must return a value of some kind. One rule (RESULT-VAR) handles the case where the value is an integer or constructor (i.e., a value in weak head normal form), while the other (RESULT-CLO) handles the case where the value is a closure.

The main behavior of RESULT-VAR is popping a letK continuation off the continuation stack to resume execution at the previously saved point. The first two premises indicate that x is looked up in the environment Γ to produce a value v, and v is not a closure. The third premise states that the top of the continuation stack must be a letK continuation. The final premise creates a new environment out of the saved one where we map the variable from the letK to the value being returned. This is the final step to finish execution of the `let` instruction that caused the function call, as now the result of that call is bound to the variable indicated in the `let` instruction (refer to the LET-SAT rule to see how the letK continuation was first created). Execution continues with the saved expression, which was the body that followed the `let` instruction.

RESULT-CLO has fewer premises, but has a more complicated underlying behavior. We look up x in the environment Γ and see that it must be a closure. Then, we require that the continuation on top of the continuation stack is an argsK continuation. When a closure is returned, it represents handing back an unfinished computation that is the result of an undersaturated function call. It must, to be well-formed, correspond to an oversaturated call taking place lower on the stack. Those excess arguments from the oversaturated call are saved in the argsK continuation, and then later consumed when the closure is returned. We combine the arguments in the closure with the arguments saved on the stack and process the new sequence as the arguments to the closure's function. However, we have a complicating factor: the new argument sequence could be too short, too long, or exactly right. To avoid needing three large, separate inference rules to handle these cases, which would duplicate behaviors described elsewhere in the inference rules, as a shorthand we simply dynamically create a new `let` instruction and use that as the next expression. To process the `let` instruction, the appropriate inference rule will be selected from LET-SAT, LET-UNDERSAT, and LET-OVERSAT. The alternative would be duplicating the behaviors of those three rules in three new rules to handle the RESULT-CLO case.

The `case` instruction has three possible inference rules. The first is CASE-PAT, where one cases on a constructor (and not an integer). The first premise captures this requirement, stating that x in the environment Γ maps to a constructor. Next, we require that there is a matching branch in the sequence of branches provided $\overrightarrow{br}$. (We can statically verify a `case` instruction is complete with regards to the constructors of a given type, guaranteeing there will always be a matching branch.) We will continue execution with the expression e from the matching branch, but first we must map the names given in the branch head to the fields of the dynamic constructor to which x refers. This binding gives a handle on the dynamic constructor object's fields and lets one "unfold" data structures to access their elements. The third and final premise does this binding, mapping each name given to the corresponding field from the dynamic constructor.

CASE-LIT handles the simpler situation where the scrutinee is an integer (and not a constructor). We look up x in Γ and find the matching branch in $\overrightarrow{br}$ where n is equal. Then, we continue by executing the expression e from the matching branch. There are no fields to map this time, nor anything new to introduce to K or Γ. CASE-LIT-ELSE is a symmetric rule that also handles integer scrutinees, but for the case where there is no matching branch. We see in the premise $n \Rightarrow \cdot \notin \overrightarrow{br}$, where we use a dot ($\cdot$) as a "don't care" stand-in, because the expression there is irrelevant for the rule. In this case, we simply execute the expression e from the else branch.

Note that CASE-PAT does not have an `else` branch; this is because it is assumed the programs are type-safe and checked for case completeness. If that is potentially not the situation, then only one additional rule would be needed. Just as CASE-LIT has a rule to handle the `else` case (CASE-LIT-ELSE), CASE-PAT would be extended in exactly the same way: the else expression e would be executed next, and we assume there is no matching branch ($f \cdot \Rightarrow \cdot \notin \overrightarrow{br}$).

Finally, the rule for a program (PROGRAM), is what starts execution and is quite simple: take an empty K and Γ and start executing with the body of `main`.

3.5 Binary Encoding

Figure 3.7 shows how binary programs are encoded on the Zarf platform. We go through the possible types of encoding here. Function Headers are used to declare functions and constructors (algebraic data types). It has a bit to determine if the declaration is a constructor or a function (*isCons*), 11 bits to determine the number of arguments expected (*arity*), 10 bits that

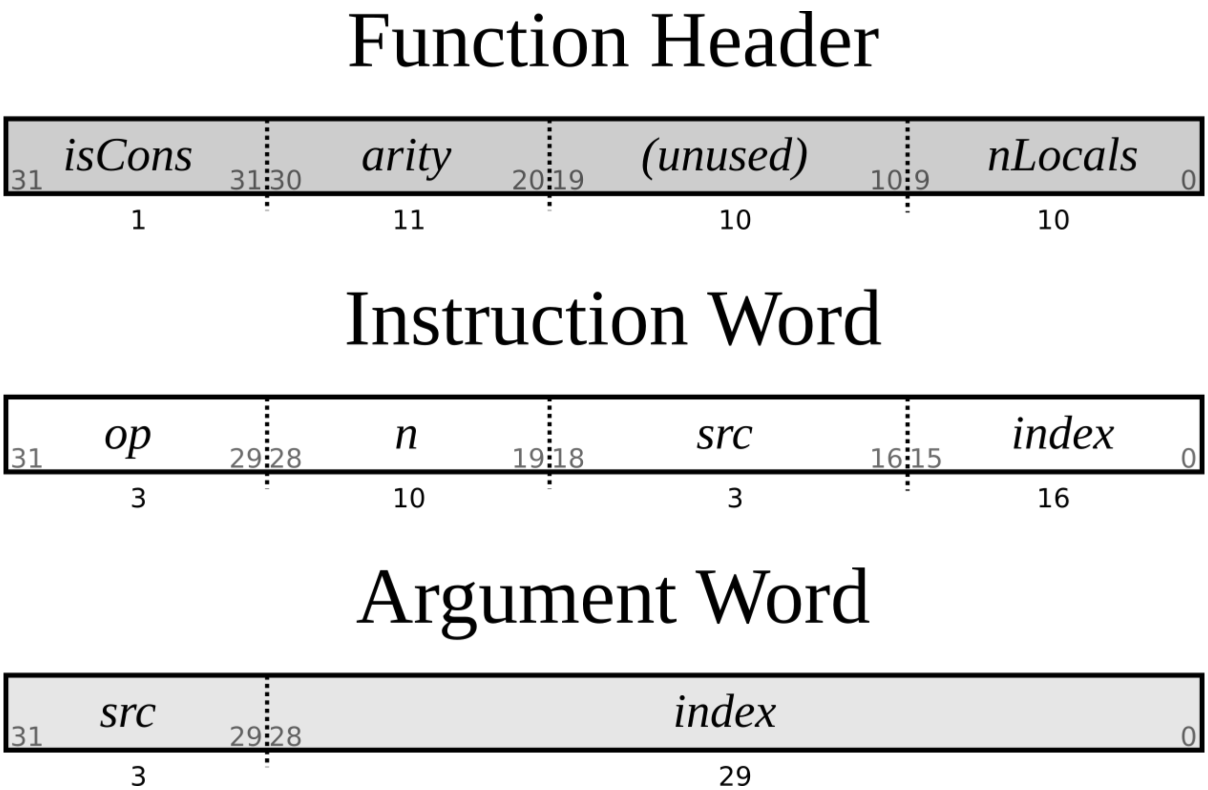


Figure 3.7: Binary encoding for the Zarf platform. Function headers give the signature for a function, followed by a body, or a constructor (a body-less function used to hold data). Instruction words are for the ISA's three instructions, while Argument Words are used to pass arguments into `let` instructions.

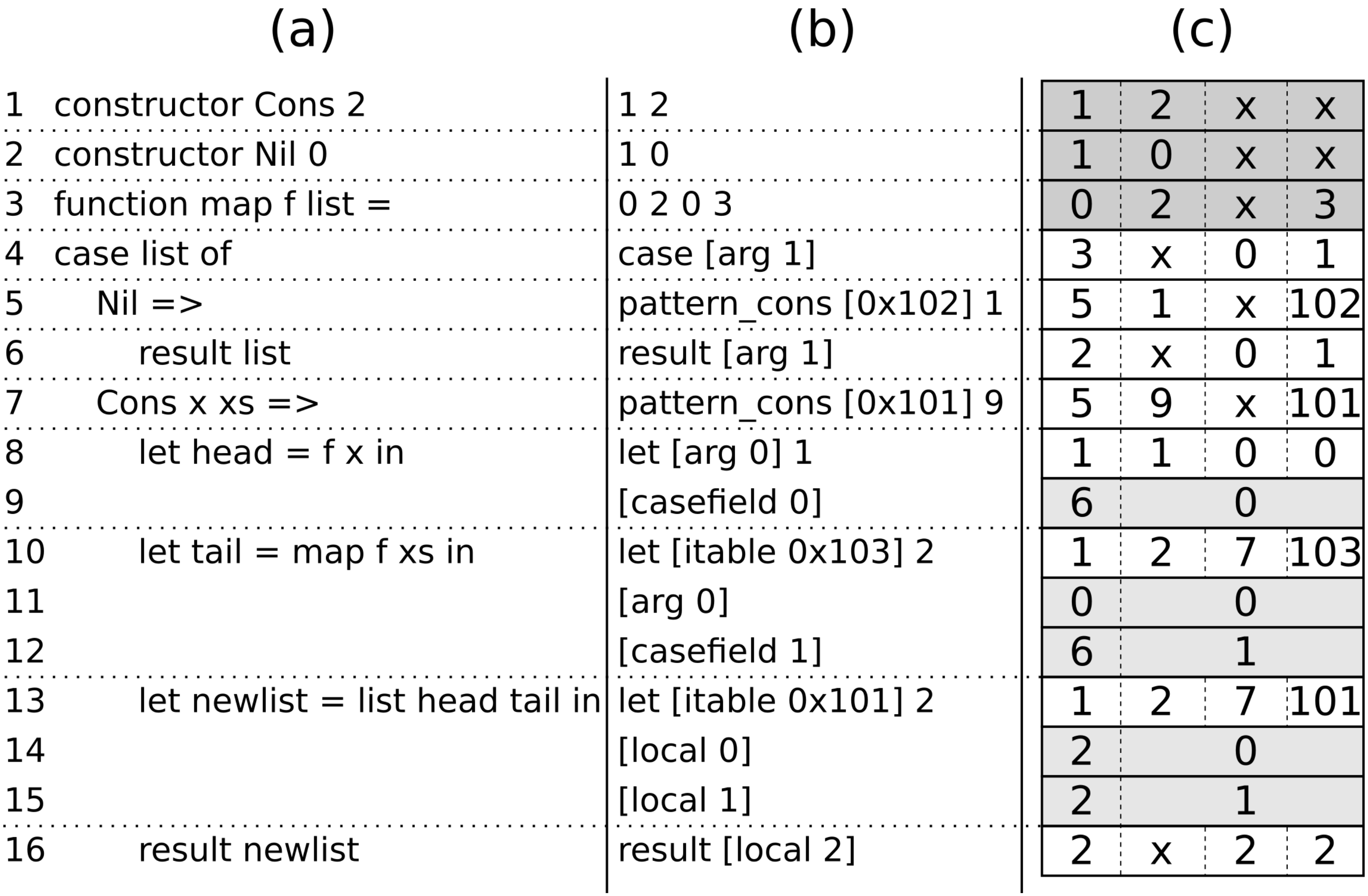
are unused,[12] and 10 bits for the number of local variables that will be allocated (*nLocals*).

A series of Instruction Words make up the body of a function. The first field of an instruction is the 3-bit opcode, which can be `let`, `case`, `result`, `pattern_cons`, or `pattern_lit`. The second field is *n* and gives the number of arguments (in the case of a `let` instruction), or the number of words to skip (in the case of a `pattern` instruction). The *src* field is 3 bits and, along with the *index* field (16 bits), can be used to identify a piece of data in scope when required in a an instruction. In `let` instructions, the pair is used to identify where to find the function to apply on; in `case` instructions, the pair identifies what to case on; in `result` instructions, they identify what to return. For `pattern` instructions, the *index* field is used to give the itable number to match against (for `pattern_cons`) or the literal integer to match against (for `pattern_lit`). The 16-bit limit in the *index* field of the instruction means that to match against a 32-bit value exactly, two separate matches must be performed on the high- and low-portions of the word. However, the normal way to match against a specific value is with comparators, where you `case` on the result of the comparison (a 0 or a 1). There, because argument words have a 29-bit *index*, the value limit is much greater for single-op comparisons.

Argument Words as used in a sequence after a `let` instruction. However many arguments the `let` instruction specifies in its *n* field, that many Argument Words must follow. Each Argument Word is made up of a *src* (3 bits) and *index* (29 bits). As with the *src*/*index* pair in Instruction Words, these uniquely identify a data source to feed as the next argument in the `let` instruction. Possible sources are arguments, locals, literals (integers), casefields, and itables. For each source, the *index* has a different meaning, as detailed in section 3.2.

Understanding the binary encoding is best elucidated by an example. Figure 3.5 shows the function `map`, and how it is compiled nearly line-by-line into a Zarf binary executable. Below,

[12] This field was originally used to specify the number of free variables that the function would take, which was an argument-passing optimization where some arguments did not need to be copied to re-apply a closure to new arguments. Later versions of the ISA eliminated free variables because the ISA was simpler without them, and their use-case was rather narrow.



	(a)	(b)	(c)			
1	constructor Cons 2	1 2	1	2	x	x
2	constructor Nil 0	1 0	1	0	x	x
3	function map f list =	0 2 0 3	0	2	x	3
4	case list of	case [arg 1]	3	x	0	1
5	Nil =>	pattern_cons [0x102] 1	5	1	x	102
6	result list	result [arg 1]	2	x	0	1
7	Cons x xs =>	pattern_cons [0x101] 9	5	9	x	101
8	let head = f x in	let [arg 0] 1	1	1	0	0
9		[casefield 0]	6	0		
10	let tail = map f xs in	let [itable 0x103] 2	1	2	7	103
11		[arg 0]	0	0		
12		[casefield 1]	6	1		
13	let newlist = list head tail in	let [itable 0x101] 2	1	2	7	101
14		[local 0]	2	0		
15		[local 1]	2	1		
16	result newlist	result [local 2]	2	x	2	2

Figure 3.8: How high-level assembly instructions are directly compiled into a valid Zarf binary for execution. This example shows the `map` function, along with the necessary list constructors (`Cons` and `Nil`), in (a) untyped high-level assembly, (b) low-level assembly, and (c) binary.

we detail the makeup of each column of the figure. The shading in **(c)** of the figure corresponds to the shading in Figure 3.7, showing which words are Function Headers, Instruction Words, and Argument Words.

Column (a) of Figure 3.5 contains the standard linked-list definitions for Cons (a 2-element struct with a head and a tail) and Nil (an empty list constructor) on lines 1-2. The function map has its declaration on line 3, which gives the name of the function (map) and its two arguments (a function f and list list). map will recursively build a new list, applying f to each entry in the given list. First, map branches on the structure of the given list, on line 4. Line 5 matches against the Nil constructor, making up the base case when the list is empty. If it matches, the function will return back an empty list (line 6).

There is a small quirk here, where to return an empty list, the function returns back the same list it was passed. We know the list is empty inside of this case branch, so the move is sound; we do it this way because of the way the Zarf hardware distinguishes between functions and closures, which is analogous to the way an object-oriented language distinguishing between classes and objects. A function cannot return a function, which is a pointer into a set of info tables; instead, it must return an "object," or closure: that function applied to zero or more arguments. It would be perfectly valid to apply Nil to no arguments and return that, but we can write the function more simply by returning the Nil object we were given.

On line 7, we handle the case where the list is not empty: we match against the constructor Cons and we bind the names x and xs to the head and tail of the dynamically matched list. Line 8 applies the function that was passed in to the head of the list, while line 9 makes a recursive call to apply map (with the same function f) to the tail of the matched list. A new list is built out of this head and tail in line 13, and this is returned as the result of the function in line 16. The else-branch of the case statement is not shown for brevity.

Column (b) is a lowered, "de-sugared" version of the same map program in column (a). In this lowering, we acknowledge constraints of the machine that can be made transparent to

Source	Meaning and *index* usage
`itable`	Static function pointer where *index* is the function ID
`arg`	Argument number *index* to this function context
`local`	Local number *index* in the current context
`literal`	Integer constant indicated by *index*
`caseField`	Field number *index* of the most recent case matched constructor

Table 3.1: Zarf binaries used a scoped system of references to access data and functions. A *source* and *index* pair uniquely identifies a function to apply arguments to or a piece of data to pass as an argument. Possible sources are listed on thel left, with the meaning of the source (and how *index* is interpreted) on the right.

higher-level assembly code. Function and constructor signatures turn into a series of flags and fields; variable names are erased and replaced with a scoped system of references; longer instructions are broken down into single-word operations that be encoded in fixed-width binary.

This scoped system of references is based on the *source* and *index* fields discussed earlier. We covered the topic in Section 3.2, but we summarize them here. The *source* indicates where the data comes from, while the *index* says which piece of data to grab. Possible *source*s are `itable` (function), `arg`, `local`, `literal` (integer), and `caseField`. The `itable` source is used only to apply arguments to static function pointers; it is invalid to pass a function pointer as an argument because of the function/closure difference already discussed. The `literal` source can only be used as an argument or as a function source with no arguments (which just returns the integer). The `arg`, `local`, and `caseField` sources can be used either to fetch a closure to apply arguments to, or to indicate data to be fed as an argument. The possible *source*s are listed in Table 3.1.

Many of the *source*s are context-sensitive, both in terms of function context, and where in the function the *source* is used. `literal` will return the same value anywhere in the program, because it represents an integer constant. Similarly, `itable` will have the same effect when evoked anywhere, because static function pointers (itables) do not change as a program executes. `arg` will have the same evaluation anywhere in a function body, because the arguments to a function cannot change once applied[13]. `local` accesses the *n*th local variable, where *index* indicates *n* and each allocated local dynamically takes the next available index (de Bruijn indices). So the first `let` instruction creates local 0, the next local 1, etc. etc. Because different program paths in a function can have different locals, a `local` source with the same *index* may return different data if placed in different parts of the function — however, it will always be consistent on a given path; e.g., it's impossible that accessing local *n* will return a different data value from the previous time local *n* was accessed. Only comparing across execution paths could we see different values emerge.

The final source is `caseField`, which a special deconstruction accessor. The `caseField` source type is valid only inside of a `case` statement that matched against a constructor. Each branch of that `case` statement will list a constructor to match against; on a match, it introduces that constructor into scope and binds the dynamic object's data to the static constructor's field numbers. Field numbers begin at 0, and one can easily verify statically that `caseField` numbers are never out of bounds by checking the enclosing branch's constructor to find the maximum number of fields. In the situation where a function has nested `case` instructions, `caseField` only applied to the most recent match. This means that, on occasion, the compiler has to introduce some new instructions to copy data (with a simple function like `id`) to prevent needed data values from going out of scope.

Turning our attention back to Figure 3.5, we can now understand the dataword encoding used in the instruction (most of the phrases in brackets in column `(b)`). Lines 1 and 2 declare our two constructors, while line 3 declares the function (which takes 2 arguments and has a maximum of 3 local variables). Line 4 is our `case` instruction, for which the scrutinee is the list passed in: `arg` 1. The first branch, line 5, matches against constructor 0x102[14] (the `Nil`

[13] In a lazy implementation, it's possible the arguments are not yet evaluated when we enter the function, but because of referential transparency and a lack of side effects, it's guaranteed that evaluating them at any time will yield the same results.

[14] Recall that function numbering starts with 0x100, and `main` is always 0x100. This makes the `List` constructor 0x101, and the `Nil` constructor 0x102.

constructor), indicating that there is 1 instruction to skip if the match fails. The branch returns `Nil` on a match (line 6). Line 7 is the next branch head, matching against constructor 0x101 (the `List` constructor), indicating to skip 9 instructions on a failed match. The remainder of the function is inside that branch head.

The `let` instruction on line 8 becomes 2 lower-level instructions: 1 `let` instruction (line 8) indicating that we are applying 1 argument to `arg` 0 (the function that was passed in), and 1 dataword (line 9) indicating to pass in `caseField` 0, which is the first field of the most recently matched constructor — the entry of the `List` we just matched. The next `let` instruction becomes three words. The first is the `let` instruction on line 10, which says to apply 2 arguments to `itable` 0x103 (our `map` function). Lines 11-12 have the two datawords to pass in as arguments: `arg` 0 and `caseField` 1. This makes the recursive call to `map`, passing in the same function and the tail of the list we matched against. The third let instruction also becomes 3 words when lowered. The `let` takes line 13, saying to apply 2 arguments to `itable` 0x101 (the `List` constructor). Lines 14-15 give the two arguments, which are `local` 0 and `local` 1, the head of the new list and the recursive call to map, which makes up the tail.

Finally, we return the new list (`local` 2) on line 16.

Column `(c)` contains an actual executable binary Zarf program. It is the same program as column `(b)`, where each line has been translated into exactly 1 word of the binary program (refer to Figure 3.7 for how the low-level instructions map to binary words). Figure 3.5, taken altogether, shows how high-level assembly code can, with minimal transformation be turned into actual executable code on the Zarf platform.

Chapter 4

Lazy Implementation

4.1 Introduction

The very first iteration of the Zarf platform used a lazy evaluation semantics. This means that expressions are not evaluated until their answer is required, rather than immediately when the expression is processed. This is a departure from the way things are normally done, but not so much as one might think — even "typical" languages like C have some laziness[1], and hardware mimics some properties of lazy execution with out-of-order and superscalar processing, as these things build a dependency graph of the computation and then evaluate it. Despite this, a piece of hardware with fully-lazy evaluation semantics is a novel and strange thing.

Laziness was chosen for a few reasons. One major component of the decision was that we drew a lot of early inspiration from Haskell, perhaps the most successful example of a pure-functional language. Haskell uses laziness to help enforce purity, though we didn't realize that was the extent of its usefulness at the time. The second reason was to enable easier hardware optimizations. As mentioned, an out-of-order execution engine effectively builds a running

[1] For example, the clauses in a compound Boolean expression are lazy. This allows one to write `(x != Null) && (x.field == 0)` without segfaulting when `x` is `Null`, because the second clause won't be evaluated until the first is checked.

graph of computation, building it on one end and evaluating it on the other. What if some of that was baked directly into the execution engine of the hardware; would that make optimizations easier? We would never arrive at a conclusion to that question, because it turned out building an out-of-order version of lazy Zarf had more complications than we realized initially (this is discussed somewhat in Section 8.1).

The final reason was that no one had ever built lazy-evaluation hardware before. That on its own seemed like a good enough idea to explore it.

The evaluation mechanisms employed by the hardware have to differ from "normal" processes in several places to enable the lazy execution. The first of these is that arguments cannot go on the stack, because applying arguments to a function is now divorced from the evaluation of that function. If we do not directly jump into a function after putting its arguments on the stack, the arguments will go out of scope and be lost. This means we must place function arguments inside of objects in the heap. We refer to those objects as "closures," a structure with a function pointer and zero or more arguments. In Haskell parlance, they might be called "thunks." In either case, they represent a computation that has not yet taken place, and have all the ingredients to make that computation happen: the pointer to the function and the arguments to be fed in.

We know that functions are not evaluated when they are applied, but this leads to the question: when do we actually evaluate anything? Evaluation takes place when the answers from a function call are required; the only place that an answer is definitely required is in a branch, because we need to know which branch to take. On Zarf, the only branching construct is the `case` instruction, so that is the only place where evaluation of closures is forced.

What this means in terms of evaluation hardware is that the thread of execution does not follow the normal pattern of changing at call/return points. Instead, it changes at `case`/`result` instructions. This makes `case` not only a branch head, but a conditional jump. If the thing being case'd on (the scrutinee) is unevaluated, a comparison cannot be made for branching, so the hardware must now evaluate the closure to arrive at a return value. If the scrutinee already happens to be evaluated, no jump needs to take place.

When context changes at the `case` instruction, a set of state is pushed onto a hardware stack, which we call the "continuation stack." The continuation stack stores whatever state is necessary to restore the state when evaluation of the scrutinee is done. Since evaluation can be arbitrarily long, involving an arbitrary amount of additional code and `case` evaluations, this state must be stored in a stack (rather than in registers or another simpler structure). Of course, the hardware cannot actually tolerate an arbitrary amount of additions to the continuation stack, because the continuation stack is finite. The state on the continuation stack just contains the PC, local context, local stack, pointer, and other things necessary to fully restore the execution state.

The restoration of state and continuation of the `case` instruction and its following branches will happen when a `result` instruction is encountered that is attempting to return an evaluated value. A value is "evaluated" if it is either a primitive integer or a constructor[2]. The only other thing that can be returned would be a closure. In the case that a closure is being returned, the machine must continue evaluation to attempt to reduce it to an evaluated value before returning to the previous `case` instruction. In this way, `result` instructions can also trigger evaluation via automatic tail calls.

Since the execution engine is *always* evaluating one closure or another (it begins with an evaluation of `main`), every `result` instruction will follow this rule and spawn a context switch to evaluate a closure that is unevaluated. As stated, this is different from a normal context switch, which places state on the continuation stack, because it just calls directly into the new function (a tail call). It is possible to return a closure that cannot be evaluated on its own because it has too few arguments inside it (undersaturation). This is still legal, because it's

[2] A constructor is just a special function that has no body; in effect, it is a type of closure that cannot be evaluated any further, its only purpose being to hold data.

possible the current context was given too many arguments, with the intent that the returned undersaturated closure will consume the excess arguments and be reduced to an evaluated value before execution resumes at the previous `case` instruction. This checking of saturation and shuffling of arguments is handled by the hardware evaluation engine.

Of course, this feature can be abused or used in err, returning an undersaturated closure when there are no arguments waiting to be consumed. This condition can be detected dynamically, forcing the hardware into an error state, which will propagate up the stack until handled. In Chapter 7, we show that it is possible to place binary types on a program that can ensure, at the binary level, that these sorts of conditions never occur, guaranteeing a program will be free from error states.

Another important difference in the implementation of the lazy Zarf processor is the inclusion of what we call the “Name Table.” This is a simple memory block that adds a degree of indirection to all heap addresses. Incoming addresses index into the memory and are translated into physical heap address locations. This indirection (or virtualization) means that the “addresses” used throughout the system are, in fact, “names,” or indices into this table. The benefit of this extra layer of abstraction is that an object can be relocated in physical memory without needing to change the corresponding names that point to it throughout the system; no matter how many references exist to this object, one update to the Name Table is enough to make them all point to the new version. This enhancement makes garbage collection easier, obfuscating the need for forwarding addresses.

However, the design decision to include a Name Table and indirect all heap addressing had its costs. Accessing memory — already a significant bottleneck in many computing applications — was delayed by an additional cycle for the cost of moving through the Name Table. Many parts of the evaluation state machine need to read pieces of memory; every one of those states needed to be expanded into multiple states to handle the added latency of a Name Table access. The result was that though garbage collection was simplified, the remainder of the design was made more complex and given more overhead.

In addition, the choice of a lazy abstraction created overheads that compounded with the inclusion of the Name Table. In the lazy world, because all objects need to be stored in the heap rather than on the stack, we encounter what is known as “thunk explosion” — every function application gives rise to a new heap object, and since function application is the only mechanism of the system, it rapidly populates your memory space. Along with the extra latency required to go through the Name Table to access memory, this created a new bottleneck and serious overhead for the initial system.

The final design quirk, which is not related to the choice of laziness, is a runtime one-bit tag placed on all values to distinguish between integers and addresses. These tags serve two purposes. The first is that is allows the machine to dynamically catch serious errors, like trying to use an integer as an address or add a number to a reference. The second is in `case` instructions, we can know if the scrutinee is either an integer or a reference based on its tag. Some software systems, such as Haskell, employ 1-bit tags on their values as an optimization; for this reason, Haskell has 31-bit integers. However, we are in a pure-hardware system, and have the freedom to make a 33-bit datapath and memory. This allows us to have full 32-bit integers and still keep our single-bit tags.

In the following subsections, we will discuss in greater depth the hardware implementation for the lazy version of Zarf, as well as the mechanism of its garbage collection.

4.2 Hardware Implementation

Let us begin by enumerating the major micro-architectural structures present in the design.

- The instruction heap, or “IHeap,” stores the binary program. It is loaded once by a sequence of loader states, and then never changes (Zarf, targetting simple embedded platforms, does not support dynamic loading or program mutation).

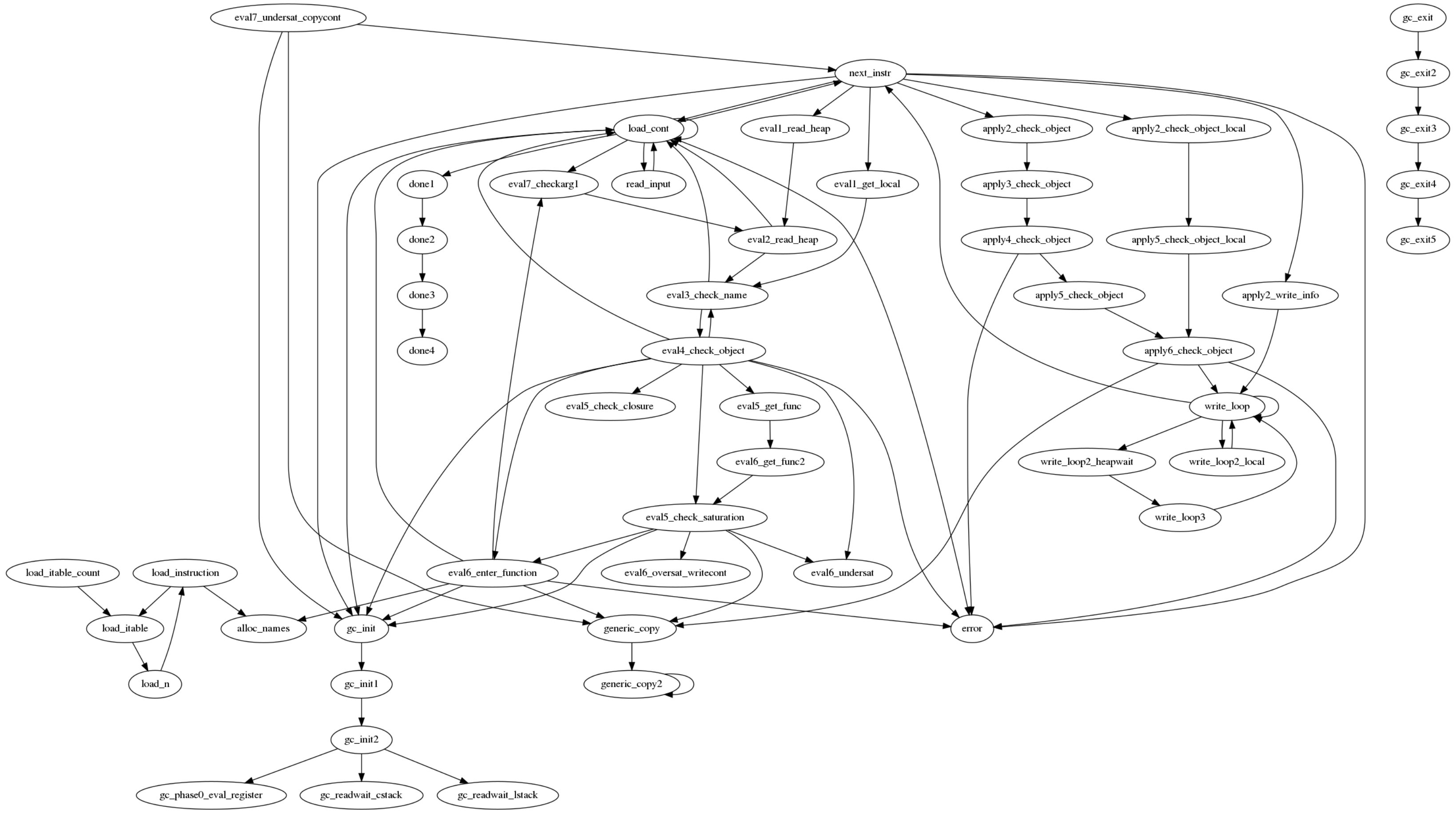


Figure 4.1: State machine for the lazy implementation of the Zarf hardware. The main garbage collection phases (21 states) are omitted.

- Along with the IHeap, a set of info tables ("ITables") are created during loading that provide a fingerprint for each function. Most of the fingerprint is statically included in the binary; added to that static portion is an entry point assigned during loading, which references the first instruction of the function. The static fingerprint includes a 1-bit tag indicating if the function is a constructor, an 11-bit field for the arity, 10 now unused bits[3],and 10 bits for the number of locals the function will allocate (at maximum).

- The Name Table, already discussed in the introduction of this chapter, provides a degree of indirection for all heap objects. It contains a 1-bit tag indicating if the stored value is a heap reference or an integer, and then a series of bits interpreted differently depending on the tag. If it is an integer, the remaining 32 bits contain the integer's value. If it is not, it contains a heap address (whose width is the log of the size of the heap, a parameter that is not necessarily fixed for the Zarf design) and a GC tag (whose width is also an adjustable parameter). GC tags are discussed more in Section 4.3.

- The Locals Stack, or "LStack," contains the name referring to the result of each `let` allocation. Every `let` performs allocation, so each will result in a name. As already mentioned, the hardware pre-allocates a name for each possible `let` instruction before beginning execution of the function. The LStack is the same width as the system's name-width (which is set by the Name Table size). A bigger Name Table means a smaller likelihood of stalling on name allocation, but a larger hardware overhead for the Name Table and LStack.

- The continuation stack, or "CStack," is what's used to store state when execution jumps for closure evaluation. One entry has a 3-bit tag indicating the type of continuation (case continuation, arg continuation, thunk continuation, two-op primitive continuation, or one-op primitive continuation; the different continuations are discussed below), space

[3] This field was originally for the number of free variables (mentioned in Section 3.5).

for two pointers into the LStack (the base and top of stack for the current frame), the pc, and one name. This final name is the contents of the "env_args" register, which identifies the current function context (from which closure arguments should be drawn). Notably, we do not have to store the contents of the "eval_register," which stores the value of the most-recently-evaluated thing for use in `case` comparisons. This is because a new continuation means a new `case` instruction, so the value of the eval_register will be overwritten on return.

- The final major structure is the Allocation Cache ("ACache"). This is essentially a small cache for free names, pointing to an unused or reclaimed entry in the Name Table. This structure is an optimization that helps prevents stalls on name allocation; it is discussed more in Section 4.3.

The number of locals is required in the function fingerprint because the lazy machine supports forward references, meaning a `let` instruction can reference the result of a `let` instruction further down in the function, which has not been executed yet. To support this, we must pre-allocate a name (reference) for each local variable before beginning execution of the function. Once the actual allocation takes place, we can bind the name in the Name Table to the resulting closure in the Heap. Before that, the name still exists and gives a handle for forward references to work. Note that a forward reference can only extend as far as the current block of uninterrupted `let` instructions — i.e., you cannot refer to a `let` instruction past the next `case` instruction. This prevents one from references a closure that does not exist yet, and then requiring the result of that closure in a branch instruction, which would be impossible to resolve.

The decision to allow integers to be stored in a Name Table entry costs extra bits for the Name Table (because fitting a 32-bit integer takes more space than a heap address, which, for our embedded systems, is generally on the order of 20 bits), but is an optimization that prevents

Purpose	Number of States
Program Loading	4
Function Application	15
Function Evaluation	18
Garbage Collection	29
Total	66

Table 4.1: Number of states for each purpose in the lazy Zarf hardware implementation.

the need for boxed integers in the Heap. The underlying principle here is that we should never duplicate work and evaluate the same closure twice — this could have disastrous results and destroy correctness if a closure contained I/O, but otherwise is an important performance optimization. So, if a closure evaluates to an integer, we have to note that that reference now refers to the evaluated integer, not the unevaluated closure. One way to do this is to overwrite the closure in the Heap with a boxed integer, but another method (that saves the extra cycles of reading the Heap) is to store the final integer directly in the Name Table.

Pre-allocating names before beginning function execution is an overhead that scales with the number of locals in a function. For each possible local, it is (at best) one extra cycle to allocate the name and write it to the LStack. At worst, there is a name allocation stall and we have to wait for a new free name to be found (these concerns are discussed in the following section, 4.3). One cycle per local is amortized over the execution of the whole function, so the only place where the overhead becomes a concern is where the machine allocates a large number of locals, using *n* cycles, but then the function only ever actually allocates a few of the locals (much less than *n*). This can occur if a function has branches of drastically different lengths. The function fingerprint (ITable) must indicate the maximum number of locals that could be allocated on the longest branch. The hardware doesn't know when it starts the function which branch will be taken, so it must plan for the worst case and allocate all of the indicated locals.

With the major micro-architectural structures covered, let us go over the structure of an

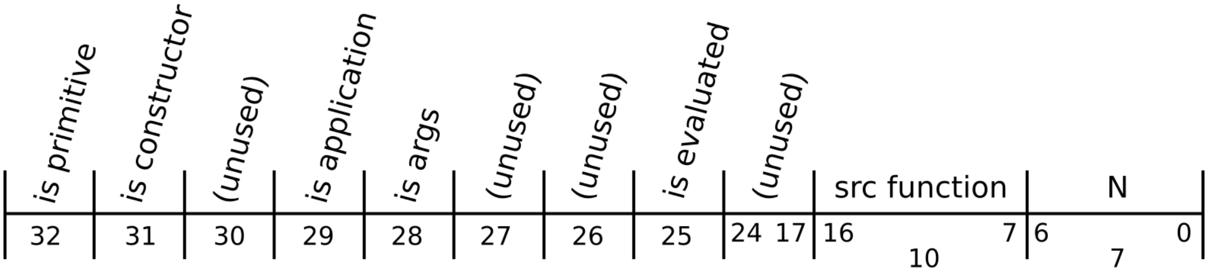


Figure 4.2: Structre of the “info word” of heap objects in the lazy version of the Zarf hardware.

object in the Heap. The `is_primitive` tag is one of several flags and pieces of information encoded into heap objects on the lazy Zarf platform. Every heap object begins with a one-word “info word” that contains flags and fields of information. Figure 4.2 shows what they are, and what bit-spaces they occupy. There are five single-bit flags, two integer fields, and four currently-unused portions. The fields are used as follows.

- `is_primitive` distinguishes between the object being a primitive integer or an object (closure) of some sort. If it is an integer, the remaining fields are irrelevant, as the 32 bits just encode the integer’s value.

- `is_constructor` is set to 1 if the object is a fully-saturated constructor. The `src_function` field will indicate what the type of the constructor is (which function ID), and the `N` field will indicate how many fields the constructor has[4].

- The first unused flag (in bit 30) formerly indicated if the object was a closure holding free variables. The free variable optimization, which was removed in later versions of the ISA, is mentioned briefly in Section 3.5.

- `is_application` tells if the closure object is a generic function application that has at least the expected number of arguments. It’s possible there are extra arguments, which will be handled when the closure is evaluated.

[4] How many fields are in the constructor are statically knowable, but it is convenient in some places for the hardware evaluation mechanisms if the information is accessible in the info word.

- The is_args flag is set to 1 when the closure contains excess arguments, created from the evaluation of an oversaturated closure. A corresponding argument continuation will be placed on the continuation stack; it points to the args closure so the arguments can be recovered when an undersaturated function is returned, requiring them.

- The next two unused fields (bits 27 and 26) formerly distinguished if the generic application was an application on a function or a free-variable closure. Since free variables were eliminated, these options no longer needed to be distinguished.

- is_evaluated is necessary because evaluation is lazy, but crucially each closure can only be evaluated once — otherwise, side effects (like I/O) may be repeated, violating correctness. When a closure is evaluated, its info word is updated with a "1" in this flag, and the src_function field will hold the name of whatever it evaluated to.

- The next unused field, bits 17-24, was also a result of the free-variable "optimization." Since the space of possible dynamic names is large compared to the space of possible functions (~17 bits vs 10 bits in the current iteration), an extra set of bits was needed on top of the src_function field to hold a name of a source free-variable closure. This was required if a function applied arguments to a free-variable closure: the info word would store the name of the source free-variable closure, and the corresponding function application closure would hold the arguments.

- src_function is a 10-bit field that holds the function ID for generic applications. That way, the hardware knows what function the arguments are being applied to. If the object is a constructor, then this field holds the function ID of the constructor.

- The final field, N, is a 7-bit field that indicates the number of arguments in this closure. It's used for constructors, generic applications, and argument closures. N determines the length of the object in the heap, and is also critical for garbage collection.

If `is_primitive` is set, the remainder of the fields are ignored, because they hold the integer's value. `is_primitive` must be 0 to interpret the other fields. The remaining 1-bit flags are mutually exclusive; i.e., of `is_constructor`, `is_application`, `is_args`, and `is_evaluated`, only one may be set at a time.

There are 62 registers in the lazy Zarf design, so we do not enumerate them here. Several are only 1 or a few bits; many of them are pointers into memories. A large set are used only in garbage collection.

The state machine governing the operation of the lazy Zarf implementation is illustrated in Figure 4.1. Garbage collection main states are omitted for brevity (a summary of those states is found in Figure 4.3 in Section 4.3). Similar to explaining the usage of every register, going over the operation of Zarf state-by-state would be an unnecessary level of detail. Instead, we summarize the major regions of the state machine. We note that one major factor in the state machine design is the use of synchronous memories. This means that reads (in addition to writes) can only occur on clock edges. The requirement causes many more states to to be added to the state machine. For example, casing on a local variable requires reading from the LStack, putting that name through the Name Table, and then using that address in the Heap to get the obejct's info word. Now, each of those requires its own cycle. We enforce the use of synchronous memories to allow the design to be synthesized to an FPGA, where the memories (BRAMs) have only a synchronous-read mode. The change to the design over the previous iteration is most likely an improvement, despite the increase in states, because it brings the clock cycle time down.

Each instruction needs to reference a piece of data. Aside from `let` applying on a static function, `let` applies on a closure, `case` indicates a value to scrutinize, and `result` indicates a value to return. In addition, each arguments to `let` indicates a value or closure to pass in. Since the functionality is required in four places, an orthogonal piece of functionality called the "data mux" performs the necessary steps of multiplexing in the appropriate signals depending on the bits of the current instruction. Some of these are readily available in one cycle (such as reading a local name off the LStack), but others require multiple cycles to complete (such as reading a field from a constructor, which requires going through both the Name Table and the Heap). The data mux's functionality only extends for one cycle; properly handling multi-cycle behavior is the responsibility of the part of the state machine that invokes it.[5]

The data mux configures two values that are used to fetch the required data: data_basename and dword_offset. The first is either the base pointer on the LStack (in the case of a local being fetched) or the name of the object being accessed in the heap. Dword_offset is the offset from the base to the desired data; e.g., the relative index on the LStack or the field within a constructor.

The state machine begins with an initial state into which the machine boots. At the RTL level, we assume a power-on reset functionality is baked into the system, causing all registers to start with an initial value of 0. State 0 is then hardwired as the initial state (the remaining states are parameterized by state name; the actual state numbers do not show up in the design). The initial state handles setting a few initial values in certain registers and jumping to the loading states.

The loader states read the input from input port 0. The program is read one 32-bit word at a time. The structure of the binary is such that the length of the binary, and the length of each function, is included and transmitted. This means the loader knows exactly how many words are going to be received, and the structure of the binary (which words belong to which function) is properly conveyed. When the loader is finished, it begins execution of function 0x256, which is the first function loaded and always defined as "main."

The top-level dispatch state is called "next_instr." It is the most complex state, because

[5] While a better programming style would be for the data mux to fully handle the responsibility of fetching necessary data, we are limited by the paradigs of hardware design. Writing a functional unit that operates within a single cycle allows it to be inserted anywhere in a design, but a piece of hardware that takes multiple cycles cannot be arbitrarily called and inserted.

in addition to branching for each different instruction, the first cycle of functionality of each instruction is included in its definition. For `let`, it decides if the application is happening on a dynamic object (closure) or a static function, configured a load from the LStack, Name Table, or ITables as appropriate. On a `result` or `case` instruction, the state configures a load from the LStack (in the case of casing/returning a local) or the Name Table (in the case of an argument). If an integer is the argument to the `case`/`result` instruction, the state machine moves on to the next instruction, because the value is already evaluated. Otherwise, it jumps to the appropriate region of the state machine.

To process function application, a set of "apply" states are used. They handle writing the info word in the Heap and copying arguments into the structure. As mentioned, the states need to be able to handle both static function application and application on a dynamic closure. For closures, there is a path of states that read the info word from the heap to discover the source function, as well as the number of arguments already in the closure. The states create a new object with the combined list of arguments (the ones in the closure first, followed by those specified in the program). In either case (static or dynamic application), the job is done when the closure object is created, because we are using lazy evaluation.

The two branches of the apply states have two helper routines used through the full state machine. These are effectively subroutines that can be invoked from different places; a set of input registers are configured, a register stores the return state, and then the subroutine executes until finished. One of the two subroutines is a "write loop," which handles writing the arguments specified in the program into the Heap. This is used to process the argument list in the `let` operation. The other we refer to as "generic copy," and it copies arguments from a source object (name) into the heap. It is used, for example, in application on a closure: before we put the new arguments in the new closure, we have to copy the arguments from the old one. This shuffling of arguments is required in several places throughout the state machine.

Processing either a `case` or `result` instruction triggers the "evaluation routine," because both instructions require unevaluated closures to be evaluated to weak head-normal form. The evaluation routine takes a closure and does such evaluation. First, it reads the Name Table to see if the given name references a primitive value; if not, it is an object. Then, it checks the info word in the Heap output to see what type of object it is. A constructor is already evaluated; a closure is not. If it is a closure, it reads the ITables to see how many arguments are expected and determine if the closure is properly saturated (how many arguments are present vs how many are expected). If oversaturated, it writes an argument continuation before entering the function — the arguments will be consumed when the function returns; it must, if well-formed, return an undersaturated closure that expects additional arguments. If the closure being evaluated is undersaturated, it reads an argument continuation from the CStack (if one isn't on the top of the stack, an error has occured); then, it creates a new closure that contains both the arguments from the old closure and the arguments referenced from the continuation on the CStack (this is done via two calls to the generic copy routine). One the new closure is created, it starts evaluation over to determine the saturation of the new closure, writing argument continuations or consuming them as necessary. If the closure being evaluated has the correct number of arguments, the evaluation routine jumps into it.

When the evaluation routine is about to jump into a closure to evaluate it (begin execution of the closure's function), it first pushes an update continuation onto the CStack. This continuation will be seen when evaluation of the closure finishes; it signals to the system to perform an update of the name currently pointing to the closure to instead point to the evaluated value. Requiring update continuations is a result of the lazy abstraction: when things are evaluated immediately, there is no need to track updates for later.

When the evaluation routine hits a final value (a constructor or an integer), it moves to a state called "load continuation." Load continuation handles all implicit control flow, deciding what to do next based on the continuation on top of the CStack. If the CStack is empty, main has finished and execution can terminate. The continuation on the CStack will be identified with a 3-bit tag. A case tag will cause state restore and execution to resume at the indicated point; the newly-evaluated value will be the eval_reg, which can then be used in comparisons. If an update continuation is on the CStack, we overwrite the indicated closure's info word to point to the evaluated constructor that was the result; if an integer was the result, we update the Name Table and no forwarding needs to take place. If we see an argument continuation on the CStack, we throw an error — it means there were excess arguments fed that were never consumed.

There are two primitive evaluation continuations that can be on the CStack. If we see one from load continuation, we know we are in the middle of a primitive operation. When we evaluate a 1-op primitive, we push the continuation, then evaluate the argument; when evaluation is done, we see the continuation and know to do the operation. WHen we evaluate 2-op primitives, we push the continuation, then evaluate the first argument; when evaluation is done, we see the continuation and know based on a flag there that we now need to evaluate the second argument. The flag is updated and the evaluated first argument is stored in the continuation. When we see the continuation the second time, we read the first argument back out and do the operation. This system allows us to execute primitive operations in a lazy evaluation context where evaluating an operand could spawn arbitrary additional computation.

Finally, there is an error state which writes an error constructor and then goes to the load continuation state to push the error up the stack.

Performance of the Lazy Zarf Platform

One important standing question is how Zarf performs compared to a standard embedded C processor. This can be a difficult comparison to make, for a number of reasons. One is the apples-to-oranges nature of a functional processor vs and imperative one, especially when the imperative processor has benefited from decades of research and commercial optimizations. However, we strive to make as accurate a comparison as possible, which is summarized in

	Zarf			**Idris–ARM**			**C–ARM**		
Program	inst	cyc	mem	inst	cyc	mem	inst	cyc	mem
Fibonacci	1.6k	7.2k	6kB	110k	153k	35MB	2.4k	5k	0B
Hanoi	70k	322k	166kB	18.5M	23M	35MB	465k	731k	0B
Quicksort	81k	315k	15kB	5.2M	6.3M	35MB	92k	143k	400B
Matrix Mult	122k	618k	18kB	7.5M	9.2M	35MB	180k	249k	1.1kB

Table 4.2: For each benchmark the instruction count / cycle count / memory usage is shown. The first set of columns contains statistics for programs executing on Zarf. The second column is the same programs, written in Idris, compiled for ARMv7 and run under gem5. The final set of columns is native C implementations, also run using ARMv7 on gem5.

Resource	Zarf	MicroBlaze
LUTs	4,337	1,840
FFs	2,779	1,556
Cycle Time	20ns (50 MHz)	10ns (100 MHz)

Table 4.3: Resource usage of Zarf and basic MicroBlaze (3-stage pipeline) on an Artix-7 FPGA. In total, the logic of the lazy Zarf implementation uses 29,980 gates.

Tables 4.2 and 4.3.

Table 4.2 shows the dynamic instruction count, execution time in cycles, and heap usage of four different simple programs on the Zarf platform, an Idris (a research functional language using dependent types) implementation on ARM, and a C implementation on ARM. The later two platforms were executed using the Gem5 architectural simulator [80], which simulates a more complex CPU; to combat this, the parameters were adjusted so that only 1 instruction is dispatched or committed per-cycle. This forces the super-scalar, out-of-order CPU model to imitate operation of a simple embedded CPU. Further, we adjust the size and latency of the L1 cache to take up the entire memory space with only 1 cycle of delay; this is to emulate the small block of SRAM that would be available on embedded device as its memory. We see Zarf generally fall in the middle of the two other options, meaning Zarf is generally better than using a functional language on an imperative processor, but not so good as an optimized C version. However, there are several finer points in the table worth mentioning.

Notably, the cycle count for Zarf is orders of magnitude better than Idris. There is a caveat that the cycle time for the prototype lazy Zarf was approximately twice that of the imperative comparison processor (see Table 4.3). Despite this, the figures still suggest using Zarf would be much faster than some functional languages on imperative processors. As bad as very-imperative benchmarks, like matrix multiply, are on Zarf, they are much, much worse for the Idris alternative. The Zarf figures compared to C, meanwhile, are within one order of magnitude — and better in one case. Zarf is around 2x the number of cycles in two cases, 40% more in one case, and 50% *fewer* in one case. Again, these figures need to be mediated by the slower cycle time, but even then, Zarf performs surprisingly well for its high-level interface. We assume the cycle count compared to C is best on Hanoi because it is a highly-recursive program: C pays overhead for every function call, while functions are native on Zarf and have a smaller overhead. For Quicksort and Matrix Multiply, the 2-4x Zarf slowdown makes sense, because these are optimized, array-based programs for C. Zarf does not have mutable arrays, and so must used linked-lists and other structures, which incur added overhead.

Memory usage is a little trickier to compare — we were only able to measure heap usage, not heap and stack, for the imperative processor. For Zarf, though, we list total memory usage. This does not necessarily correspond to peak memory usage, because of garbage collection; rather, it is the total memory used over the course of the program. The Idris runtime begins by allocating a big block of memory, and then handling actual object allocation internally. We were not able to measure those internal allocations, so all we have is the initial 35MB block that the runtime claims at start time.[6] On the C side, two of the benchmarks used no heap allocation, because they are entirely recursive functions that only use the stack.

The instruction count for Zarf has an even better comparison vs Idris than for cycle count; similar to the cycle count, it is orders of magnitude better. Part of this is due to the high-level natur of the Zarf ISA. Performing the equivalent of a Zarf instruction on an imperative

[6] Though it seems uninformative to include the 35MB figure, it does make the important point that if one wanted to run functional code in an embedded setting, a statically-linked executable compiled from Idris would still be unsuitable, because the platform likely does not have enough memory.

architecture takes several native instructions. However, as a tradeoff, each Zarf instruction takes multiple cycles, where the imperative instruction most likely takes 1 (on average; pipelining and stalls clearly affect the throughput). We note that the instruction count for Zarf is also smaller than the instruction count for the C programs. This suggests that, if pipelining were very successful, performance could be brought very close to the C version.

Table 4.3 shows the resource usage of the lazy Zarf implementation on an Artix-7 FGPA, compared to the Xilinx soft-core MicroBlaze (configured with 3-stage pipeline). The MicroBlaze stands in as our "baseline" of what a very small microcontroller might look like. Despite using around twice the look-up tables (LUTs) of the MicroBlaze, Zarf is still only using 7% of the available resources on the FPGA. When synthesized as an ASIC for 130 nm, the final design takes up .274 mm^2 (including memories). Though larger than the MicroBlaze, the Zarf implementation is still quite a bit smaller than many common microcontrollers.

4.3 Garbage Collection

Garbage collection (GC) is handled by the Zarf hardware. Making memory management entirely a hardware responsibility helps enforce purity and transparency at the ISA level, since it prevents software from reaching outside of its accessible state or mutating any piece of state. The garbage collector is a tracing copy-collect semispace garbage collector. The heap is divided into two semispaces. One is the "from" space and one is the "to" space; after each collection, their roles flip. Beginning with a root set, the garbage collector recursively traces all live objects in the system, copying them from one heap semispace into the other.

The root set is composed of the locals stack, the continuation stack, the evaluation register[7], and the argument register[8]. There is a deprecated fifth member of the root set: as discussed,

[7] The evaluation register holds the object currently under evaluation; i.e., the thing most recently case'd on.

[8] The argument register holds the name of the closure used for the current function context (from which arguments are drawn).

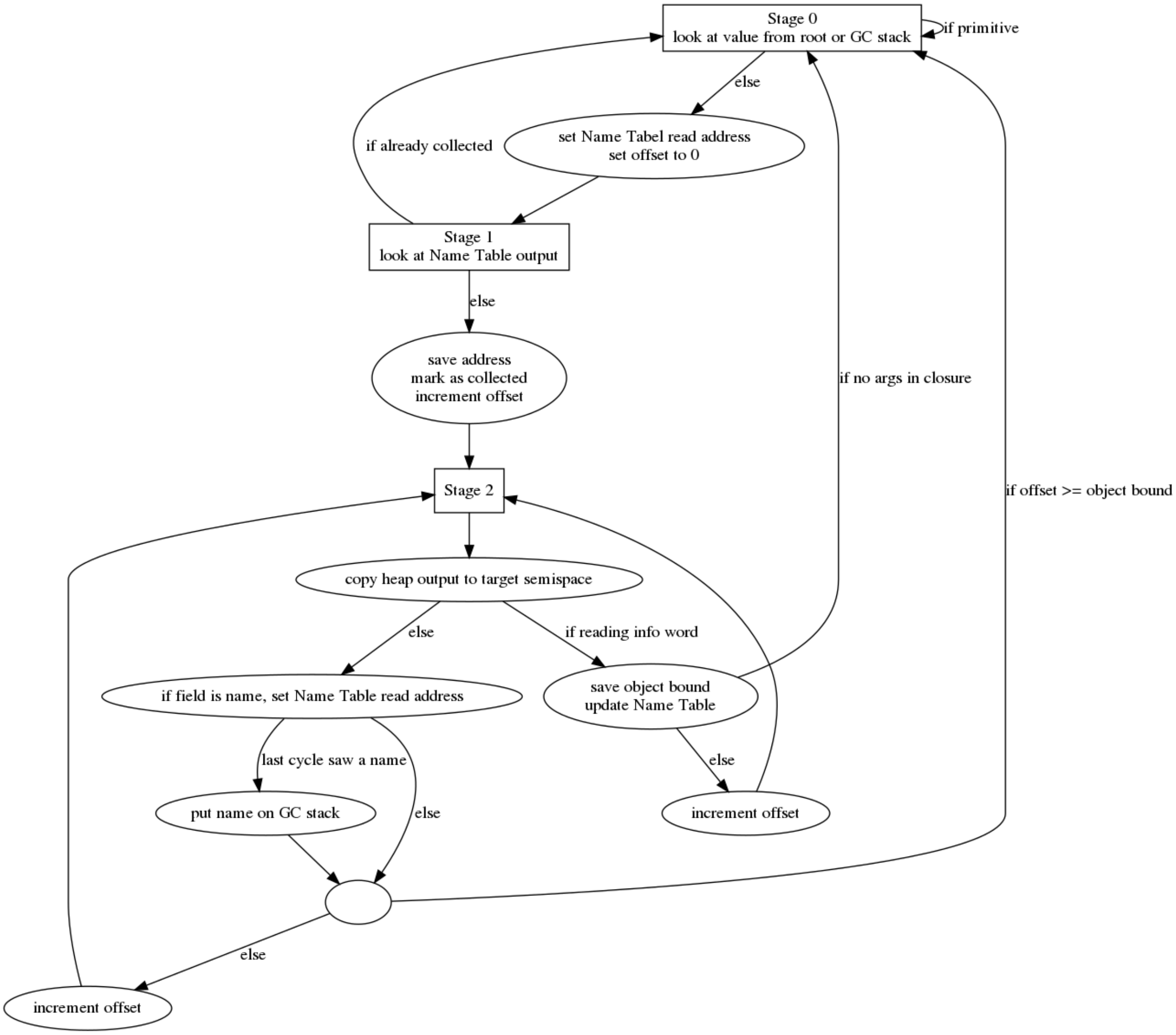


Figure 4.3: Model of garbage collection steps. Square boxes indicate actual hardware states (sequential cycle change), while actions are grouped in circles for easier reading. The actual implementation is a “Mealy” type state machine, where edges control outputs (rather than a “Moore”).

an earlier design of the machine had “free variables” in addition to arguments. The register holding the closure for the free variable context was also in the root set.

In addition, the locals stack is used during garbage collection as a GC stack; as objects are traced by the system, whenever an uncollected object is seen, it is put on the GC stack to be collected in a future iteration of this GC run.

For each of the GC roots, and the GC stack, there is a three-state state machine that examines values and walks through objects, doing the copying and adding things to the GC stack as necessary. Some meta-design code instantiates the three-state machine for each of the roots and the GC stack, stitching them into a larger 29-state machine. This total comes from the five members of the root set (including the deprecated one, which the lazy hardware still considers) with the GC stack; each of those has an instantiation of the three-state machine, making 18 main GC states. In addition there are 3 initialization states, 5 exit states, and 3 waiting states for reading from synchronous memories. Altogether, this makes the 29 states seen in Table 4.1.

Figure 4.3 shows, with some abstraction, how the three-state garbage collection works in the hardware. Showing the actual operation with a three-state state machine would be too difficult to read, so it is abstracted as a “Moore” style machine where outputs occur in states, rather than on edges. Square boxes indicate hardware cycle changes (actual states).

Stage 0 begins by looking at the next value for collection; based on the higher-level state machine, this could be something from the root set, or the next value on the GC stack. If the value is just a primitive, it loops, incrementing the stack pointer as appropriate; otherwise, it must be a name, so it sets the Name Table read address and initializes the offset register to 0.

Stage 1 looks at the Name Table output, which includes a tag for garbage collection. If the tag indicates the object has already been collected, we loop back to Stage 0. Otherwise, we save the address (so we can address the heap without going through the Name Table), mark the object as collected in the Name Table, and increment the offset register[9].

[9] The offset register affects the heap read address, which itself is registered, so changes to the offset take 2

Stage 2 is the complex portion, where we walk objects and make decisions based on what we see. First we copy the heap output into the target semispace, whatever it may be — we're walking through the object, so it may be the info word or a field of the closure. We can keep track of when we see the info word with a 1-bit register that is reset in Stage 0. If it's 1 when we hit Stage 2, we know it's the first word of the closure, and therefore the info word; after that, we set it to 1. If we're reading the info word, we save the bounds of the object (its number of fields) and update the Name Table with the new physical address for this name (the current free pointer). If there are no arguments in this closure, we go back to Stage 0 to collect the next object; otherwise, we increment the offset and loop to Stage 2 to continue our walk.

If the heap output is not an info word, we check if it's a name (a reference). If it is, we set the Name table read address to that value so we can check next cycle if its been collected yet. We set a special flag that lets the state machine know it's checking a field in the Name Table. We also check the flag value from last cycle; if last cycle was looking at a name, we put it on the GC stack if the Name Table output says it isn't already collected. Finally, we check if our offset has exceeded the object bound; if it has, we go back to Stage 0 and collect the next object; if it hasn't, we increment the offset and loop to Stage 2 to continue our walk.

As mentioned, the Name Table stores a tag with each reference so we can tell if it's been collected already. There is one complexity that arises from the usage of this tag in name reclamation. The names are resources themselves in the system, and must be collected. Because we are building a hardware system, this can be done in parallel with normal operation. Naively, we would want a 1-bit GC tag for each reference, which tracks if the object has been collected yet. So initially we move objects from the "0" space to the "1" space, and correspondingly update the tags from 0 to 1, and then next time GC runs, we move things from "1" to "0."

Now, imagine that Name Table reclamation occurs after garbage collection is finished: any name tagged with the currently active tag must be assumed to be live and the name cannot be

cycles to propagate.

reclaimed. The issue is that our garbage collection algorithm only ever touches lives objects. This would typically be a benefit, because collection time is proportional to the live set, rather than all objects. Here, though, it means that objects that have gone out of scope keep the current "live" tag and cannot be reclaimed. Furthermore, the problem is not necessarily solved by garbage collection running, because if the name is not reclaimed soon, GC will just run again, and now the dead name's tag matches the live tag again.

This is further complicated by the reclamation process itself: there is no "free list" of names available for allocation. There's just a pointer into the Name Table marking the next free name[10]. So, even if a linear scan could reclaim all the free names in-between executions of garbage collection, there's no place to store them without incurring a new hardware cost as big as the Name Table itself. Instead, the linear scan only cares about finding a free name for the next allocation. This means that long periods of time can elapse before the scan loops around and comes back to a name, and therefore GC tag collisions have a real probability of occurring. Luckily, this does not affect correctness, merely performance: sometimes, we pass over a dead name because the hardware thinks it may be live.

One solution to this that we implement is to use more than 1 bit for the GC tag. Assuming a uniform distribution of objects belonging to different GC tags, the odds of a dead object being mistaken for a live one are $\frac{1}{2^N}$ for an N-bit tag, because those are the odds that its random tag happens to match the currently live tag. We found that with an 8-bit GC tag, there was never a stall in name allocation for any of our benchmark programs.

That fact, however, is also the result of a small Name Cache. The Name Cache is a small memory block (currently configured to 256 entries) that stores names that are known to be free. This prevents stretches of allocated names in the Name Table from causing stalls as the linear scan moves past them. Instead, the linear scan is constantly running, filling the Name Cache

[10] There is a small cache of free names, but it does not solve the GC tag problem; it merely lowers the likelihood of observing a stall because of it.

on one end, while name allocation draws from the other. In addition to multi-bit GC tags, it resulted in never observing stalls due to name allocation demands in any of our benchmark programs.

Chapter 5

Strict Implementation

5.1 Introduction

In the timeline of Zarf development, the strict implementation occurred after types were added to the ISA (covered in Chapter 7). Because it simplifies the run-time hardware checks that are required, for the strict implementation it is assumed that all programs on the platform are typed. The run-time checks that can be eliminated are discussed in Chapter 7.4 and summarized in Table 7.1. The change to strict evaluation also vastly simplifies the information that must be encoded in a heap closure info word, and therefore also simplifies the hardware that handles info words, because there are fewer cases to deal with.

We do note that the one-bit tags that differentiate objects from integers are still required on the datapath in the system, because garbage collection needs to make that distinction.

Aside from simplification due to types, the other major change in the strict hardware is the use of an argument stack instead of argument closures. This is a direct result of the strict evaluation paradigm: because we jump into a function right when we perform saturated (or oversaturated) application, we can place the arguments on a stack, as they do not need to survive beyond this scope. There are complexities in dealing with the stack — especially when it comes to shuffling arguments for oversaturated/undersaturated application — but these are, again, managed by the hardware evaluation engine.

The former heap closure info word structure (Figure 4.2) contained several flags for differentiating the type of object in the heap. Now, there are exactly two cases: a closure, representing an undersaturated application, or a constructor, holding data fields. Notably, a closure *must* be an undersaturated application, because we now strict: the lazy machine would build closures for every application, but now we build them only when necessary (undersaturated applications can leave the current scope, so must live in the heap). The flag for primitives, as mentioned, is still present for arguments of a closure and fields of a constructor. The flag for constructors is no longer needed, because we statically know where constructors show up thanks to our type system. We are guaranteed that a closure will never be used as a constructor (or vice versa), and so, trusting in our type checker, we do not need to differentiate at runtime. As we will discuss in Chapter 7, without a hardware type checker, one could compile or code a binary that causes crashes; but with binary checking, it is impossible.

The lazy info word had two one-bit flags to distinguish between a generic application and an argument closure. Both are no long required. The generic application no longer exists because in strict evaluation, we do not build closures for normal applications. Similarly, because of the strict evaluation, arguments now live on the stack: this means excess args do not end up in closures, but stay on the stack, so no flag is required for an argument closure. The final flag (for evaluated objects) is not necessary as we no longer lazily evaluate objects. A function is evaluated immediately upon saturated application, so there is no object representing the application to be updated and evaluated later on.

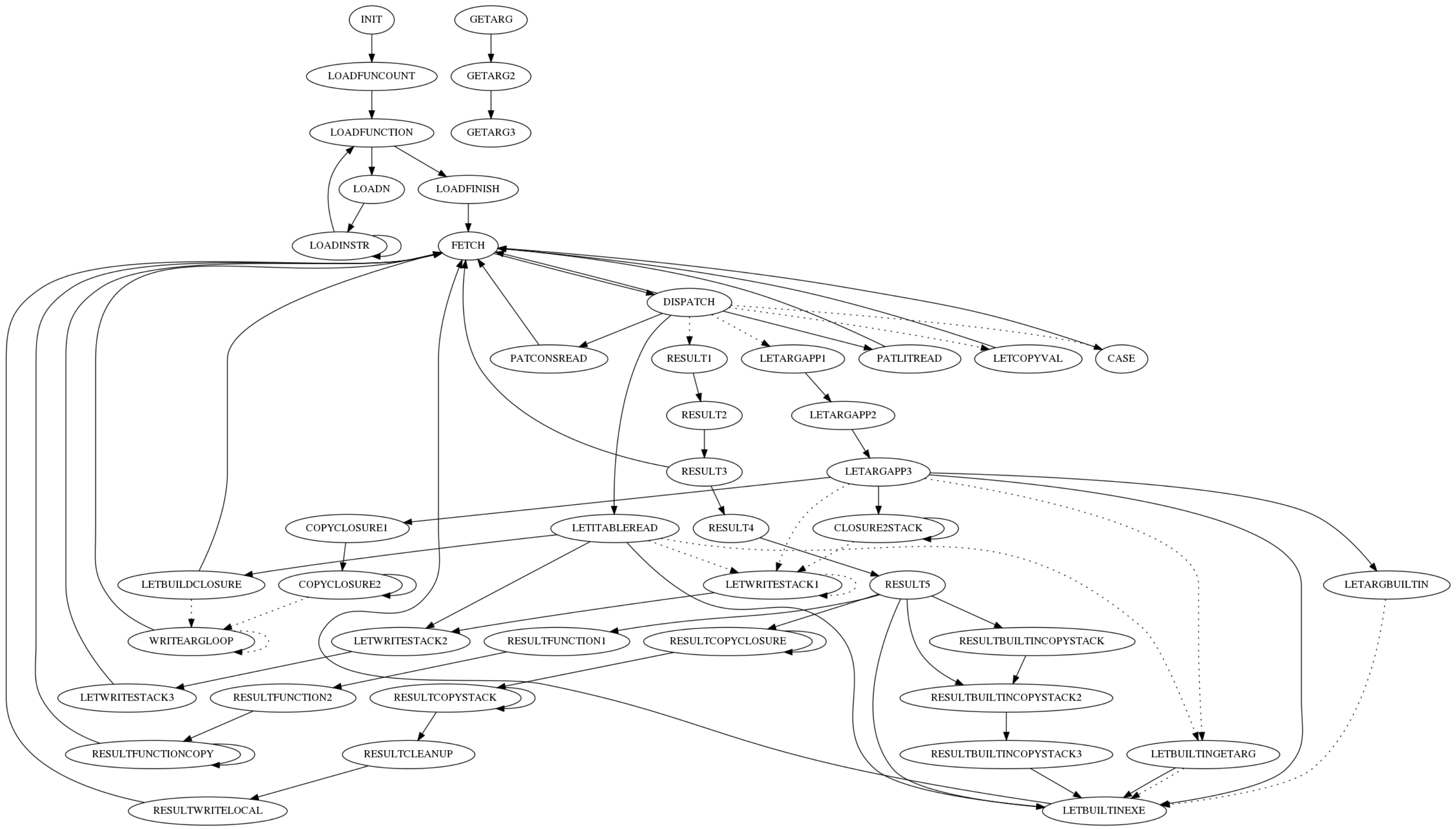


Figure 5.1: State machine for the strict implementation of Zarf hardware.

Initialization/Loading	6
Argument Fetching	4
Dispatch	2
Let Application	11
Closure Copying	3
Result	14
Case	1
Pattern_Cons	1
Pattern_Lit	1
Garbage Collection	11
Total	54

Table 5.1: Number of states devoted to each purpose in the strict Zarf state machine.

5.2 Hardware Implementation

The previous subsection categorized the major differences — heap objects and stack usage — but now we go more in depth into how the strict evaluation semantics are implemented. Many of the same hardware structures are present. As before, we enumerate and describe them.

- The instruction memory, or IMem, holds the binary of the program. As with the lazy architecture, it is loaded once and then never changed as the program executes. The limitation of no self-mutating code is now even more important, because all of our binaries are checked by the hardware type check (Chapter 7).

- The static info tables, or ITables, which hold fingerprints for each function. The data for each entry is the same as with the lazy version: a flag for constructors, a field for arity, an unused field, and the maximum number of locals.

- The Heap is used for objects that have arbitrary lifetimes. As discussed, objects are more limited in the strict evaluation semantics. The primary two are closures and constructors, which we distinguish statically using our type system, but not at runtime. Notably, closures are only for undersaturated applications. The third object type is a forwarding address, used only by garbage collection. As we will discuss in Section 5.3, the tracing collector makes two passes, the second of which is to update all forwarding addresses.

- The Stack — singular — is where local variables live. Before, we had two separate stacks: one for locals and one for continuations. Now that we use strict evaluation, the semantics of function calls are similar to traditional calls, and we can use a more-standard activation record on the single stack to track returns.

One of the fields in the ITables is actually no longer used in the strict implementation. Previously, maximum number of locals was required to allow for pre-allocation of names to enable forward references within a single block of `let` instructions. Now, we have eliminated the Name Table and cannot pre-allocate locals. Allowing for forward references without the extra degree of indirection would be very difficult, so we simply no longer allow them in the system.

To accelerate comparisons for `pattern` instructions inside of a `case` statement, we would ideally like the thing we are comparing against to be in a register. This is relatively simple when we `case` on an integer: just load the value into the register. When we `case` on a constructor, though, we have to read the Heap first to get the infoword of the constructor, and then we can compare against the function ID there. This read will occur on the first `pattern_cons` instruction encountered after a `case` instruction, but then the pattern (function ID of the constructor) is cached in a special register, which we call the "casecache." When the load occurs on the first `pattern_cons` instruction, we set a one-bit flag so that subsequent instructions know to use the cached value, instead of initiating another Heap read.

Once the scrutinee value is known, it is also stored on the stack. This because, under strict evaluation, (potentially) every `let` instruction causes a change of scope. If the scrutinee value were just stored in a register, it would be rapidly clobbered. We can't let this happen, as instructions may use the value to access data fields of a constructor. So, to persists, we reserve a special stack slot for the result of case instruction, just below local variables.

Figure 5.1 shows the full state machine (excluding garbage collection) for the strict implementation. Table 5.1 summarizes the number of states devoted to each purpose. Before, under lazy evaluation, let was relatively simple[1], while case was complicated (it had to handle evaluation). Now, let triggers evaluation directly and so has added complexity, while case requires only 1 state. Similarly, result has to handle automatic tail calls and the shuffling of arguments for undersaturated/oversaturated applications, where before it simply invoked the evaluation routine already in place for case instructions. The closure copying routine (3 states) can be invoked from either the let state machine or the result machine; it handles building a new closure when an existing closure is fed more arguments, but still results in an undersaturated application. One of the three states is used from let when a closure needs to be copied to the stack to evaluate it; this happens only when additional arguments are being applied and the result is fully- or over-saturated.

Figure 5.2 illustrates the Stack activation record for the strict Zarf implementation. Each function context is defined by a base pointer, bp, and Stack pointer, sp. Note that, because we are in custom hardware, we choose to use a separate memory block for the Stack. Since the memory is private to the Stack, we are free to start at 0 and grow upwards; if one were to use a single, combined memory for Heap and Stack, then it would start at the end of the memory space and grow down, as is normally the case. The function context's arguments are stored on the stack in reverse order, so argument 0 is at bp-1 and argument N is at bp-N-1. There are three stack slots for special values. (There are actually five values we need to keep track of, but to lower the function-call overhead, we combine some of the values into a single 33-bit word.) The first of these, stored at bp, is a combination of the old bp and the number of excess arguments for the current context. Just above that is the old PC, to which we will

[1] We note that it was conceptually simple, but required several states due to the delay of the Name Table in accessing the Heap, with branches to optimize faster cases (like applying on a local).

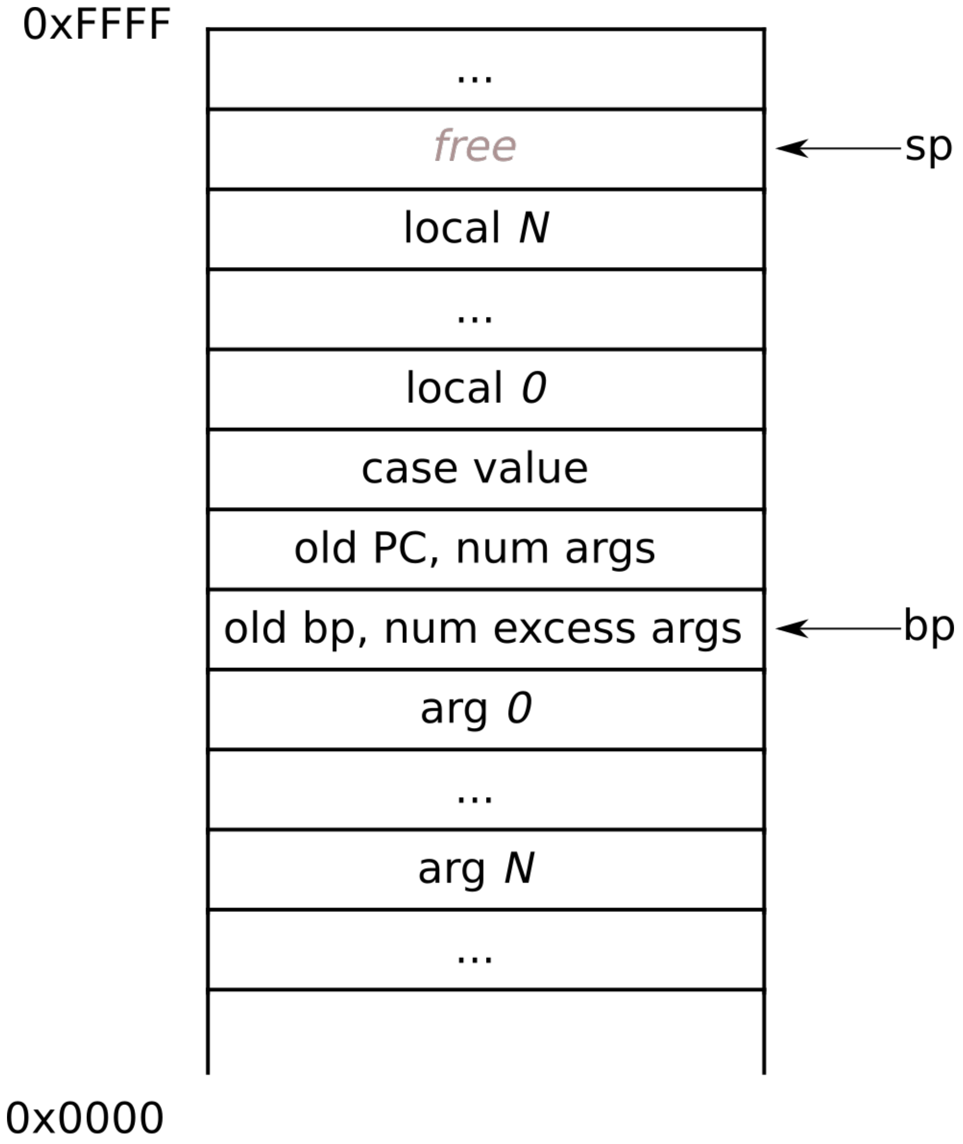


Figure 5.2: Activation record for the strict Zarf implementation.

return, and the total number of arguments on the stack. Above that, at `bp+2`, is the special slot for `case` values — after a `case` instruction, the value of the scrutinee is placed there for use in instructions. Local variable *n* is accessible at `bp+n+3`.

The number of excess arguments is required because we need to keep those arguments around when we clear the stack. At some point in the future, we will return to this activation record and have an undersaturated closure to return. When that occurs, we will copy its arguments onto the stack to combine with the excess arguments already there. This requires a bit of arithmetic on the part of the machine to place all the arguments correctly, but it is readily handled by an extra adder.

Our trick of combining values into a single Stack word does have its limitations and pitfalls. Obviously, it means the values in each of the pairs cannot be a full word, or the pair wouldn't fit on the Stack. The ISA places a limit on the maximum number of arguments a function can take; this is enforced mainly in the ITable binary encoding, since it has a fixed-width field. We currently set this limit at 1023 arguments (10 bits). This leaves 23 bits for the `bp` and PC fields (recall that we use 33-bit words to allow for full integers and 1-bit tags). Since we are working in the embedded space, 23 bits (8 million instructions or 32 MB) is more than enough for any program; similarly, 23 bits of addressable Stack words (again 32 MB) is likely more memory than we would ever include in the system.

However, if we wanted to expand Zarf to allow for larger Stacks, there are still tricks that can be employed. Since there is a limit on the maximum number of locals and arguments a function can have, an activation record has a maximum size as well. This means that an adjacent record (and thus the old `bp`) will be within some threshold value. If it is not in this 8 million word Stack segment, we should observe clear overflow in the old `bp` value (i.e., it is larger than the current one). Put another way, we leverage the knowledge that the previous frame will always be *below* the current one on the Stack.

If we wanted to allow for more than 32 MB programs, we would have to change the functionality of the hardware to use a separate word of the stack for return addresses. This would also incur 1 extra cycle of overhead on calls and returns to read/write the value.

As before, we will give a cursory overview of the state machine that makes up the Zarf implementation.

The top-level dispatch state is one of the largest in terms of code and functionality because of the branches for each instruction. The actions for `result` and `case` are quite simple: the state machine goes to the argument fetching routine (GETARG), with the subroutine configured to return to the appropriate state (the `result` handling states or the `case` handling state). For `pattern_lit`, we know the casecache has the value of the scrutinee, so we can directly do the comparison and adjust the PC without needing an additional state. For `pattern_cons`, it's possible the casecache has the pattern we want to compare against; if it does, we can do everything in one cycle as with `pattern_lit`. If not, we go to another state that will handle reading the Heap, doing the comparison, and then updating the casecache.

The final instruction, `let`, has three main branches in the dispatch state, depending on what's being applied on. If the source "function" is a literal (which comes up in instructions like "`let` x = 2"), the integer is copied to the stack and we move on. If the source is an itable, we have standard static function application. If the itable is a built-in function, and the correct number of arguments are given, we go to the GETARG subroutine to fetch the first argument, with the return state configured to a sequence of states that handles built-in application and execution. The final case is application on an argument, local, or constructor field. We branch on the special case where no arguments are applied (in which case we just copy a reference to the closure), and otherwise go to a series of states that handle dynamic function application (application on a closure).

Benchmark	**Lazy Zarf Cycles**	**Strict Zarf Cycles**	**% Improvement**
Fibonacci 100	8,068	5,212	35%
Hanoi 10	425,973	401,432	6%
Quicksort 100	487,980	478,795	2%
Matrix Multiply 10x10	148,371	115,948	22%

Table 5.2: Performance of the strict Zarf vs lazy Zarf for a small set of micro-benchmarks.

Performance of Strict Zarf

Figure 5.2 gives performance figures for the lazy version of Zarf vs the strict version. It is worth mentioning that the lazy version of Zarf has an RTL implementation (actual hardware design), while the strict version has a state-level simulation. The strict implementation will correspond one-to-one with states in the eventual strict hardware implementation, so we can still use that simulator for a fair comparison with the lazy version. Dynamic instruction count is not given in the table, because it is the same for the two implementations. The strict version of Zarf has an IPC (instructions-per-cycle) of .17-.18 for each of the micro-benchmarks (approximately 5 cycles per instruction). We can see from the table that the strict Zarf implementation is always an improvement, with the speedup ranging from 2% to 35%, based on the micro-benchmark.

Since we do not have an RTL implementation for the strict version, we cannot compare resource usage and cycle time, but the two implementation would likely be close.

The comparison between the lazy and strict Zarf also does not include GC overhead, because all of the micro-benchmarks can run in the available memory space (1 MB). GC overhead should also favor the strict implementation: the collection algorithm stays fundamentally the same, but without laziness causing "thunk explosion" (a proliferation of objects in the heap), there should be fewer objects to move on each run of GC. However, the collection algorithm only needs to move live objects; if most objects have short life-times, then perhaps there won't be too many more to collect for the lazy version.

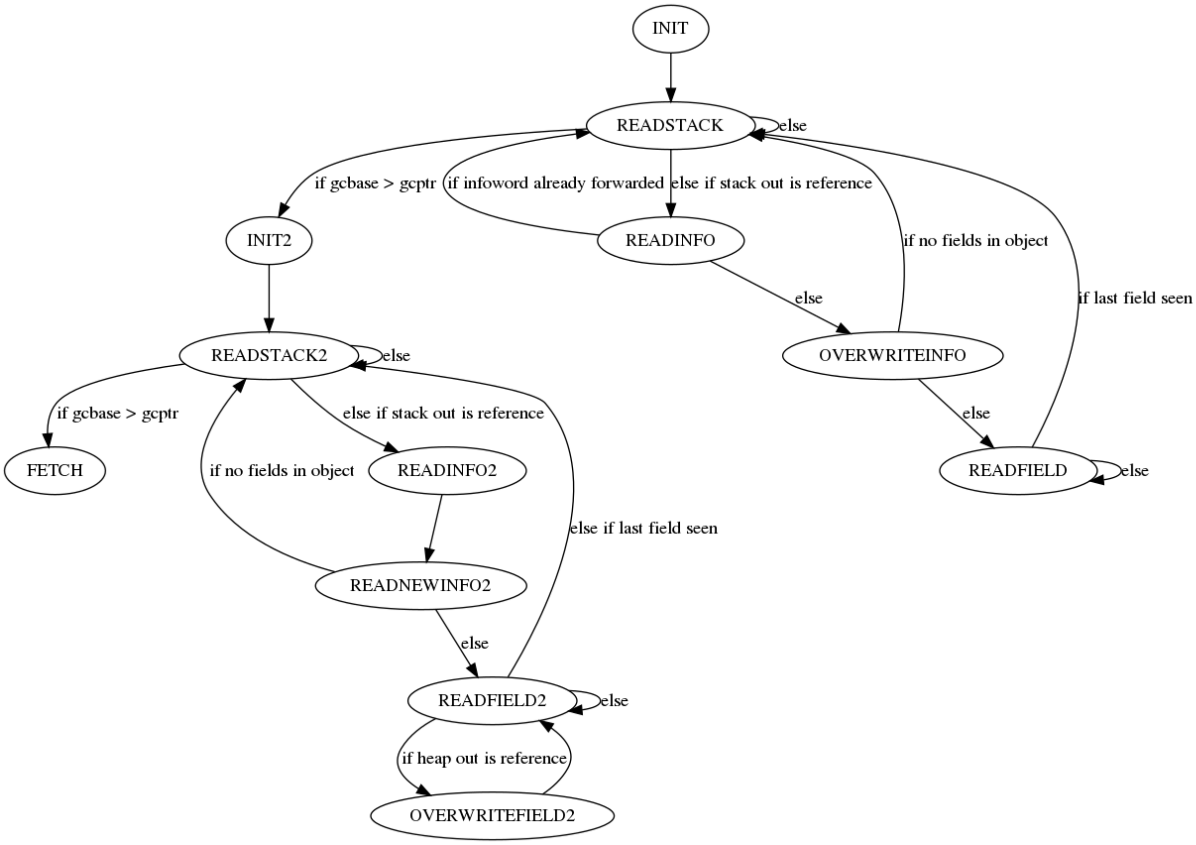


Figure 5.3: Full state machine for strict Zarf garbage collector.

5.3 Garbage Collection

With types enforced on our system, we could possibly do away with the 1-bit object tags on every value in the system (as previously mentioned). But in doing so, we complicate garbage collection: the ability to distinguish between integers and references depends on those tags. In their absence, heap objects would need some kind of bitmap in the info word to know which fields are pointers. Then, because integers might be on the stack, we would have to limit the system to boxed integers only, so that every stored value on the Stack is a reference. Of course, some of the values on the Stack, like the stored bp values and stored PC, are not references; we would have to leverage knowledge of the Stack structure to properly walk the root set. In the face of these challenges, we choose to keep the 1-bit tags in the strict version for the sake of garbage collection. With them, we can blindly walk the stack or trace objects in the Heap and know which words are references.

Figure 5.3 shows all 11 states used for garbage collection in the strict Zarf machine. Because we have a single stack now, and no argument or evaluation registers, the root set is vastly simplified. This means we do not need "phases" as in the lazy machine, where we duplicate the same state machine for each item in the root set. The root set now is composed of just the Stack. There are two main phases to the garbage collection on the strict machine: one pass moves each object to the other semispace and updates the old objects with forwarding addresses; a second pass updates the addresses on the stack and in object fields with the new locations, according to the forwarding addresses from the first phase.

An INIT state configures the pointer registers that GC uses: gcptr, which points to the top of the Stack and is initialized with `sp`, and gcbase, which points to the next thing to collect. Because of the synchronous memory delays, gcbase is always pointing at the *next* thing to look at, while the read address register for the Stack holds the Stack address of what we're examining now. We also move the free pointer (fp) in this state to the start of the other semispace (whichever is not currently being used).

READSTACK is responsible for, as its name suggests, reading values from the Stack. We check the Stack output to see if it is a reference; if not, we update the read address and gcbase registers and loop. If it is a reference, we store the address of the first field of the object (base address + 1) in the gcaddr register and address the Heap with the address of the info word (the base address); then, we go to READINFO. If we enter READSTACK and we see that gcbase is greater than gcptr, we know we've finished reading the Stack and go to INIT2 for the second GC pass.

READINFO checks the info word output by the Heap to see if the object has already been forwarded. If it has, we can move on and return to READSTACK. If not, we check the info word to see how many fields this object has; we store it in a register named "gcN."

The info word is copied to Heap[fp] in the other semispace; we increment gcaddr and go to OVERWRITEINFO.

OVERWRITEINFO overwrites the info word we just saw with a forwarding address. One bit indicates that it is a forwarding address, and the remainder of the bits can be used to store the actual address. We now increment the free pointer fp. If the object we're looking at has no fields (gcN == 0), we set the Stack read address to gcbase and go to READSTACK to check the next stack value. Otherwise, we need to trace the object; for that, we go to READFIELD.

READFIELD loops over the fields of an object, tracking location with the gcaddr register and how many fields are left with gcN. We check the Heap output to see if it is a reference; if so, we store it on the Stack at gcptr (and increment the pointer). Regardless of whether the value is a reference or not, we copy it to fp in the other semispace. If this was the last field (gcN == 0), we read the Stack with gcbase and go to READSTACK. Otherwise, we increment gcaddr and loop.

INIT2 starts the second phase of GC. At this point, we've traced the entire live set and moved every object to the new semispace. However, the addresses on the Stack and in objects in the Heap still point to the old semispace. This second pass updates each of them. The second phase follows the same structure as the first phase, with same changes. We don't need an OVERWRITEINFO state, because we don't actually update any info word or relocate objects in the Heap. However, we need two extra states no in the first phase: one is needed to follow forwarding addresses to walk objects; the other is to read the Heap an additional time to find the forwarding address a given reference points to (so it can be overwritten with the new value).

INIT2 moves to READSTACK2, which reads the Stack for the second GC phase. As before, if gcbase is greather than gcptr, we've finished walking up the Stack, so we finish GC by going back to the top-level FETCH state. Otherwise, if the Stack output is not a reference, we increment and loop; if it is, we go to READINFO2.

CREADINFO2 looks at the output of the Heap, which necessarily must be a forwardingaddress — this is because the addresses on the Stack haven't been updated yet, so they must refer to objects in the "from" semispace, all of which we visited (if they were live). We take that forwarding address and overwrite the reference on the Stack with the new address. Then, we have to walk the object to update any references it has in its fields; to do that, we need to read the new address, so we configure the Heap read address and go to READNEWINFO2.

READNEWINFO2 behaves like READINFO in the first phase. It stores the number of fields in the object in the gcN register and increments gcaddr. If a 0 is being stored in gcN, the object has no fields, and we return to READSTACK2 to read the next value on the Stack. Otherwise, we go to READFIELD2 to read the first object field.

READFIELD2 checks the Heap output to see if it is a reference. If it is, we read the Heap with that address (which should point to an object in the "from" semispace, with a forwarding address in the info word) and go to OVERWRITEFIELD2. Otherwise, if this was the last field (gcN == 0), we read the Stack with gcbase and go back to READSTACK2 to see the next Stack value, incrementing gcbase. If it was not a reference and was not the last field, we increment gcaddr to see the next field and loop.

OVERWRITEFIELD2 overwrites the field we're looking at with an updated address, which we read from the Heap. Last cycle configured the Heap to read the old address, which should point to the info word containing the forwarding address. If this was the last field (gcN == 0), we read the Stack with gcbase and go back to READSTACK2, incrementing gcbase. Otherwise, we increment gcaddr to see the next field and loop back to READFIELD2.

Chapter 6

Software Reasoning

6.1 Introduction

One of the main premises of the Zarf architecture is that it improves upon the state of the art for assembly- and binary-level reasoning. To help substantiate that claim, here we briefly cover (anecdotally) the difficulties involved in proofs of correctness for a few assembly programs. After that introductory discussion, we introduce the main subject of the chapter: analysis of the software of an embedded medical device. The correctness proofs were performed using the Coq theorem prover [81]. The proofs use an earlier version of the machine semantics, which are not included in this thesis; however, the points made stand for the current versions.

For the **map** function, we compare the proof to a similar proof of correctness using a model of the MIPS ISA. This comparison shows that performing equivalent reasoning on MIPS — even when making vastly simplifying assumptions — is much more difficult. Figure 6.1 shows the lines of Coq code required for the correctness proofs for each of our small test programs.

Map

In Coq it is simple to prove theorems about the results of the predefined **map** library function. The goal of our proof is the following idea: if we can show that the execution of a **map** function written in the assembly language the Zarf ISA is equivalent to Coq's **map** function in an appropriate sense, then all existing proofs written in Coq about the values returned by the Coq **map** function become true of our assembly function. When translated from Coq, our main theorem statement is the following:

$$\forall l, \forall l', \text{ if } l \equiv l' \wedge \mathbf{f} \equiv \mathbf{f'} \wedge \text{other hypotheses } H,$$

$$\text{then Coq's } \mathbf{map}\ \mathbf{f}\ l \equiv \mathbf{interpret}(\text{ISA's } \mathbf{map'}\ \mathbf{f'}\ l')$$

Since the **map** function is defined in Coq by recursion on the structure of the list l, the proof of our main equivalence theorem proceeds by induction on the structure of the Coq list l. The convenience of Coq is evident in this proof: as our programs and proofs are written in the same language, we may *execute* our interpreter in the process of proving the main theorem about it. The interpreter executes a given function in a symbolic manner; this allows it to accept generic arguments and to reflect a function's execution on arbitrary inputs. The excerpt of the proof translated from Coq given below will demonstrate this ability. We will use the notation in which $\bigcirc'$ represents terms in the ISA's assembly, while $\bigcirc$ represents terms in Coq. We give the $l = x :: xs$ case:

Because $l \equiv l'$, by an assumption in H there exist x' and xs' such that $x \equiv x'$ and $xs \equiv xs'$ and $x :: xs \equiv \mathbf{cons}(x', xs')$, where **cons** is the ISA's :: operator.

We get from our induction hypothesis that:

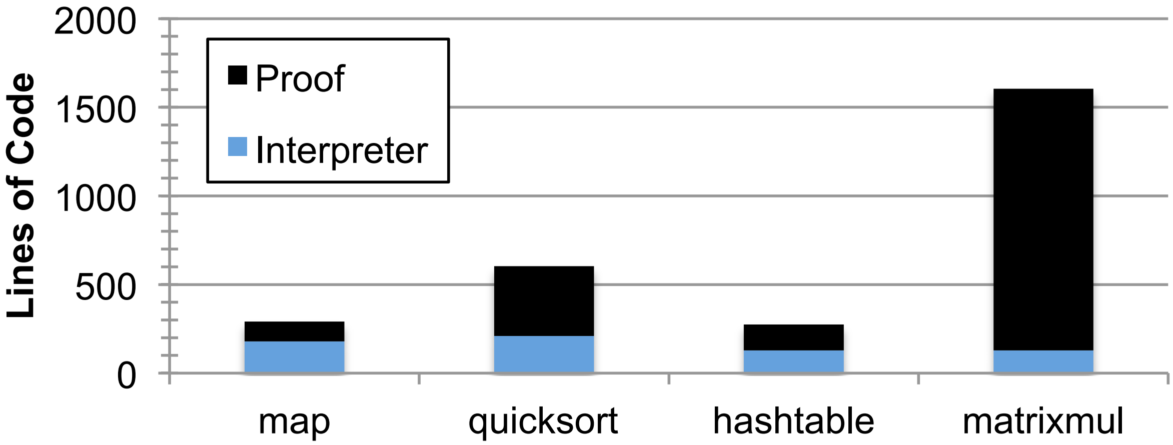


Figure 6.1: Lines of code for the initial machine-code level correctness proofs, formalized in the Coq proof assistant. The Interpreter portion reflects reusable machine modeling, while the Proof portion captures proof-specific code.

$\textbf{map f } xs \equiv \textbf{interpret}(\textbf{map}' \textbf{ f}' \ xs')$.

Running both versions of the function again we get:

$\textbf{map f } (x :: xs) \mapsto^* \textbf{f } x :: \textbf{map f } xs$

$\textbf{interpret}(\textbf{map}' \textbf{ f}' \textbf{ cons}(x', xs')) \mapsto^*$

$\textbf{cons}(\textbf{Zarf-function-call}(\textbf{f}', [x']),$

$\textbf{Zarf-function-call}(\textbf{map}', [\textbf{f}', xs']))$

$\textbf{f } x \equiv \textbf{Zarf-function-call}(\textbf{f}', [x'])$ by a few applications of our initial and local hypotheses.

By a hypothesis about the equivalence of :: and cons in the obvious sense, it remains to be shown that $\textbf{map}(\textbf{f}, xs) \equiv \textbf{Zarf-function-call}(\textbf{map}', [\textbf{f}', xs'])$. This follows from the induction hypothesis and a hypothesis that relates **interpret** and **Zarf-function-call** in an appropriate way. □

The interpreter for our ISA written in Coq simulates the vast majority of the features present in our machine during a single execution of a function. The size of the Coq development containing the interpreter and proofs is 293 lines.

Here we discuss our corresponding proof of correctness for MIPS, mentioning some of the many simplifying assumptions and shortcuts taken to avoid vastly increased proof complexity.

The theorem statement for this proof of equivalence is deceptively similar:

$$\forall l, \forall l', \text{ if } l \equiv l' \wedge \mathbf{f} \equiv \mathbf{f'} \wedge \text{other hypotheses } H, \text{ then}$$
$$\text{Coq's } \mathbf{map\ f}\ l \equiv \mathbf{interpret\text{-}MIPS}(\text{MIPS's } \mathbf{map'}\ \mathbf{f'}\ l')$$

Again we follow the proof style used above, but the hypotheses and interpreter functions are vastly different this time around. Our RISC interpreter must work at a much lower level than the one for our ISA. We must keep track of all values stored in registers, the contents of the stack, and the value of the program counter; this adds to the complexity of the formalization, and leaves more room for error.

The interpreter for the RISC machine is also quite incomplete. The amount of instructions and other features in, for example MIPS, outnumber those in our ISA by several orders of magnitude. For this reason, our interpreter is not comprehensive—it only contains the necessary functionality to allow the **map** function to evaluate properly. The following is a list of the features that our MIPS interpreter supports:

- Registers supported: `$sp`, `$s0`, `$s1`, `$a0`, `$a1`, `$v0`, `$ra`, `$zero`
- Instructions supported: `addi`, `sw`, `lw`, `jr`, `jal`, `move`, `beq`

Due to the presence of branch instructions, the this interpreter has the possibility of looping infinitely, even though we execute a given assembly function symbolically. We do not run into this issue in our ISA's interpreter, where it is impossible to jump to arbitrary locations in a given function. As Coq does not allow non-terminating functions, we overcome this limitation using a less-than-pretty solution—we add to the main MIPS interpreter function a "counter" variable and only allow the interpreter to execute for that many steps. This guarantees termination, while requiring the user to input a large enough starting value for the counter variable.

To be clear, non-termination is still possible in our ISA — but the formal semantics allow constructive proofs about functions so that an automated system like Coq can guarantee that a program terminates. In this case, for example, Coq can induce that every recursive call reduces the length of the list, and the list is finite.

The MIPS **map** function with which we perform our proofs updates its list in-place, while the Coq specification (and our ISA's map) return a *new* list, leaving the original values unchanged. Having MIPS match the specification would require adding memory semantics to the MIPS interpreter, which is no small formalization on its own, and so is ignored.

As our main theorem statement is quite similar to the one from the previous section, we still prove it in the same way: by induction on the Coq list *l*. However, the definition of **interpret-MIPS** is very different; it provides only a simplified view of the 9 instructions `map` requires (as opposed to a complete modeling of the roughly 85 core instructions in MIPS), and is complicated by state elements, like explicit references to the `v0` register in order to pass return values.

We further forego proof that the MIPS **map** function obeys stack discipline and restores the callee-saved registers. Additionally, to simplify modeling, we have to assume that the few instructions included cause no exceptions or side effects.

Even with these severe simplifying assumptions on the MIPS semantics, the Coq file containing the interpreter and proofs for the MIPS version of the **map** function *still* required 501 lines of code, a 1.7× increase — even without modeling memory, stack behavior, calling conventions, exceptions, branch delays, or even the majority of MIPS instructions. Ignoring virtually all complicating factors, the inherently stateful and imperative nature of the ISA still presented a larger hurdle to the modeling process.

Quicksort

While the example above is really intended to give some intuition for what it means to prove something about code written in these ISAs, in addition to the proof of the equivalences of the **map** functions, we also have proofs concerning the **quicksort** function for our ISA, along with **quicksort**'s counterpart function, **partition**. These two functions were programmed in our ISA's assembly, and the proofs, when taken together, constitute a proof that the these functions correspond to their Coq equivalents in terms of results. The proofs are more difficult due to the structure of the functions.

The proof of the **partition** function requires case analysis on the result between the pivot value and the head of the list, in the case where the list is not `[]`. We therefore have three separate proofs about partition: it must correspond to the Coq version (1) when the list is `[]`; (2) when the list is not `[]`, and the head of the list is less than the pivot; and (3) when the list is not `[]`, and the head of the list is greater than or equal to the pivot. Since the **quicksort** function makes recursive calls to arbitrary sublists, we must use strong induction on the length of the Coq list to prove our desired equivalence theorem. This requires knowledge that the call to **partition** returns two lists that are both smaller than the original list.

Given these difficulties, the size of the Coq file containing the interpreter and proofs is 609 lines. We did not successfully recreate these proofs for a MIPS version of **quicksort**.

Matrix Multiply and List Lookup

We also proved the equivalence of two more sets of functions for our ISA:

1. a naive, purely-functional equivalent of a hash table, and

2. a suite of several functions that together perform list-based matrix multiplication.

We have shown that both of these implementations are equivalent to the same functions implemented in Coq. Figure 6.1 summarizes the size of the proofs we developed.

The hash table required 280 lines of code, including the interpreter, which puts it on the same level as the **map** function. It required two functions—**lookup** and **insert**—both implemented using list operations.

The matrix multiplication algorithm is by far our most extensive example of assembly verification. Again including the interpreter for our ISA, this example required 1611 lines of Coq code to complete. We believe that this nontrivial algorithm demonstrates our ISA's ability to facilitate real-world verification at the assembly level.

Real Application: Implantable Cardioverter Defibrillator

In the remainder of this chapter we focus on the analysis of a proof-of-concept piece of embedded software that shows the utility and versatility of the Zarf platform. We have implemented a simple version of an ICD (implantable cardioverter defibrillator), an important embedded medical device that exists in the real world.

ICDs are small, battery-powered, embedded systems which are implanted in a patient's chest cavity and connect directly with the heart. For patients with arrhythmia and at risk for heart failure, an ICD is a potentially life-saving device. Currently, the primary use of ICDs is to detect dangerous arrhthymias (such as ventricular tachycardia, or VT) and administer pacing shocks (anti-tachycardia pacing, or ATP). These shocks help prevent the acceleration in heart rate leading to ventricular fibrillation, a form of cardiac arrest. Originally, their primary use was defibrillation to restart the heart during cardiac arrest; however, they are also able to detect ventricular tachycardia (a fast, irregular heartbeat which can be a precursor to heart failure) and administer pacing shocks, called antitachycardia pacing (ATP), to restore a normal rhythm and prevent cardiac arrest from ever occurring. Research documenting the success of pacing shocks even in cases of ventricular fibrillation (a form of cardiac arrest) have made ATP the primary goal of ICDs [82], with defibrillation available as a backup. Because of this, our ICD application just implements the antitachycardia pacing.

From 1990 to 2000, over 200,000 ICDs and pacemakers were recalled due to software issues [83]. Between 2001 and 2015, over 150,000 implanted medical devices were recalled by the FDA because of life-threatening software bugs [84]. However, ICDs are credited with saving thousands of lives; for patients who have survived life-threatening arrhthymia, ICDs decrease mortality rates by 20-30% over medication [85, 86, 87]. Currently, around 10,000 new patients have an ICD implanted each month [88], and around 800,000 people are living with ICDs [89].

The core of our ICD is an embedded, real-time ECG (electrocardiogram) algorithm that performs QRS[1] detection on raw electrocardiogram data to determine the timing between heartbeats. We work off of an established real-time QRS detection algorithm [90], which has seen wide use and been the subject of studies examining its performance and efficacy [91]. An open-source update of several versions of the algorithm [92] is available; we use the results of this open-source work as the basis of our algorithm's specification as well as the C alternative. After the ECG algorithm detects the pacing between heartbeats, the ATP function checks for signs of ventricular tachycardia and, if found, administers a series of pacing shocks. We implement the VT test and ATP treatment published in [93].

Our software is architected using a very small microkernel managing three cooperative coroutines [94, 95]. Separating responsibilities into coroutines aids in our software analysis. The microkernel is responsible for scheudling the coroutines in a round-robin fashion and handling communication between them. The main coroutine handles the core ICD functionality, while another handles all I/O — both reading values from the heart and potentially delivering treatment shocks. A third coroutine is purely for diagnostic purposes, logging how often

[1] The "QRS complex" is made up of the rapid sequence of Q, R, and S waves corresponding to the depolarization of the left and right ventricles of the heart, forming the distinctive peak in an ECG.

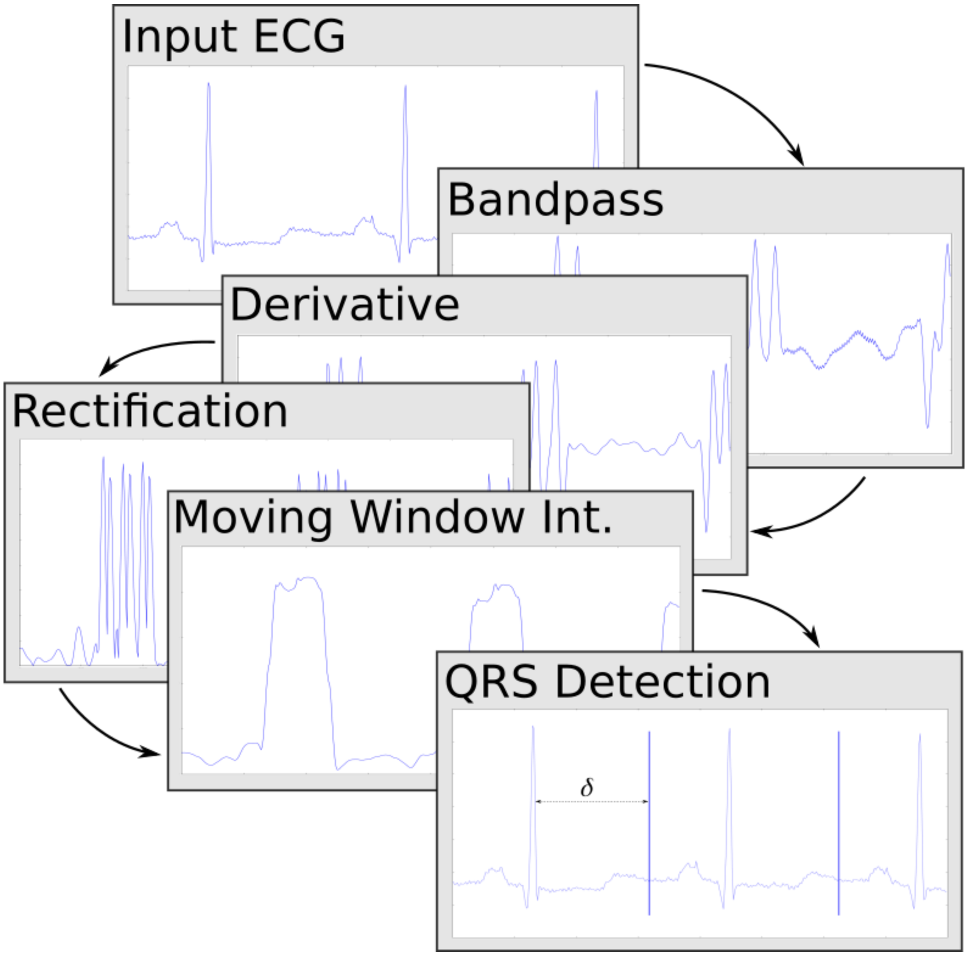


Figure 6.2: The series of filter passes that make up the real-time QRS detection algorithm, which is the heart of the ECG. The input raw ECG data undergoes bandpass filtering, then a derivative pass, followed by rectification. Peaks are combined with a moving-window integration, and finally the downward slopes can be used to detect beats. The final window shows the detected beats (vertical lines) superimposed on the input data. The beats are delayed by δ, the processing time of the filter passes and detection algorithm.

treatment has occured.

The I/O coroutine is passed the output of the previous iteration of the ICD coroutine. A hardware timer is used to ensure that I/O events occur at the correct frequency. When the correct time has elapsed (5 ms), the I/O coroutine outputs the given value and reads the next input value. It yields this value to the microkernel.

This input is then passed through to the ICD coroutine, which implements a series of filter passes to detect the spacing between QRS complexes (Figure 6.2 illustrates the ECG filter passes). If 18 of the last 24 beats had periods less than 360 ms (corresponding to a heart rate greater than 167 bpm), the ICD coroutine moves into a treatment-administering state, where it outputs three sequences of eight pulses at 88% of the current heartrate, with a 20 ms decrement between sequences. This is designed to prevent continued acceleration and restore a safe rhythm.

The monitoring software receives the output of the ICD coroutine each cycle. A command can be given on the diagnostic input channel for the software to output the number of times treatment has occurred.

I/O events occur at a fixed frequency of 200 Hz. Timing analysis in section 6.4 confirms that, after an input event, the entire cycle of each coroutine running and yielding, including garbage collection, is able to conclude well within the 5 ms window, meaning that the entire system is always able to meet its real-time deadline.

We separate the verification of the embedded ICD application into three parts: verification of the correctness of the ICD coroutine, a proof of non-interference between the trusted ICD coroutine and untrusted code outside of it, and a timing analysis to show that the assembly meets timing requirements in the worst case.

6.2 Proving Correctness

We first implement a high-level version of the application's critical algorithms (the ECG filters and ATP procedure) in Gallina, the specification language of the Coq theorem prover [81], using this version as our specification of functional correctness. This specification operates on streams — a data type that represents an infinite list — by taking a stream as input and transforming it into an output stream. By sticking to a high-level, abstract specification, we can be more confident that we have specified the algorithm correctly. An ICD implementation cannot operate on streams, as all data is not immediately available; instead, it takes a single value, yields a single value, and then repeats the process.

The form of the correctness proof is by refinement: first, we create a Coq implementation of the ICD algorithm that is "lower-level" than the Coq specification. This lower-level implementation operates on machine values rather than streams, isolates function applications to let expressions, and avoids the use of "if-then-else" expressions, among other trivially-resolved differences. We then create an extractor that converts this lower-level Coq code directly into executable Zarf functional assembly code (see Figure 6.3). If, for all possible input streams, we can prove that the output stream produced by the high-level Coq specification is the same sequence of values produced by the lower-level implementation, we can conclude that the program we run on Zarf is faithful to the high-level Coq specification. This proof of equivalence between the two Coq implementations is done by induction over the program, showing that if output has matched up to point N, and the computation of value N is equivalent, then value $N + 1$ will be equivalent as well. As compared to extracting for an imperative architecture, we avoid needing to compile functional operations to an imperative ISA and do not require a large software runtime — or any software runtime at all. The translation simply replaces Coq keywords with Zarf assembly keywords.

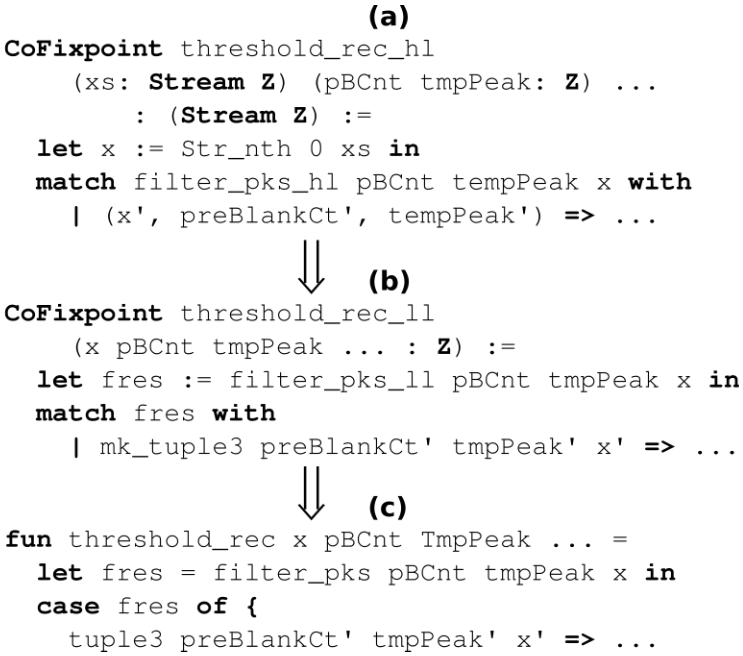
The full proofs of correctness of the assembly-level critical ECG and ATP functions take under 2,500 lines of Coq. The implementations are converted line-for-line into Zarf assembly code, which is combined with assembly for the microkernel and other coroutines.

In total, the Trusted Code Base for the correctness proof includes: the hardware, the Coq proof assistant, and the small extractor that converts the low-level Coq code into Zarf functional assembly code. All other code is untrusted and may be incorrect, and the proof will still hold. The high-level ISA and clearly-defined semantics make this very small TCB possible, allowing the exclusion of language runtimes, compilers, and associated tooling that is frequently present in the TCB in verification efforts.

In total, the Trusted Code Base for the correctness proof includes: the hardware, Coq, the small program extractor (which just does syntax replacement), and the assembler. All other code is untrusted and may be incorrect, and the proof will still hold. The high-level ISA and clearly defined semantics make this very small TCB possible, allowing the exclusion of language runtimes, compilers, and associated tooling that is frequently present in the TCB in verification efforts.

An alternative method of proving correctness, with a smaller TCB, is to take as input as binary program and reason that it is equivalent to the specification. This can be more challenging, because it requires reasoning both about a low-level series of bits and a high-level Coq specification, but it does not rely on trusting a program extractor or, critically, the assembler.

To verify a program with this method, we require an interpreter in Coq that executes the instruction semantics, letting the theorem prover see what the result of each instruction is. This is made easier on Zarf by the compact and precise semantics, which make the interpreter rather small. However, we do require a small script to disassemble the binary program, which in theory could be implemented in Coq and verified (and therefore not in the TCB), but would be quite an endeavor. This script prints each instruction word as a Coq data structure. The correctness proof is very similar to the first method: we show that, for any possible input stream, the output stream produced by the Coq specification is the same as the sequence of values



```
                         (a)
CoFixpoint threshold_rec_hl
    (xs: Stream Z) (pBCnt tmpPeak: Z) ...
        : (Stream Z) :=
  let x := Str_nth 0 xs in
  match filter_pks_hl pBCnt tempPeak x with
    | (x', preBlankCt', tempPeak') => ...
                       ⇓  (b)
CoFixpoint threshold_rec_ll
    (x pBCnt tmpPeak ... : Z) :=
  let fres := filter_pks_ll pBCnt tmpPeak x in
  match fres with
    | mk_tuple3 preBlankCt' tmpPeak' x' => ...
                       ⇓  (c)
fun threshold_rec x pBCnt TmpPeak ... =
  let fres = filter_pks pBCnt tmpPeak x in
  case fres of {
    tuple3 preBlankCt' tmpPeak' x' => ...
```

Figure 6.3: Extraction of verified application components, summarized for a small excerpt. **(a)** The high-level Coq specification is written to operate on `Streams` (infinite lists); values are pulled from the front of the stream. **(b)** An intermediate version is written in Coq which operates on integers instead of streams, and unfolds nested operations so each function call and arithmetic operation takes one line. This intermediate version is proven equivalent in Coq to the high-level specification — meaning that repeated recursive application of (b) will always output the same sequence of values as (a). **(c)** A simple extractor just replaces the keywords in (b) to produce valid assembly code that can run on Zarf. (Refer to Figure 3.5 to see how assembly code maps directly to the binary, which the hardware actually executes.

produced by the Zarf binary program. As before, this requires induction over the program, showing that if values matched up to point N, and the computation of value N is equivalent, then the value $N + 1$ will be equivalent as well.

The main challenge in this style of proof is bridging the semantic gap between the high-level Coq Streams and the binary program. Each step in the program yields both a value and a collection of state that is used in the computation of the next value; this forms a mismatch with the specification, where all values are computed at once and state is not carried in the same way. To show equivalence, we would need to write an intermediate version of the program in Coq that operates on values, not streams, prove its equivalence to the original specification, and then prove equivalence between the intermediate version and the binary program.

6.3 Integrity Types

Because the ICD coroutine has been proven correct (Section 6.2), we treat its output as trusted. This output must then travel through the rest of the cooperative microkernel until it reaches the outside world via the I/O coroutine's `putint` primitive. In order to guarantee the integrity of this data (meaning it is never corrupted nor influenced by less-trusted data), we rely on a proof of non-interference. Non-interference means that "values of variables at a given security level $\ell \in \mathcal{L}$ can only influence variables at any security level that is greater than or equal to ℓ in the security lattice $\mathcal{L}$" [96]. In a standard security lattice, **L** (low-security) $\sqsubseteq$ **H** (high-security), meaning that high-security data does not flow to (or affect) low-security output. In our application, however, we are concerned with integrity; our lattice is composed of two labels, **T** (trusted) and **U** (untrusted), organized such that **T** $\sqsubseteq$ **U**. Therefore, our integrity non-interference property is that untrusted values cannot affect trusted values [14].

To prove this about the Zarf program, we create a simple integrity type system that provides a set of typing rules to determine and verify the type of each expression, function, and constructor in a program. After providing trust-level annotations in a few places and constraining the normal Zarf semantics slightly to make type-checking much easier, we can run a type-checker over the resulting Zarf code to know whether it maintains data integrity. We extend the original Zarf syntax to allow for these type annotations, as follows:

$$\begin{aligned}
\ell, \textsc{pc} \in \mathit{Label} &::= \textsf{\textbf{T}} \mid \textsf{\textbf{U}} \\
\tau \in \mathit{Type} &::= \textsf{\textbf{num}}^{\ell} \mid (cn, \vec{\tau}) \mid (\vec{\tau} \to \tau) \\
\mathit{func} \in \mathit{Function} &::= \textsf{\textbf{fun}}\, \mathit{fn}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ : \tau\ \textsf{\textbf{=}}\ e \\
\mathit{cons} \in \mathit{Constructor} &::= \textsf{\textbf{con}}\ cn\ x_1 : \tau_1, \ldots, x_n : \tau_n
\end{aligned}$$

Specifically, following the spirit of Abadi et al. [9] and Simonet [97], types are inductively defined as either labeled numbers, or functions and constructors composed of other types. Our proof of soundness on this type system follows the approach done in work by Volpano et al. [10]. We show that if an expression e has some specific type τ and evaluates to some value v, then changing any value whose type is less-trusted than e's type results in e evaluating to the same value v; thus, we show that arbitrarily changing untrusted data cannot affect trusted data. We prove soundness case-wise over the three types of expressions in our language, combining our evaluation semantics with our security typing rules.

The full integrity typing rules, and a proof of soundness of those rules, can be found in Appendix B.

6.4 Bounding Execution Time

With a knowledge of how the Zarf hardware executes each instruction, we create worst-case timing bounds for each operation. In general, in a functional setting, unbounded recursion makes it impossible to statically predict execution time of routines. Though our application uses infinite recursion to loop indefinitely, the goal is to show that each iteration of the loop meets the real-time deadline; within that loop, each coroutine is executed only once, and no functions call into themselves. This allows us to compute a total worst-case execution time for the sum of all the instructions by extracting the worst-case route through the hardware state machine to execute each possible operation. For example, applying two arguments to a primitive ALU function and evaluating it has a maximum runtime of 30 cycles — this includes the overhead of constructing an object in memory for the call, performing a function call, fetching the values of the operands, performing the operation, marking the reference as "evaluated" and saving the result, etc. In an average case, only a fraction of the possible overhead will actually be invoked.

Hardware garbage collection is a complicating factor on timing. GC can be configured to run at specific intervals or when memory usage reaches a certain limit; for our application, to guarantee real-time execution, the microkernel calls a hardware function to invoke the garbage collector once each iteration. To reason about how long the garbage collection takes, we bound the worst-case memory usage of a single iteration of the application loop. The hardware implements a semispace-based trace collector, so collection time is based on the live set, not how much memory was used in all. For the trace-collector state machine, each live object takes N+4 cycles to copy (for N memory words in the object), and it takes 2 cycles to check a reference to see if it's already been collected. We bound the worst-case by conservatively assuming that all the memory that is allocated for one loop through the application might be simultaneously live at collection time, and that every argument in each function object may be a reference which the collector will have to spend 2 cycles checking.

From the static analysis, we determine that the worst execution of the entire loop is 4,686 cycles, not including garbage collection. Garbage collection is bounded by a worst-case of 4,379 cycles, making a total of 9,065 cycles to run one iteration of system — or 181.3 μs on our FPGA-synthesized prototype running at 50 MHz, falling well-within the real-time deadline of 5 ms.

Chapter 7

Implementing a Hardware Typechecker

7.1 Introduction

The demand for more connectivity and richer interactions in every day objects means that everything from light bulbs to thermostats now contain general-purpose microprocessors for carrying out fairly straightforward and low-performance tasks. Left unanalyzed, these systems and their associated software stacks can be expected to hold a seemingly endless collection of opportunities for attack. Static analysis provides powerful tools to those wishing to understand or limit the set of behaviors some software might exhibit. By facilitating sound reasoning over the set of all possible executions, this type of analysis can identify important classes of behavior and prevent them from ever happening. If embedded system developers simply *never* released software that failed, such that those well-analyzed applications were the *only* things to ever execute on platforms under our control, many of the bugs and vulnerabilities that plague our life would be eliminated. Unfortunately, realizing this in practice has proven incredibly hard due to pressure to market, pressure to reduce cost, and the delayed and stochastic cost associated with vulnerabilities and bugs.

While larger software companies might be more trusted to rigorously verify their software releases, the embedded systems market has a long and heavy tail of providers with a much wider distribution of expertise and resources at their disposal. When we bring an embedded device into our home or business, how can we have confidence that the software running there (which depends on chains of control well outside our ability to observe) is "above the bar" for us? Seemingly innocuous issues, for example passing a string instead of an integer, can open the door for an attacker to gain root privileges and serve as a base for other attacks (exactly this happened already in a class of WiFi routers [98]). Similar attacks targeting embedded devices and firmware updates have succeeded on everything from printers [99] to thermostats [100].

Such a machine would reject any attempt to load it with code that fails to meet the specified "bar," independent of who wrote it, who signed it, how it was managed, or where the software came from. The trust one could put in aspects of execution on such a processor could be independent of measurement, attestation, or other active third-party evaluation. By doing the checks in hardware, we can make them intrinsic to the device's functionality: the checks will be fully live right from power-up; the checks will require no dependency on other software on the system functioning correctly (zero TCB); and if properly designed, they will be directly wired into the operation of the system, making them provably impossible to bypass.

7.2 Static Semantics

Figure 7.1 defines the static semantics for typed Zarf. As with the explanations of Figures 3.2 and 3.3 in Section 3.3, a reader more familiar with the mathematical notation used can skip the following explanation; similarly, a reader that does not need an in-depth explanation of the typing rules should also feel free to skip to Section 7.3.

The following are used in Figure 7.1 and the following sections. `map`, `filter`, and `concatMap` refer to their standard definitions. `delete` removes the first instance of an element from a list. We use the notation $\Gamma[x \mapsto \tau]$ to represent the creation of an updated map Γ where the new entry $x \mapsto \tau$ has been added to the old map; similarly, $\Gamma[\vec{x} \mapsto \vec{\tau}]$ denotes mapping the first variable in $\vec{x}$ to the first type in $\vec{\tau}$, etc. for all point-wise pairs. Unless otherwise stated, the lengths of both lists must be the same. We use the notation $\mathcal{O}(\cdot)$ to indicate an "Option" type: essentially a set guaranteed to contain zero or one values. We use the symbol $\bullet$ to represent the absence of a value where an "Option" type is required. We differentiate between metavariables with subscripts that can be a combination of letters or numbers; for example, τ and τ_1 represent distinct metavariables denoting types.

A static semantics itself isn't formally worthwhile unless the system is proved to be sound. For us, soundness can be reduced to the properties of "progress" and "preservation." Progress means that, if a program is well-typed, at every step of computation, there is always another valid step to be taken. Preservation says that, if a program is well-typed, then every step taken will result in another well-typed term. Together, they mean that a program can never "get stuck," i.e., hit a run-time error with regards to the semantics. Note that the semantics specifically excludes finite memory, so running out of memory remains the final runtime error that can occur. Other than that, our proofs of progress and preservation ensure that a well-typed program will never encounter an error. The proofs of progress and preservation rely on the details of the following subsections, and can be found in Appendix A.

7.2.1 Static Semantic Domains

The following domains are used in the static semantics.

$$\Gamma \in \mathit{Environment} = \mathit{Variable} \rightarrow \mathit{Type} \qquad C \in \mathit{ConstraintSet} = \mathcal{P}(\mathit{Type} \times \mathit{Type})$$

$$\sigma \in \mathit{Substitution} = \mathit{TypeVariable} \rightarrow \mathit{Type} \qquad b \in \mathit{Bool} = \textbf{true} + \textbf{false}$$

Γ (gamma) is an environment, which maps a variable to its type. C is a set of constraints; each constraint in the set is a pair of types which are constrained to be equal. A constraint set can be consistent or inconsistent; each will be checked in the course of type checking a program for consistency (any contradictions within a given constraint set result in type checking failure for the entire program). σ (sigma) is a substitution, which maps type variables to types. Type variables allow for polymorphic types by declaring something like "`List a`", where `a` is a type variable; from this one declaration, we can instantiate `List int`, `List char`, `List String`, etc. throughout the program, without needing new type declarations. As a result of the constraint unification procedure, we will be left with a σ that maps all of the type variables used to the concrete instantiated types. Finally, b is a Boolean value, either true or false.

7.2.2 Static Semantic Rules

There are two forms of typing derivations in Figure 7.1. The first is for typing top-level function definitions ($\vdash \mathit{func} : \tau$). The second is for typing expressions ($\Gamma \vdash e : \tau$).

The first rule, FUNC-RET, is used for a function of no arguments. Some function fn has a function body is an expression e with an annotated return type τ_r. We employ the `makeRigid` helper function on the annotated return type to replace all the type variables with concrete (rigid) types; we call the new rigid return type τ_{r1}. This process is necessary to avoid confusion of type variables as we check the function. In a given execution of the function, the type for each type variable is concrete and cannot change; it can only change across executions. Making the types concrete for checking solves two problems: it ensures the function is consistent for that type (by making sure there are no type inequalities), and it ensures the polymorphic function is actually universal (by making sure there is no coercion of the rigid type variable to another rigid type). For example, if a function is supposed to be polymorphic in its arguments, but uses the integer `add` function on them, then it actually only works for integers. Making the

Functions $\boxed{\vdash \mathit{func} : \tau}$

$$\frac{\tau_{r1} = \mathtt{makeRigid}(\tau_r) \qquad \vdash e : \tau_{r2} \qquad \mathtt{principalType}([\tau_{r1}, \tau_{r2}]) = \tau_{r1}}{\vdash \textbf{fun}\ \mathit{fn}\ []\ \tau_r\ \ \textbf{=}\ e : \tau_r}\ (\textsc{func-ret})$$

$$\frac{\begin{array}{c}(\vec{\tau}_{p1} \ \rightarrow\ \tau_{r1}) \ =\ \mathtt{makeRigid}(\vec{\tau}_p \ \rightarrow\ \tau_r) \\ \vec{x} \mapsto \vec{\tau}_{p1} \vdash e : \tau_{r2} \qquad \mathtt{principalType}([\tau_{r1}, \tau_{r2}]) = \tau_{r1}\end{array}}{\vdash \textbf{fun}\ \mathit{fn}\ \overrightarrow{x : \tau_p}\ \tau_r\ \ \textbf{=}\ e : \vec{\tau}_p \rightarrow \tau_r}\ (\textsc{func-params})$$

Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\begin{array}{c}\mathtt{idTy}(\Gamma, \mathit{id}) = \tau_i \qquad \alpha = \mathtt{freshGenTV} \qquad \Gamma_1 = \Gamma[x_1 \mapsto \alpha] \\ \mathtt{map}(\mathtt{argTy}(\Gamma_1), \overrightarrow{\mathit{arg}}) \ =\ \vec{\tau}_a \quad \mathtt{applyType}(\tau_i, \vec{\tau}_a, [], \alpha) \ =\ \tau_1 \quad \Gamma[x_1 \ \mapsto\ \tau_1] \ \vdash\ e \ :\ \tau\end{array}}{\Gamma \vdash \textbf{let}\ x_1\ \textbf{=}\ \mathit{id}\ \overrightarrow{\mathit{arg}}\ \textbf{in}\ e : \tau}\ (\textsc{let-var})$$

$$\frac{\Gamma[x \mapsto \textbf{Int}] \vdash e : \tau}{\Gamma \vdash \textbf{let}\ x\ \textbf{=}\ n\ \textbf{in}\ e : \tau}\ (\textsc{let-int})$$

$$\frac{\begin{array}{c}\Gamma(x) = \mathit{dt} \qquad \overrightarrow{\mathit{cons}} = \mathtt{getConstructors}(\mathit{dt}) \qquad \mathtt{allConsPres}(\overrightarrow{\mathit{cons}}, \overrightarrow{\mathit{br}}) = \textbf{true} \\ \vec{\tau} \ =\ \mathtt{branchTypes}(\Gamma, \overrightarrow{\mathit{br}}, \overrightarrow{\mathit{cons}}) \qquad \mathtt{principalType}(\vec{\tau}) \ =\ \tau\end{array}}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{\mathit{br}} : \tau}\ (\textsc{case-con})$$

$$\frac{\begin{array}{c}\Gamma(x) = \mathit{dt} \qquad \overrightarrow{\mathit{cons}} = \mathtt{getConstructors}(\mathit{dt}) \qquad \Gamma \vdash e : \tau_e \\ \vec{\tau} = \mathtt{branchTypes}(\Gamma, \overrightarrow{\mathit{br}}, \overrightarrow{\mathit{cons}}) \qquad \mathtt{principalType}(\tau_e :: \vec{\tau}) = \tau\end{array}}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e : \tau}\ (\textsc{case-con-else})$$

$$\frac{\Gamma(x) = \textbf{Int} \quad \overrightarrow{(n_i \Rightarrow e_i)} \in \overrightarrow{\mathit{br}} \quad \Gamma \vdash e_i : \tau_i \quad \Gamma \vdash e : \tau_e \quad \mathtt{principalType}(\tau_e :: \vec{\tau}_i) = \tau}{\Gamma \vdash \textbf{case}\ x\ \textbf{of}\ \overrightarrow{\mathit{br}}\ \textbf{else}\ e : \tau}\ (\textsc{case-int})$$

$$\frac{\mathtt{argTy}(\Gamma, \mathit{arg}) = \tau}{\Gamma \vdash \textbf{result}\ \mathit{arg} : \tau}\ (\textsc{result})$$

Figure 7.1: Static semantics (typing rules) of the typed Zarf ISA.

type variables rigid means that they cannot be unified with the type `int`, causing a type error.

Next, we use the type checking rules to reduce the body expression e to a type τ_{r2}. Our goal is to check that the annotated type with rigid type variables τ_{r1} matches this returned type τ_{r2}. However, we have to take into account type variables in the return type. To check this, we employ the `principalType` helper function on the two types. It will unify them and substitute rigid type for type variables as necessary; then, we require that this unified type is equal to τ_{r1}[1]. τ_{r1} was already made rigid, so this just makes sure that τ_{r2} can be made rigid to match.

The second function rule is FUNC-PARAMS, used when a function takes arguments. Some function fn has a series of arguments $\vec{x}$ that have types $\vec{\tau_p}$ and an annotated return type τ_r. The body e is thus of the type $\vec{\tau_p} \mapsto \tau_r$. The only difference from the previous rule is that now we have to use the types of our arguments to help reduce the body to its return type: instead of $\vdash e : \tau_{r2}$, we write $\vec{x} \mapsto \vec{\tau}_{p1} \vdash e : \tau_{r2}$, creating an environment where the args map to types for use in the other rules to reduce e to a type τ_{r2}.

The remaining rules deal with how to reduce and check expressions. First is LET-VAR, which is for `let` expressions that require a function application (as opposed to a `let` that merely assigns an integer to a variable). `idTy` is used to look up id, the function being applied on, in Γ, the environment; it also replaces the type variables in the returned type expression with fresh ones to prevent accidental clashing of type variables. This sort of clashing comes up in several places; in essence, the issue is that something like `Lisa a` can be used multiple times in a function, but the `a` is different every time. Similarly, the `a` type variables from `List a` can be different from that in `Tuple a b`, so we need to be careful not to assume they are the same type.

Next, we get a fresh type variable α from `freshGenTV` to represent the return type of the function; the name in the `let` expression, x_1, is bound to α in the new environment Γ_1. This helps support forward references for lazy versions of the system. The type variable stands

[1] If the two types unified successfully, this will always be true, as τ_{r1} was already made rigid.

in for what the function actually returns, which gives us a handle to use in case it shows up elsewhere; e.g., in building an infinite list, or a node of a tree that points back to itself. In those cases, the type variable lets us add constraints that apply both to the particular argument and the return type, ensuring the entire `let` expression is valid.

This new Γ_1 is used to map `argTy` over the supplied list of argument names $\vec{arg}$. This will return a type for each argument in a vector $\vec{\tau_a}$. Finally, we can invoke `applyType` on the function type τ_i, the vector of argument types $\vec{\tau_a}$, and our return type variable α. An empty set [] is passed in to initialize the constraint set. `applyType` does the main work of type checking the function application; it will match up each argument with the corresponding expected type and generate a new constraint for each pair, along with a constraint that the return type variable is equal to the return type. Then, once it has processed the entire function application, the constraints are checked for consistency to ensure there are no type errors. Extra care has to be taken with "unfolding" function types that are nested inside other types; this and other concerns makes `applyType` somewhat complex.

The final premise states that mapping x_1 to τ_1 in our environment Γ will allow us, via application of the static semantic rules, to reduce the remaining body e to a type τ.

The rule LET-INT is relatively simple. It is used when a `let` instruction is binding an integer to a variable. We simply create a new binding in the environment Γ that x is of type **Int**, and this allows us to apply the other rules in reducing e, the remaining body, to τ.

There are three rules for dealing with `case` expressions. One for when a function cases on an integer, one for casing on a constructor without an else branch, and one for casing on a constructor with an else branch. CASE-CON handles this last situation. We are casing on a variable x and have a series of branches $\vec{br}$, with an annotated function return type τ. In addition to making sure the body of each branch is consistent, we need to make sure that each branch returns a valid type that matches the function return type[2]; finally, we the type system

[2] As stated previously, `case` instruction are non-reconvergent, so each branch must terminate in a `result`

must also ensure that, since there is no else branch, every possible constructor of the scrutinee's type is present in at least one branch.

First, we look up the type of the scrutinee in gamma and assign it to dt: $\Gamma(x) = dt$. We make a list $\vec{cons}$ that holds all the constructors of the given type. We use the helper function `allConsPres` on the constructor list and the list of branches to make sure that each constructor in the list is present in at least one branch head (if we had an else branch, this constraint would be relaxed, as seen in the next rule). `branchTypes` is going to iterate over the branches, reducing each to a type by 1) mapping the constructor fields in the branch head to the named variables, and 2) checking the body of the branch. This will yield a list $\vec{\tau}$ of types containing one type for each branch. Finally, we feed this list to `principalType`, which will ensure they all match and can be constrained to the common type τ.

In the situation where a `case` expression does have an else branch (and is casing on a constructor, not an integer), we apply the CASE-CON-ELSE rule. This rule works much the same as the previous one, except for the handling of the else branch. We now have an expression e representing the body of the else branch, so we explicitly reduce that to a type τ_e using the environment Γ, as seen in $\Gamma \vdash e : \tau_e$. We also remove the requirement that each constructor is present, because we have an else branch to catch the case where no branch matches.

CASE-INT is used when the function cases on a integer variable (here, an else branch is always required). We see the premise that our scrutinee x is an integer: $\Gamma(x) = \textbf{Int}$. We "unfold" the branches with the phrase $(\overrightarrow{n_i \Rightarrow e_i}) \in \overrightarrow{br}$; each branch has a natural number to match against n_i and a corresponding body expression e_i. Using the environment Γ, we can reduce each branch expression to a type τ_i, and the else branch expression to a type τ_e. We again employ the `principalType` helper function to make sure all the types match and reduce to the annotated type τ.

The final rule handles `result` instructions. We use the helper function `argTy` to find the

instruction.

type of the returned variable *arg*, and make sure that it matches the annotated function return type τ.[3]

The next subsection outlines in words and defines formally what each helper function does.

7.2.3 Static Semantic Helper Functions

`allConsPres`

`allConsPres` checks if all the constructors have a matching branch. Note that it is *not* ill-typed for the pattern branch to provide more binders than the constructor has fields; it's only ill-typed if those extraneous binders are *used* in the branch body. In the helper `branchTypes`, when the added binders are mapped to their types, it is implied that the mapping is guaranteed to be no larger than the number of types in the corresponding constructor. If an extraneous binder is used in the branch body, it will not appear in the environment because it wasn't added, and the type system will determine the branch body to be ill-typed, as expected. This relaxation is needed here because the machine replaces all the binder references with a field index; if the field is never used in the body of the branch, then it would not appear in the binary, and therefore we cannot try to over-constrain it here because the binary would not produce an error during execution.

Also note that there is no requirement that $\overrightarrow{br}$ contain exactly the same number of pattern branches as the number of constructors; it is okay for there to be more branches that constructors for the given datatype. In the helper function `branchTypes`, we ensure that each pattern branch matches a constructor in the cased-on datatype; therefore, the implication is that having duplicate pattern branches is acceptable, as long as they match a constructor of the scrutinized datatype.

[3] We use a helper function, rather than just looking in Γ, because the variable could be an integer. Then, we know the type is simply **Int**, and the name is not present in Γ.

$$\texttt{allConsPres} \in \overrightarrow{Constructor} \times \overrightarrow{Branch} \rightarrow Bool$$

$$\texttt{allConsPres}([], \_) = \textbf{true}$$

$$\texttt{allConsPres}((\textbf{con}\ cn\ \vec{\tau}) :: \overrightarrow{cons}, \overrightarrow{br}) = (cn\ \_ \Rightarrow \_) \in \overrightarrow{br} \wedge \texttt{allConsPres}(\overrightarrow{cons}, \overrightarrow{br})$$

`applyHelper`

`applyHelper` recursively iterates through a list of parameter and argument types, generating the constraint that the current parameter must equal the current argument. As function types are uncurried in this formalism, it is used to determine when to continue applying extra arguments to the current function's return type, which must also be a function type.

$$\texttt{applyHelper} \in FunctionType \times \overrightarrow{Type} \times ConstraintSet \times TypeVariable \rightarrow Type$$

$$\texttt{applyHelper}(\tau_p :: \vec{\tau}_p \rightarrow \tau_r, \tau_a :: \vec{\tau}_a, C_1, \alpha) = \texttt{applyType}(\tau_f, \vec{\tau}_a, C_2, \alpha)$$

where

$$C_2 = \{\tau_p = \tau_a\} \cup C_1$$

$$\tau_f = \begin{cases} \tau_r & \text{if } \vec{\tau}_p = [] \\ \vec{\tau}_p \rightarrow \tau_r & \text{otherwise} \end{cases}$$

`applyType`

`applyType` describes application of a type to a (possibly empty) list of argument types. When applying a function type to a non-empty list of argument types, it calls `applyHelper`, which performs the step of constraint generation so that this function can verify that the application is valid via unification. This and the associated helper functions are needed because of the uncurried presentation of functions in the abstract syntax, which reflects their representation in the machine more closely. Note that it is considered an error to try to apply a non-function type to a non-empty list of argument types.

The argument α is used to denote the type variable that the variable in the current let binding was bound to before calling this function. This is necessary to handle the case where an argument's type is unknown when application begins because it is being recursively defined. For example, in `let ones = Cons 1 ones in ...`, `ones` is being recursively defined and therefore will not have a type fully determined until after application; upon a call to `applyHelper`, the parameter `ones` will have type α (where α is fresh) in the environment. After the process of constraint generation and unification is completed during this application, if α is found in the substitution, that means it was used as an argument during application and we can then check that its use corresponds correctly with the result of the entire application.

$$\texttt{applyType} \in \mathit{Type} \times \overrightarrow{\mathit{Type}} \times \mathit{ConstraintSet} \times \mathit{TypeVariable} \rightarrow \mathit{Type}$$

$$\texttt{applyType}(\tau_1, \vec{\tau}_a, C, \alpha) = \begin{cases} \tau_2 & \text{if } \vec{\tau}_a = [] \\ \texttt{applyHelper}(\vec{\tau}_p \rightarrow \tau_r, \vec{\tau}_a, C, \alpha) & \text{if } \vec{\tau}_a \neq [] \wedge \tau_1 = \vec{\tau}_p \rightarrow \tau_r \end{cases}$$

where

$$\sigma = \texttt{unify}(C)$$

$$\tau_2 = \texttt{substitute}(\sigma, \tau_1)$$

$$\textbf{true} = (\alpha \notin \mathit{dom}(\sigma)) \vee ((\alpha \mapsto \tau_\alpha) \in \sigma \wedge \texttt{substitute}(\sigma, \tau_\alpha) = \tau_2)$$

`argTy`

`argTy` determines the type of an argument; if the argument is a variable, it looks up its mapping in the type environment.

$$\texttt{argTy} \in \mathit{Environment} \times \mathit{Argument} \rightarrow \mathit{Type}$$

$$\texttt{argTy}(\Gamma, \mathit{arg}) = \begin{cases} \textbf{Int} & \text{if } \mathit{arg} = n \\ \tau & \text{if } \mathit{arg} = x \wedge (x \mapsto \tau) \in \Gamma \end{cases}$$

`branchTypes`

`branchTypes` iterates over a list of pattern branches, evaluating the branch's body expression in an environment where the pattern's binders have been mapped to the matching constructors field types. Note that the number of variables in a pattern branch need not equal the number of fields in the matching constructor; any variables without a matching field (because too many variables were supplied in pattern) are ignored. This is okay because any reference to those omitted variables (which shouldn't be allowed) will be caught during typechecking of the branch's body expression, giving an ill-typed result due to a bad environment lookup, as desired. This rule also shows that each pattern branch must have a matching constructor in the list of constructors; it is considered ill-typed for a pattern matching a constructor of a different datatype to be present in the list of branches of the current case.

$$\texttt{branchTypes} \in \mathit{Environment} \times \overrightarrow{\mathit{Constructor}} \times \overrightarrow{\mathit{PatCons}} \rightarrow \overrightarrow{\mathit{Type}}$$

$$\texttt{branchTypes}(\Gamma, \overrightarrow{cons}, []) = []$$

$$\texttt{branchTypes}(\Gamma, \overrightarrow{cons}, (cn\ \vec{x} \Rightarrow e) :: \overrightarrow{br}) = \tau :: \vec{\tau}$$

where

$$\textbf{true} = (\textbf{con}\ cn\ \vec{\tau} \in cons)$$

$$\Gamma[\vec{x} \mapsto \vec{\tau}] \vdash e : \tau$$

$$\vec{\tau} = \texttt{branchTypes}(\Gamma, \overrightarrow{cons}, \overrightarrow{br})$$

BuiltinTypes

BuiltinTypes maps primitive operator identifiers ($\oplus$ as shown in the abstract syntax) to the types of the primitive operations they represent. For example, + maps to the function type $[\textbf{Int}, \textbf{Int}] \rightarrow \textbf{Int}$. The set of builtin operators and their types is fixed, straightforward, and thus omitted here for brevity's sake.

Datatypes

Datatypes is simply the list of datatypes extracted out of the top-level program definition.

Functypes

Functypes is a map from function and constructor identifiers to their respective types, created from the list of function and datatype declarations that constitute a program. $\overrightarrow{data}$ and $\overrightarrow{func}$ are the list of datatypes and functions, respectively, that constitute the contents of a program (see abstract syntax).

$$\texttt{Functypes} \in \mathit{Operator} \to \mathit{Type}$$

$$\begin{aligned}
&\texttt{Functypes} = \vec{\tau}_c +\!\!+ \vec{\tau}_f +\!\!+ \vec{\tau}_b \\
&\quad \text{where} \\
&\qquad \vec{\tau}_c = \texttt{concatMap}(\texttt{createConsTys}, \overrightarrow{\mathit{data}}) \\
&\qquad \vec{\tau}_f = \texttt{map}(\texttt{createFuncTy}, \overrightarrow{\mathit{func}}) \\
&\qquad \vec{\tau}_b = \texttt{BuiltinTypes}
\end{aligned}$$

createConsTys

createConsTys is a helper function for Functypes, creating types for each of the constructors that are part of a datatype. If a constructor has fields, its type is a function type where those fields are the parameter types and the datatype of which it is a member is the return type. If the constructor doesn't have fields, its type is just the datatype of which it is a member.

$$\texttt{createConsTys} \in \mathit{Datatype} \to (\mathit{Name} \to \mathit{Type})$$

$$\texttt{createConsTys}(\textbf{data}\ \mathit{tn}\ \vec{\alpha}\ \ \textbf{=}\ \overrightarrow{\mathit{cons}}) = \texttt{map}(\texttt{createConTy}(\mathit{tn}\ \vec{\alpha}), \overrightarrow{\mathit{cons}})$$

createConTy

createConTy is a helper function for createConTy, creating a types a constructor that is a part of a datatype.

$$\texttt{createConTy} \in \mathit{Datatype} \times \mathit{Constructor} \rightarrow (\mathit{Name} \rightarrow \mathit{Type})$$

$$\texttt{createConTy}(dt, \textbf{con}\ cn\ \vec{\tau}_c)$$

$$\begin{cases} cn \mapsto (\vec{\tau}_c \rightarrow dt) & \text{if } |\vec{\tau}_c| \geq 1 \\ cn \mapsto dt & \text{otherwise} \end{cases}$$

`createFuncTy`

`createFuncTy` is a helper function for `Functypes`, creating a type for a functions defined in the program.

$$\texttt{createFuncTy} \in \mathit{Function} \rightarrow (\mathit{Name} \rightarrow \mathit{Type})$$

$$\texttt{createFuncTy}(\textbf{fun}\, fn\ \overrightarrow{x : \tau}\ \tau\ \ \textbf{=}\ e) =$$

$$\begin{cases} fn \mapsto (\vec{\tau} \rightarrow \tau) & \text{if } |\overrightarrow{x : \tau}| \geq 1 \\ fn \mapsto \tau & \text{otherwise} \end{cases}$$

`freshTypes`

`freshTypes` is a helper function for `idTy`, replacing all of a type's generic type variables with fresh generic type variables, consistently across the type.

$$\texttt{freshTypes} \in \mathit{Type} \rightarrow \mathit{Type}$$

$$\texttt{freshTypes}(\tau_1) = \tau_2$$

where

$$\vec{\alpha} = \texttt{filter}(\texttt{isGeneric}, \texttt{getTyVars}(\tau_1))$$

$$\sigma = \vec{\alpha} \mapsto \overrightarrow{\texttt{freshGenTV}}$$

$$\tau_2 = \texttt{substitute}(\sigma, \tau_1)$$

freshGenTV

freshGenTV gets a fresh uniquely-identifiable generic type variable.

freshRigTV

freshRigTV gets a fresh uniquely-identifiable rigid type variable.

getConstructors

getConstructors retrieves the constructors associated with a datatype and replaces any type parameters in those constructors' field types with any type variable instantiations recorded in the datatype.

$$\texttt{getConstructors} \in \mathit{Datatype} \rightarrow \overrightarrow{\mathit{Constructor}}$$

$$\texttt{getConstructors}(tn\ \vec{\tau}_t) = \overrightarrow{cons}_2$$

where

$$(\textbf{data}\ tn\ \vec{\alpha}_t\ \textbf{=}\ \overrightarrow{cons}_1) \in \texttt{Datatypes}$$

$$\sigma = \texttt{zip}(\vec{\alpha}_t, \vec{\tau}_t)$$

$$\overrightarrow{cons}_2 = \texttt{map}(\texttt{instCon}(\sigma), \overrightarrow{cons}_1)$$

getTyVars

getTyVars gets the set of type variables used in a type. unions takes the union of each set in a list.

$$\texttt{getTyVars} \in \mathit{Type} \rightarrow \mathcal{P}(\mathit{TypeVariable})$$

$$\texttt{getTyVars}(\tau) = \begin{cases} \{T\} & \text{if } \tau = T \\ \{\vec{T}\} & \text{if } \tau = dt\ \vec{T} \\ \texttt{unions}(\texttt{map}(\texttt{getTyVars}, \vec{\tau}_p)) \cup \texttt{getTyVars}(\tau_r) & \text{if } \tau = \vec{\tau}_p \rightarrow \tau_r \end{cases}$$

idTy

idTy gets the type associated with an identifier from the type environment, if present; otherwise, it looks up the identifier in the set of declared function types. Afterwards, it uses freshTypes to replace all generic type variables in the type with fresh new ones. This function is used during the **let** expression for getting unique instances of types so that type variables do not inadvertently clash during the process of unification and substitution (let-polymorphism).

$$\texttt{idTy} \in \mathit{Environment} \times \mathit{Identifier} \rightarrow \mathit{Type}$$

$$\texttt{idTy}(\Gamma, id) = \begin{cases} \texttt{freshTypes}(\tau) & \text{if } id = x \wedge (x \mapsto \tau) \in \Gamma \\ \texttt{freshTypes}(\tau) & \text{if } id = op \wedge \tau = \texttt{Functypes}(op) \end{cases}$$

instCon

instCon instantiates a constructor by replacing the type variables in its field types with its mapping in the constraint list.

$$\texttt{instCon} \in \mathit{Substitution} \times \mathit{Constructor} \rightarrow \mathit{Constructor}$$

$$\texttt{instCon}(\sigma, \textbf{con}\ cn\ \vec{\tau}) = \textbf{con}\ cn\ \texttt{substitute}(\sigma, \vec{\tau})$$

makeRigid

makeRigid converts all generic type variables in a type into consistently-renamed rigid type variables.

$$\begin{aligned}
&\texttt{makeRigid} \in \mathit{Type} \rightarrow \mathit{Type} \\
&\texttt{makeRigid}(\tau_1) = \tau_2 \\
&\quad \text{where} \\
&\qquad \vec{\alpha} = \texttt{getTyVars}(\tau_1) \\
&\qquad \sigma = \vec{\alpha} \mapsto \overrightarrow{\texttt{freshRigTV}} \\
&\qquad \tau_2 = \texttt{substitute}(\sigma, \tau_1)
\end{aligned}$$

principalType

principalType determines if a list of types all refer to the same type, determining the most general type that matches all of them. For example, it is used at the end of the case typing rule to check that all branch bodies have a compatible type. This check is different than a normal check for equality because it takes into account type variables.

$$\begin{aligned}
&\texttt{principalType} \in \overrightarrow{\mathit{Type}} \rightarrow \mathit{Type} \\
&\texttt{principalType}(\tau_h :: []) = \tau_h \\
&\texttt{principalType}(\tau_1 :: \tau_2 :: \vec{\tau}_{tl}) = \texttt{principalType}(\tau_3 :: \vec{\tau}_{tl}) \\
&\quad \text{where} \\
&\qquad \sigma = \texttt{unify}(\{\tau_1 = \tau_2\}) \\
&\qquad \tau_3 = \texttt{substitute}(\sigma, \tau_2)
\end{aligned}$$

`substitute`

`substitute` takes a constraint set and a type, recursively replacing all type variables present as keys in the constraint set with their associated value.

$$\texttt{substitute} \in \mathit{ConstraintSet} \times \mathit{Type} \to \mathit{Type}$$

$$\texttt{substitute}(C, \tau) = \begin{cases} \tau_2 & \text{if } \tau = T \land (T = \tau_1) \in C \land \tau_2 = \texttt{substitute}(C, \tau_1) \\ tn\ \vec{\tau}_2 & \text{if } \tau = tn\ \vec{\tau}_1 \land \vec{\tau}_2 = \texttt{map}(\texttt{substitute}(C), \vec{\tau}_1) \\ \vec{\tau}_{p2} \to \tau_{r2} & \text{if } \tau = \vec{\tau}_{p1} \to \tau_{r1} \land \\ & \quad \vec{\tau}_{p2} = \texttt{map}(\texttt{substitute}(C), \vec{\tau}_{p1}) \land \\ & \quad \tau_{r2} = \texttt{substitute}(C, \tau_{r1}) \\ \tau & \quad \text{otherwise} \end{cases}$$

`unify`

`unify` performs the standard process of unification, iterating over each constraint in the constraint set and creating a substitution that contains a mapping from type variables to types. Any cases unlisted in this function imply that the function fails in that case. Regarding rigid type variables: the rigid types β_1 and β_2 are successfully unified if and only if β_1 equals β_2. This is because in the context of checking a function, rigid type variables cannot be replaced or unified with any other concrete type or polymorphic type variable. For compound types like functions and data types, unification proceeds recursively.

Similar to map creation, we use the notation $\{\vec{\tau}_1 = \vec{\tau}_2\}$ to denote the creation of a constraint set where the first element of $\vec{\tau}_1$ is paired with the first element of $\vec{\tau}_2$, the second element of $\vec{\tau}_1$ paired with the second element of $\vec{\tau}_2$, etc.

$$\texttt{unify} \in \mathit{ConstraintSet} \rightarrow \mathit{Substitution}$$

$$\texttt{unify}(\emptyset) = \{\}$$

$$\texttt{unify}(\{\tau_1 = \tau_2\} \cup C_1) =$$

$$\begin{cases} \texttt{unify}(C_1) & \text{if } \tau_1 = \tau_2 \\ \sigma[\alpha \mapsto \tau_2] & \text{if } \tau_1 = \alpha \wedge \alpha \notin \texttt{getTyVars}(\tau_2) \wedge \\ & C_2 = \texttt{updateConstraints}(C_1, \{\alpha = \tau_2\}) \ \wedge \sigma = \texttt{unify}(C_2) \\ \sigma[\alpha \mapsto \tau_1] & \text{if } \tau_2 = \alpha \wedge \alpha \notin \texttt{getTyVars}(\tau_1) \wedge \\ & C_2 = \texttt{updateConstraints}(C_1, \{\alpha = \tau_1\}) \ \wedge \sigma = \texttt{unify}(C_2) \\ \texttt{unify}(C_2) & \text{if } \tau_1 = \mathit{tn}\ \vec{\tau}_1 \wedge \tau_2 = \mathit{tn}\ \vec{\tau}_2 \wedge |\vec{\tau}_1| = |\vec{\tau}_2| \wedge \\ & C_2 = C_1 \cup \{\vec{\tau}_1 = \vec{\tau}_2\} \\ \texttt{unify}(C_2) & \text{if } \tau_1 = \vec{\tau}_{p1} \rightarrow \tau_{r1} \wedge \tau_2 = \vec{\tau}_{p2} \rightarrow \tau_{r2} \wedge |\vec{\tau}_{p1}| = |\vec{\tau}_{p2}| \wedge \\ & C_2 = C_1 \cup \{\vec{\tau}_{p1} = \vec{\tau}_{p2}\} \cup \{\tau_{r1} = \tau_{r2}\} \end{cases}$$

`updateConstraints`

`updateConstraints` iterates through a constraint set, replacing all type variables in each constraint by their mapped types, if present in the substitution. This is a helper function used only by `unify`.

$$\texttt{updateConstraints} \in \mathit{ConstraintSet} \times \mathit{Substitution} \rightarrow \mathit{ConstraintSet}$$

$$\texttt{updateConstraints}(\emptyset, \_) = \emptyset$$

$$\texttt{updateConstraints}(\{\tau_1 = \tau_2\} \cup C_1, \sigma) = \{\tau_3 = \tau_4\} \cup C_2$$

where

$$\tau_3 = \texttt{substitute}(\sigma, \tau_1)$$

$$\tau_4 = \texttt{substitute}(\sigma, \tau_2)$$

$$C_2 = \texttt{updateConstraints}(\sigma, C_1)$$

7.3 Instruction Checking

As mentioned earlier, the Binary Exclusion Unit (BEU) can be used as a runtime guard, checking programs right before execution when they are loaded into memory, or as a program-time guard, checking programs when they are placed into program storage (flash, NVM, etc.). In either case, checking works the same way: each word of the binary is examined one at a time as it streams through. Central to this process is the embedded Type Reference Table (TRT), which is copied from the binary into the checker's memory and contains the type information for the binary. This serves as a reference during all stages of the checking process, and will be extended during the checking of each function as local variables are introduced. Later, when the BEU arrives at a new function, it consults the function signature which provides type information for the arguments and the return type of the function. Each instruction in the function is then scanned word-by-word, guaranteeing type safety of each instruction according to the static semantics (Figure 7.1).

The high-level idea of the checking is as follows. The binary TRT supplies type information for the arguments and return type of each function. Starting with the types of the arguments, the Binary Exclusion Unit can propagate type information through each function, inferring the type of each variable by comparing the types of indicated functions with the types of the arguments supplied. At the end of the function, the type of the variable being returned must match the return type specified in the signature. Checking can fail at any step of the process: e.g., a function might expect an `Integer` but is passed a `List`, or the `add` function, which expects two `Integer` arguments, is given three. A single type violation causes the entire program to be rejected. The steps required to check each instruction class are described in more detail below:

When a `Let` instruction is encountered, we first check for special-case operations: applying no arguments to something will always result in the same thing, so we can simply assign the result to that type and do no further checking. Assuming the `Let` does have arguments, the checker then gets the type of the function and creates an alias of it in a new TRT entry. The point of the alias is to make each type variable unique — e.g., the same type `List a` (a list of elements of type “a”) used in two places may not be using the same type for `a`, so the separate usages should have separate type variables. In order to allow recursive `Let` operations, a type variable is assigned to the result of the operation; when all the arguments have been processed, that variable will be set equal to what’s left. The checker goes through each argument, one at a time, and unifies its type with the function’s expected type. This creates a list of constraints that, along with the constraint on the resulting type variable, are checked altogether as the last step. If there are no inconsistencies in the constraint set, the operation was valid, and a new valid type is produced for the local variable.

Because type inference is relatively simple, we chose to forgo type annotations on each function application that indicate the result of the operation. Instead, the checker uses function-local type inference to figure out the return type of each function application. Because function calls (`Let` instructions) make up the majority of the instructions in a binary, the absence of annotations on each one results in much smaller binary sizes for typed binaries.

Special care must be taken in Let instructions when the resulting type is a function, and when the function being applied has a function in its return type. The former requires creating a new TRT entry for the function; the latter requires a special "unfolding" routine to begin applying arguments to the function in the return type. Both of these are reasons that the Let section of the hardware checker has so many states (Table 7.2).

Case instructions are much more straightforward. The checker simply saves some type information on what the program is casing on, which is used in later instructions. Specifically, the primary task is to get the type of the scrutinee (the thing being cased on) and save a reference both to the particular variable's type and the root program datatype (assuming the variable is a constructor, not an integer). For example, this way branches will know that a List was cased on, not a Tuple, and know that the particular variable was a List Int as opposed to a List Char.

Pattern_literal branch heads are quite simple: the case head must be an integer, and the value specified in the instruction must be an integer.

Pattern_con branch heads are one of the more complex things to check. We have to reconcile the generic type of the indicated pattern (constructor) with the specific type of the variable that we're matching against. To do this, the checker must get the function type specified in the pattern head, then alias it in a new TRT entry. Then we must generate the constraint that the return type of the function is the same as the type of the scrutinee — this ensures that the type variables in this entry will be constrained to be the same as those in the original scrutinee. Constraints can then be checked, yielding a map with which the variables can be recursively replaced to the correct types. Finally, a pointer is set to where the fields of the constructor begin (if applicable). When we are done, we have direct, usable information on the type of each field in the constructor, which can be used by following instructions.

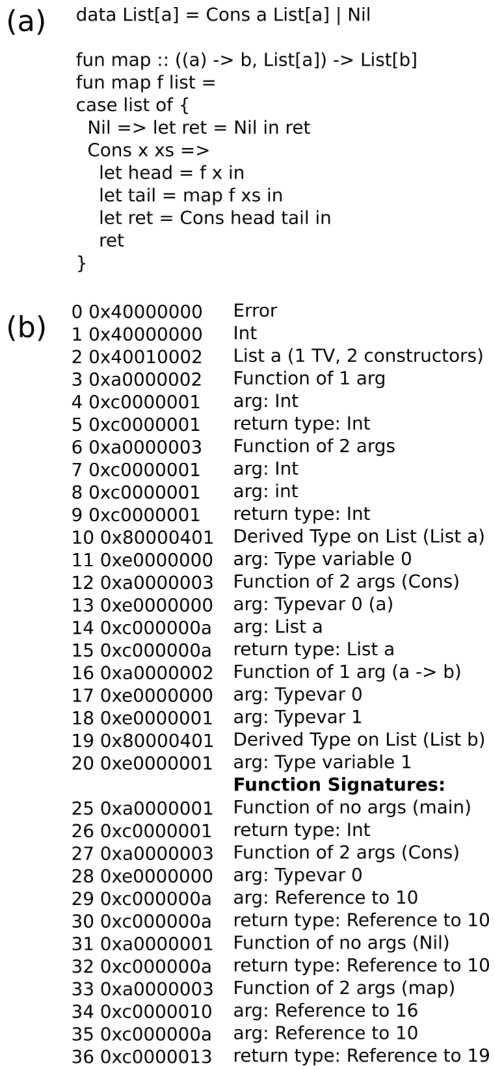
In addition, we must keep track of which constructors we've seen in this case statement; that way, when we get to the end of the Case, we'll know if all of the constructors of that type were present or not. A Case statement must either contain an else branch or use all constructors of the scrutinee's type.

7.4 Hardware Implementation

In building a static analysis hardware engine directly into an embedded micro-controller, one of the big advantages of customization is that at the hardware level we can see, *either physically through inspection or through analysis at the gate or RTL level*, exactly how information is flowing through a system to introduce safety or security mechanisms that are truly non-bypassable. No software can change the functioning of the system at that level. However, doing static analysis at the level of machine code is no easy task — even for software.

Fortunately, there are some great works to draw inspiration from. Previous work has used types to aid in assembly-level analysis; specifically TAL [55] and TALx86 [56] have created systems where source properties are preserved and represented in an idealized assembly language (the former) or directly on a subset of x86 (the latter). Working up the stack from assembly, other prior works attempt to prove properties and guarantee software safety at even higher levels of abstractions. We seek to take these software ideas and find a way to make them *intrinsic* properties of the physical hardware for embedded systems where needed.

In this work we draw upon the opportunity afforded by architectures that have already been designed with ease of analysis in mind. Specifically, we leverage the Zarf ISA, a purely functional, immutable, high-level ISA and hardware platform used for binary reasoning, which is suitable for execution of the most critical portions of a system [6]. At a high level the Zarf ISA consists of three instructions: Let performs function application and object allocation, applying arguments to a function and creating an object that represents the result of the call. Case is used for both pattern-matching and control flow. One cases on a variable, then gives a series of patterns as branch heads; only the branch with the matching pattern is executed.



(a)

```
data List[a] = Cons a List[a] | Nil

fun map :: ((a) -> b, List[a]) -> List[b]
fun map f list =
case list of {
  Nil => let ret = Nil in ret
  Cons x xs =>
    let head = f x in
    let tail = map f xs in
    let ret = Cons head tail in
    ret
}
```

(b)

```
0 0x40000000     Error
1 0x40000000     Int
2 0x40010002     List a (1 TV, 2 constructors)
3 0xa0000002     Function of 1 arg
4 0xc0000001     arg: Int
5 0xc0000001     return type: Int
6 0xa0000003     Function of 2 args
7 0xc0000001     arg: Int
8 0xc0000001     arg: int
9 0xc0000001     return type: Int
10 0x80000401    Derived Type on List (List a)
11 0xe0000000    arg: Type variable 0
12 0xa0000003    Function of 2 args (Cons)
13 0xe0000000    arg: Typevar 0 (a)
14 0xc000000a    arg: List a
15 0xc000000a    return type: List a
16 0xa0000002    Function of 1 arg (a -> b)
17 0xe0000000    arg: Typevar 0
18 0xe0000001    arg: Typevar 1
19 0x80000401    Derived Type on List (List b)
20 0xe0000001    arg: Type variable 1
                 Function Signatures:
25 0xa0000001    Function of no args (main)
26 0xc0000001    return type: Int
27 0xa0000003    Function of 2 args (Cons)
28 0xe0000000    arg: Typevar 0
29 0xc000000a    arg: Reference to 10
30 0xc000000a    return type: Reference to 10
31 0xa0000001    Function of no args (Nil)
32 0xc000000a    return type: Reference to 10
33 0xa0000003    Function of 2 args (map)
34 0xc0000010    arg: Reference to 16
35 0xc000000a    arg: Reference to 10
36 0xc0000013    return type: Reference to 19
```

Figure 7.2: An example Type Reference Table (TRT) for the function `map`. The original program is shown in **(a)**, while **(b)** shows the actual binary type information that the assembler produces (annotated for human understanding). This type information is included at the head of the binary file, leaving the program unchanged. The first section lists types used in the signatures of the program, while the second section contains type info for the parameters and return type of each function. The type system is polymorphic and uses function-local type inference.

Patterns can be constructors (datatypes) or integer values, depending on what was cased on. `Result` is the return instruction; it indicates what value is returned at the end of a function. Branches in case statements are non-reconvergent, so each must end in a result instruction.

A big advantage of this ISA for static analysis is that it has a compact and precise semantics. If we could could guarantee the physical machine would always execute only according to these semantics (e.g. always respecting call/return behavior, using the proper number of arguments from the stack, etc.) we would end up with a system that has some very desirable properties. In this chapter, we show that these include verifiable control flow integrity, type safety, memory safety, and others; e.g., ROP [101] is impossible, programs never encounter type errors, and buffer overruns can never happen.

Unfortunately, the semantics of any language govern the behavior of execution only for “well-formed” programs. When we are talking about machine code, as opposed to programming languages, things are a little trickier, because machines are expected to read instruction bits from memory and execute them faithfully as they arrive. As we describe in more detail below, checking membership in the language of well-formed Zarf programs is actually something that requires some sophistication and would be difficult to do at run-time. Even though there are just three instructions, Zarf binaries support casing, constructors, datatypes, functions, and other higher-level concepts as first-class citizens in the architecture. Our goal is to correctly implement these checks statically and show that the only binaries that can *ever* execute on this machine pass this static analysis.

7.4.1 The Analysis Implemented

While one could, in theory, capture every possible deviation from the Zarf semantics with a set of run-time checks in hardware, actually catching every possible thing that can go wrong quickly grows in complexity. An advantage of static checking over dynamic checks is that

Possible failure:	**Meaning:**
malformed instruction	Bit sequence does not correspond to a valid instruction.
fetch out-of-bounds arg	Accessing argument *N* when there are fewer than *N* arguments.
fetch out-of-bounds local	Accessing local *N* when there are fewer than *N* locals allocated.
fetch out-of-bounds field	Accessing field *N* when there are fewer than *N* fields in the case'd constructor.
fetch invalid source	Bit sequence does not correspond to a valid source.
apply arguments to literal	Treating a literal value as a function and passing arguments to it.
apply arguments to constructor	Treating a saturated constructor as a function and passing arguments to it.
application with too many args	Passing more arguments than a function can handle, even if it returns other functions.
application on invalid source	Invalid source designation for function in application.
oversaturated error closure	Passing arguments to an error closure.
oversaturated primitive	Passing more arguments than a primitive operation can handle.
passing non-literal into primitive op	Passing an object (constructor or closure) into a primitive operation.
case on undersaturated closure	Trying to branch on the result of a function that cannot be evaluated.
unused arguments on stack	Oversaturating a function and branching on the result when not all arguments have been consumed.
matching a literal instead of a pattern	Branching on a function that returns a constructor, but trying to match an integer.
invalid skip on literal match	Instruction says to skip *N* words on failed match, but that location is not a branch head.
no else branch on literal match	Incomplete case statement because of lack of else branch.
matching a pattern instead of a literal	Branching on a function that returns an integer, but trying to match a constructor.
incomplete constructor set in case statement	Incomplete case statement because not all possible constructors are present.
invalid skip on pattern match	Instruction says to skip *N* words on a failed match, but that location is not a branch head.
no else branch on pattern match	Incomplete case statement because of lack of else branch.

Table 7.1: Summary of 21 conditions that require dynamic checks in the absence of static type checking. With our approach, checking is achieved ahead of time, in a single pass through the program; energy and time are not wasted with repeated error checking. No information needs to be tracked at runtime, and the only runtime hardware check is for out-of-memory errors. All of the listed errors are guaranteed by our type system to not occur.

once the binaries are analyzed, no additional energy and time costs are required during execution. For an embedded system that runs the same code continuously, any small static cost is amortized rather quickly. As we will show later, in fact the static analysis can actually be done in a *single streaming pass* over the executable. However, just to see the scope of the problem it is useful to enumerate some of the dynamic checks that would be required to achieve the same objective as our hardware static analysis.

Table 7.1 lists ways that programs can fail and costs that are incurred if one were to dynamically check for errors on the platform. There are 21 different ways for the hardware to throw errors, the great majority of which require keeping some significant bookkeeping to actually check. At the very least, we would need to keep extra information on number of arguments, number of local variables, number of recently cased constructor fields, and runtime tags on heap objects to distinguish between closures and constructors — all of which the hardware would need to track at runtime. Crucially, this information must be incorruptible and inaccessible to the software for the dynamic checks to be sound. If software is able to access and corrupt this information, it compromises the integrity of the dynamic checks. In general, guaranteeing that the set of dynamic checks are always occurring, i.e. not bypassed, can be very difficult. With a hardware-implemented static analysis, we are able to formally prove that our checks cannot be bypassed. In addition to the hardware implementation overhead of these checks, reasoning about software behavior in the face of dynamic checks become more difficult as well if error states are returned. Programmers that wish to handle errors due to code that fails such checks are forced to reason about every situation that can arise (e.g. what if this function encounters an oversaturated primitive, or cases on an undersaturated closure, and so on.). Instead, by performing the checks statically, all software components understand that any other component that they might interact with on the system is subject to the same analysis as their own code.

7.4.2 The Bouncer Architecture

Given that we can develop a unit to actually perform the desired static analysis, a big question is where it fits into the actual micro-controller design. Figure 7.4 shows how a static analysis engine (the Binary Exclusion Unit) fits into an embedded system at a high level: all incoming programs are vetted by the checker before being written to program storage, ensuring that all code that the core executes conforms to the type system's proven high-level guarantees. During programming mode, as a binary image is loaded into the core, the checker has write access to the program store and can use a set of data memory as a working space. The Binary Exclusion Unit can thus be used as a runtime guard, checking programs right before execution when they are loaded into memory, or as a program-time guard, checking programs when they are placed into program storage (flash, NVM, etc.).

Only once the programming mode is complete do the instruction and data memory become visible. The upshot of catching errors this way is that this gives you feedback at programming time, before a device is deployed, that the binary contains errors. It further ensures that when reprogramming occurs in the field, malicious or malformed code that exploits interactions outside of the ISA semantics will never be loaded.

In either case, checking works the same way: each word of the binary is examined one at a time in a linear pass over the program as it is fed through the Binary Exclusion Unit. It is trivial to verify that the BEU is the only unit given access to write to the memory — the more interesting discussion, covered later, is the verification that the only way though the BEU is via a static analysis.

7.4.3 Implementation Details

At a high level, the BEU is a hardware implementation of a pushdown automaton (PDA) and is structured as a state-machine with explicit support for subroutine calls. While there a numerous bookkeeping structures required, we must take care to access a single structure at a time to ensure we do not create structural hazards. The final analysis hardware is the result of a chain of successive lowerings from a high-level static semantics ending with a concrete state chart that we could then implement with minimal and straighforward hardware. First, a bit-accurate software checker was made that checked binary files. Then, a cycle-accurate software pushdown automata was written from that refined specification. From that program, the leap to real hardware was somewhat straightforward. While the full details of the checker cannot

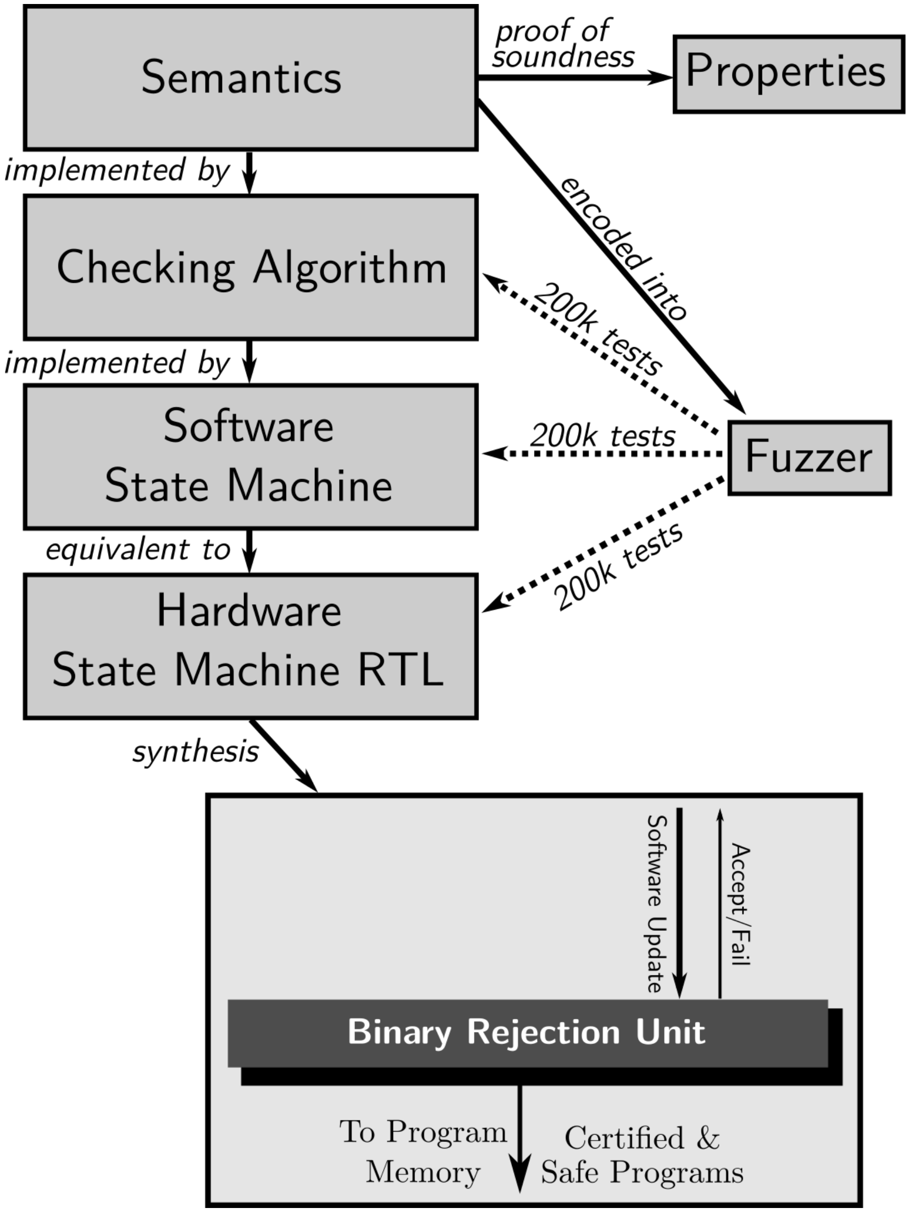


Figure 7.3: High-level flow of our system. We formalize a static semantics and prove soundness on it, from which useful security properties, like type safety, memory safety, and control integrity, can be derived. A typechecking algorithm implements the semantics, and the algorithm is lowered into a software-level state machine. A hardware RTL version is written based on that, from which actual hardware is synthesized. The three design layers are subjected to extensive testing from a program-generating fuzzer that encodes the high-level semantics to generate well- and ill-typed programs. With high confidence that the hardware correctly implements the high-level semantics, the derived properties now apply to every binary program loaded onto the machine, and broad swaths of errors and attacks become impossible.

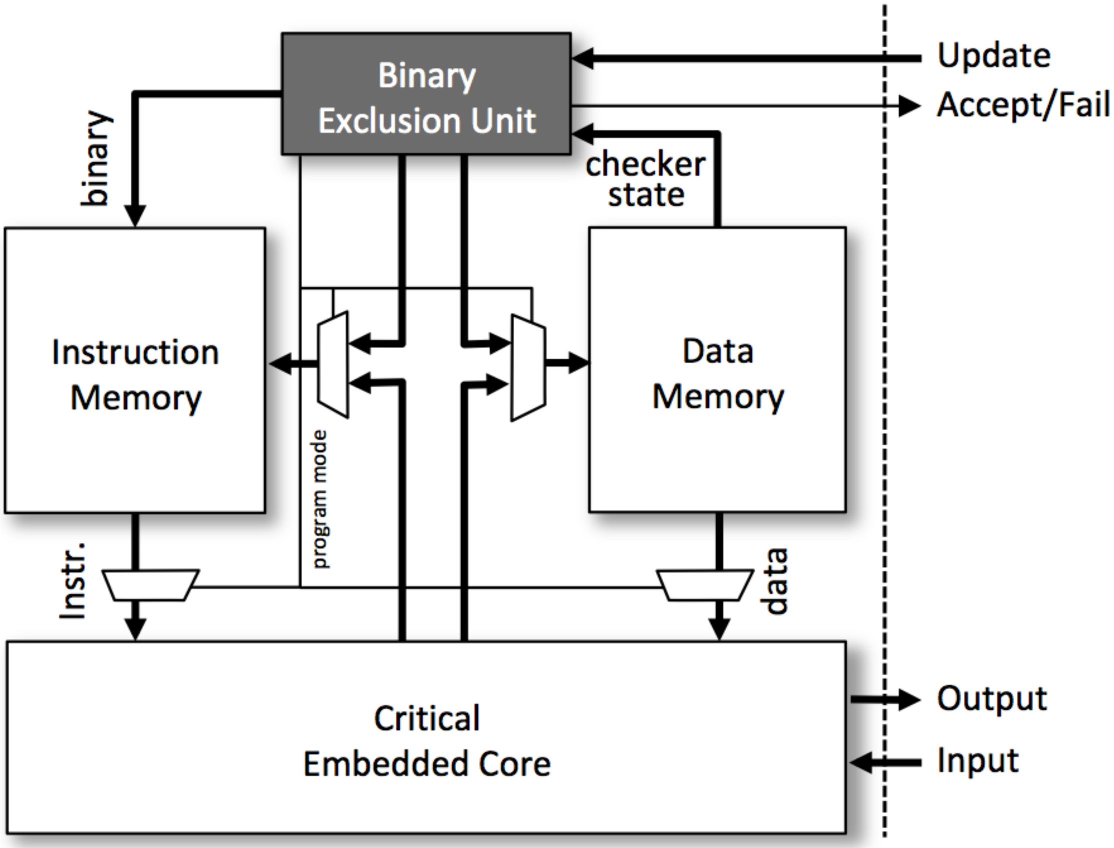


Figure 7.4: The Binary Exclusion Unit works as a gatekeeper, only allowing input binary programs if they pass the static analysis. When in "Programming" mode, the core is halted while the program is fed to the checker; if it passes, it is written to the system instruction memory. The checker makes use of the core's data memory, which is otherwise unused during system programming. At run-time, the checker is disabled and consumes no resources. Programs that pass static analysis are guaranteed to be free of memory errors, type errors, and control flow vulnerabilities. The checker is non-bypassable; all input binaries are subject to the inspection.

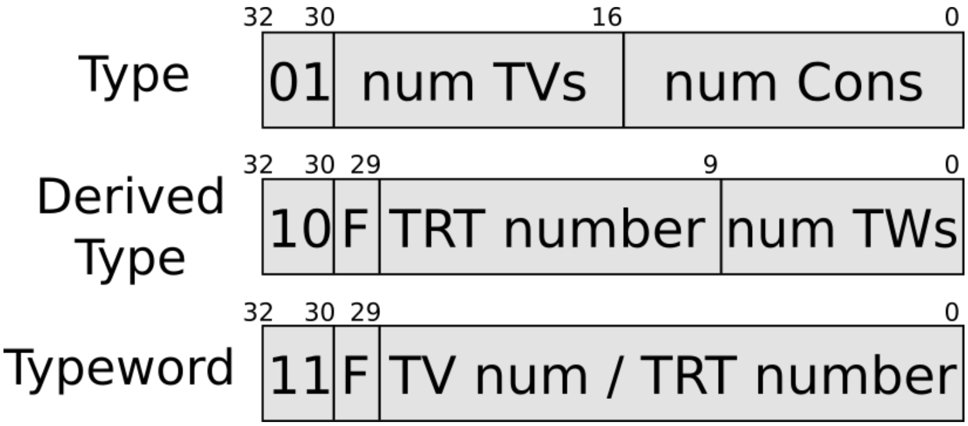


Figure 7.5: Binary encoding of entries in the Type Reference Table (TRT), indicating bit-fields usage for each entry type. A `Type` entry is used for program-defined datatypes, and specifies how many type variables are used and how many constructors are declared. A `Derived Type` contains references to other types; it can represent functions and constructors. If the flag (F) field is 1, the entry is a function, and the final field indicates how many Typewords follow as arguments. If 0, the entry is a constructor, and the type variables following indicate types to use as the constructor's type variables. A `Typeword` is used as an argument to a Derived Type. If the flag (F) field is 1, it represents a type variable; otherwise, it is a reference to another entry in the TRT.

hope to fit in this submission, we plan to make the complete design open source (including our synthesizable verilog, low-level software reference, and high level algorithm reference). We concentrate here on the strategy used at a high level and a couple of details to give a better sense for the full design.

The first challenge in implementing this analysis is how to encode the type information into the binary. As discussed in the prior section, we put this information at the head of each binary in the form of the TRT. To get a sense of what that actually looks like in a real implementation, Figure 7.2 shows an example TRT for the function `map`. This information is discarded after checking, leaving a standard, untyped binary, which executes with normal performance.

At the bit-level, we see only a sequential series of bytes. Therefore, all type information must be encoded into a single list. To avoid unnecessary complexity, we make all entries in the TRT fixed-width 32-bit words. An entry can be either 1) a program-specified datatype or built-in type[4], or 2) a derived type based on another type. Entries of type 2 can have one or more argument words, which we refer to as "typewords." "Derived" here means that the entry contains references to other types in the table. This manifests as either a type applied to some type variables, or as a function. For example, `List` is specified as a program datatype with one type variable, then derived type entries can create the types `List a`, `List Int`, etc, where `a` and `Int` are typewords following the derived type entry.

The second challenge in bringing the typechecker to a low level is dealing with recursive types. Implicitly, types in the system may be arbitrarily nested: for example, one could declare a `List` of `Tuple`s of `List`s of `Int`s. During the checking process, the hardware typechecker must be able to recursively descend through a type in order to make copies, do comparisons, and validate types. Because of this, the Binary Exclusion Unit cannot be expressed as a simple state machine — a stack is required for recursive operations (and hence the pushdown automaton).

Data structures used in the higher-level checking, like maps, need to be converted to structures native to hardware: they must flattened into a list, which can be stored in memory. In some cases, this requires a linear scan to check for the presence of some elements, such as checking case completeness — but those lists tend to be small, containing just one entry for each constructor of a given datatype. We found that all of the structures could be represented as lists with stack pointers, except in the case of the type variable map used in the recursive replace procedure, which required two lists (one to check for membership in the map, the second with values at the appropriate indices).

To create the control structure of the PDA, we started by implementing a software-level checker, broken into a set of functions implemented with discrete steps, where each step cannot access more than one array in sequence (in hardware, the arrays will become memories, which we do not want strung together in a single cycle). While, given our space constraints,

[4] The Zarf ISA includes integers and an error datatype built-in.

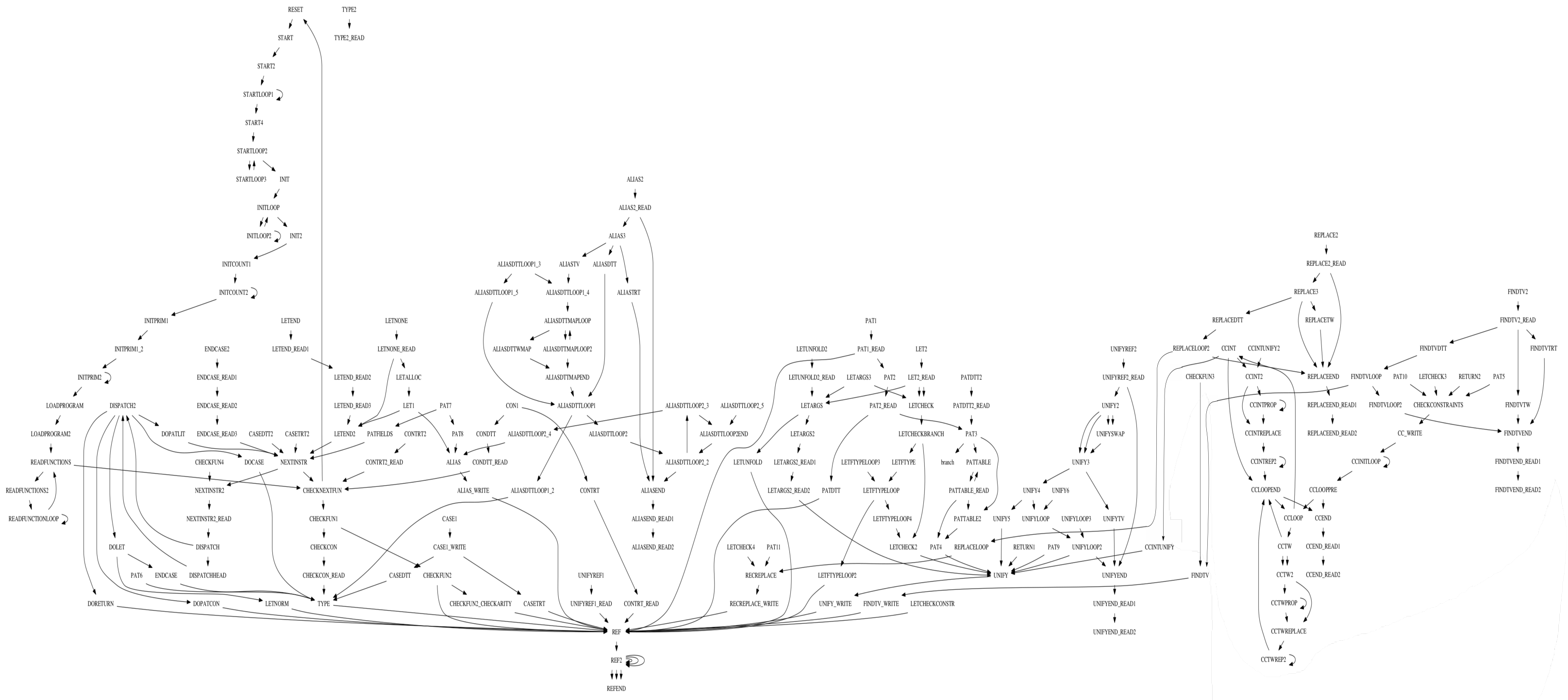


Figure 7.6: The full state machine portion of the hardware Binary Exclusion Unit, which typechecks a binary, polymorphic type system. The machine makes use of several subroutines, so some states are missing incoming or outgoing edges; these correspond to jumps to and from a subroutine.

Purpose	**Number of States**
Initialization	21
Function signatures	15
Dispatch	6
Let checking	37
Return checking	3
Case checking	7
Literal pattern checking	1
Constructor pattern checking	21
Following references	6
Type variable (TV) counting	12
Recursive TV replacement	12
Recursive TV aliasing	26
Generating constraints	19
Checking constraints	21
Total	**207**

Table 7.2: Number of states devoted to the various parts of the Binary Exclusion Unit's state machine. Checking function calls, allowing for polymorphic functions with type variables, and constraint checking were the most complex behaviors, making up most of the states.

it is tough to describe the system in detail, the number of states for each part of the analysis is a reasonable proxy for complexity. The resulting state machine has 207 states and they are broken down by purpose in Table 7.2. We summarize them briefly here, with number of states denoted in parentheses. The initialization stage reads the program and prepares the type table (21 states). Function heads are checked to ensure the argument count matches the provided function signature, and bookkeeping is done to note the types of each argument and the return type (15). Dispatch decides which instruction is executed next, and handles saving and restoring state as necessary for `Case` statements (6). `Let` (37), `Result` (3), `Case` (7), `Pattern_literal` (1), and `Pattern_con` (21) are checked as outlined in Section 7.3.

Because types can be recursively nested, a type entry in the TRT can reference other types; a set of states are devoted to following references to find root types as needed (6 states). To handle this, the state machine implements something akin to subroutines. A routine executes at the beginning of each function that counts the number of type variables used in the signature (these type variables are "rigid" within the scope of the function and cannot be forced to equal a concrete type) (12). Another routine recursively replaces type variables to make one type entry match the variables in another; it allows pattern heads to be forced to the same type as the variable in the `Case` instruction (12). The aliasing subroutine recursively walks a type and maps its type variables to a "fresh" set (26). This allows, for example, each usage of `List a` to have "`a`" refer to a different type. Part of the complexity of this task is keeping track of the variables already seen and what they map to so that a variable is not accidentally mapped twice to different values. Constraint generation takes two type entries and, based on the entries, branches and generates the appropriate constraint for the constraint set indicating that the entries should be equal (19).

Finally, we have the constraint checking routine (21). This is invoked at the end of each `Let` instruction, as well as after a `Result`. Constraint propagation proceeds by taking one constraint from the set, which consists of a pair of types, then walking all the remaining constraints in the set and replacing all occurrences of the first type with the second. In this way, for each unique constraint, one type is eliminated from the constraint set. If at some point two different concrete types (like `Int` and `List`) are found to be equal, the set is inconsistent and typechecking fails, rejecting the program. Similarly, if ever a rigid type variable (a type variable used in the function signature) is found to be equal to a concrete type, typechecking fails. This second fail condition ensures that functions with polymorphic type signatures are, in fact, polymorphic. Without it, one could write a function that takes `a` and returns `a`, which *should* work for all types, but in fact only works for integers.

As we developed our software and hardware checkers, we used a novel software fuzzing technique to generate 200,184 test cases. Rather than generating random bits, which would not meaningfully exercise the checker, we encode the type system's semantics with logic programming and run them "in reverse" to generate, rather than check, well-typed programs. By performing mutations on half of these programs, we also generate ill-typed benchmarks. In all 200,184 generated test cases, the simulated hardware RTL has 100% agreement with the high-level checker semantics. The tests provide 100% of coverage of all 207 states of the checker.

While the resulting analysis engine is complex, one could certainly reuse parts of the analysis for other sets of properties and automated translation would be an interesting direction for future work. The software model is 1,593 lines of Python, while the hardware RTL is 1,786 lines (requiring extra code for declarations and the simulation test harness). Synthesis results are found in Section 7.7.4.

7.5 Fuzzing the Checker

Even though we specifically design the type system to be efficiently checkable and to hold needed properties, a critical problem in creating a low-level typechecker suitable for hardware implementation is ensuring that the checker behavior matches the specified high-level static semantics. To provide confidence on this front, we subject all layers to a large number of test cases (200,184), generated via a structured fuzzing procedure which we believe has not been used in computer architecture prior. For all tests, the three levels (high level checker, architecture-level specification, and gate-level implementation) are in agreement on which programs are well-typed and which are ill-typed.

Fuzzing [102] is a popular technique for automatically finding bugs in software, with notable applications in testing C compilers [103, 104], JavaScript interpreters [105, 106], and constraint solvers [107, 108, 109]. The general idea behind fuzzing is to generate program inputs through some black-box process, followed by running the software under test (SUT) with these inputs. If the SUT performs incorrectly under the input (e.g., by crashing or producing malformed output), then a bug has been detected in the SUT.

Simply generating random binaries would not be helpful to fuzz our checker implementation; to find an error, we are specifically looking for both examples of rejecting a well-typed program and failing to reject an ill-typed program. Truly random binaries are incredibly unlikely to be well-typed. The resulting failures are immediate, uninteresting, and do not exercise even a small fraction of the checker state space.

We note that finding these behaviors has been examined before in the programming languages field [110]. These checkers can be implemented using constraint logic programming (CLP) [111, 112]. CLP can generate structured random binaries by exploiting its unique nondeterministic nature: the generating constraints (choices of functions, instructions, etc.) are chosen randomly at run-time, producing a different program each time it executes. Taking a program as input would fix which syntactic element is chosen at each juncture, but by effectively "reversing" the execution of the CLP program, we can instead generate well-typed programs. Instead of asking "is this program well-typed?", we can instead ask "what is a well-typed program?" This can be done efficiently with preexisting CLP engines (e.g., SWI-PL [113] and GNU Prolog [114]).

Ill-typed programs were generated with a relatively simplistic mutation-based technique inspired by Holler et al. [106]; first a well-typed program is generated, then a randomly-selected syntactic element is changed. That program is run through the CLP typechecker to ensure that the change resulted in an ill-typed program. The fuzzer is capable of generating dozens to hundreds of well-typed programs per minute.

7.6 Provable Non-bypassability

The hardware static analysis we developed has a variety of states governing when it is active, how it initializes, and so on. An important point of this paper is the non-bypassibility of these checks, but we need to know that some sequence of inputs cannot be given to the checker that cause outputs to write to memory that have not been checked by analysis. To solve this problem, we can create an assertion and employ the Z3 SMT solver [115] to check it for us. Z3 is well-suited to our task because of its ability to represent logical constructs and solve propositional queries. In addition, because we can directly represent the circuit in Z3 at the logic level (gates), we do not have to operate at higher levels of abstraction and risk the proof not holding for the real hardware.

We actually translate our entire analysis circuit into a large Z3 expression. Then, we add two constraints: the first says that, at some point in the operation of the circuit, it output the "passed" (meaning well-typed) signal, while the second says that at no point did the hardware enter the checking states. If the conjunction of the expressions is unsat, there is no way to get a "pass" signal without undergoing checking (and the program will never be loaded if it fails checking). Around 30 of the states deal with program loading, initialization, etc., and perform no checking; our proof guards against, e.g., situations in which some clever manipulation of the state machine moves it from initialization directly to passing, or otherwise manages to circumvent the checking behavior of the state machine.

In the most direct strategy, we use the built-in `bitvec` Z3 type for wires in the circuit, with gates acting as logical operations on those bitvectors. Memories are represented as arrays. Arrays in Z3 are unbounded, but because we address the array with a bitvector, there is an implicit bound enforced that makes the practical array non-infinite.

A straightforward approach to handling sequential operation of the analysis is to duplicate the circuit once for each cycle we wish to explore. The cycle number is appended to the name of each variable to ensure they are unique. Obviously, because the entire circuit is duplicated for each cycle, this method does not scale well — both in terms of memory usage and the time it takes to determine satisfiability. Checking non-bypassability for numbers of cycles up to 32 took under 2 minutes and used less than 1 GB of RAM. Checking for 64 cycles used almost 16 GB and did not terminate within four days.

To make the SMT query approach scalable, we employ Z3's theory of arrays. Instead of representing each wire as a bitvector, duplicated once for each cycle, we represent it as an array mapping integers to bitvectors: the integer index indicates the cycle, while the value at the index is the value the wire takes in that cycle. There is then one array for each wire in the circuit, and one array of arrays for each memory in the circuit (the first array represents the memory in each cycle, while the internal array gives the state of the memory in that cycle). Logical expression (gates) can then be represented as *universal quantifiers* over the arrays. For example, an AND expression might look like, `ForAll(i, wire3[i] == wire1[i] & wire2[i])`. This constrains the value of `wire3` for all cycles. Sequential operations are easy, simply referring to the previous index where necessary for register operations, e.g. `ForAll(i, reg1[i] == reg1_input[i-1])`. To bound the number of cycles, we add constraints to each universal quantifier that `i` is always less than the bound; this prevents Z3 from trying to reason about the circuit for steps beyond `i`.

Solving satisfiability with arrays took under two minutes and under one GB of RAM, no matter what bound we placed on the cycle count — in fact, even when *unbounded*, Z3 was still able to demonstrate our hardware analysis bypassibility assertion was unsatisfiable — i.e., the circuit is non-bypassable.

7.7 Checker Performance

7.7.1 Checking Benchmarks

While the fuzzer's programs took an average of 88 cycles per instruction to check, most of the benchmarks took 150-160 cycles per instruction to check (see Figure 7.8). The MiBench programs were hand-coded in typed Zarf assembly. Common imperative workloads, such as SPEC2006, are not included as compiling directly from imperative C to the functional ISA is not a solved problem. Interestingly, one benchmark (CRC32) took over 1,000 cycles per in-

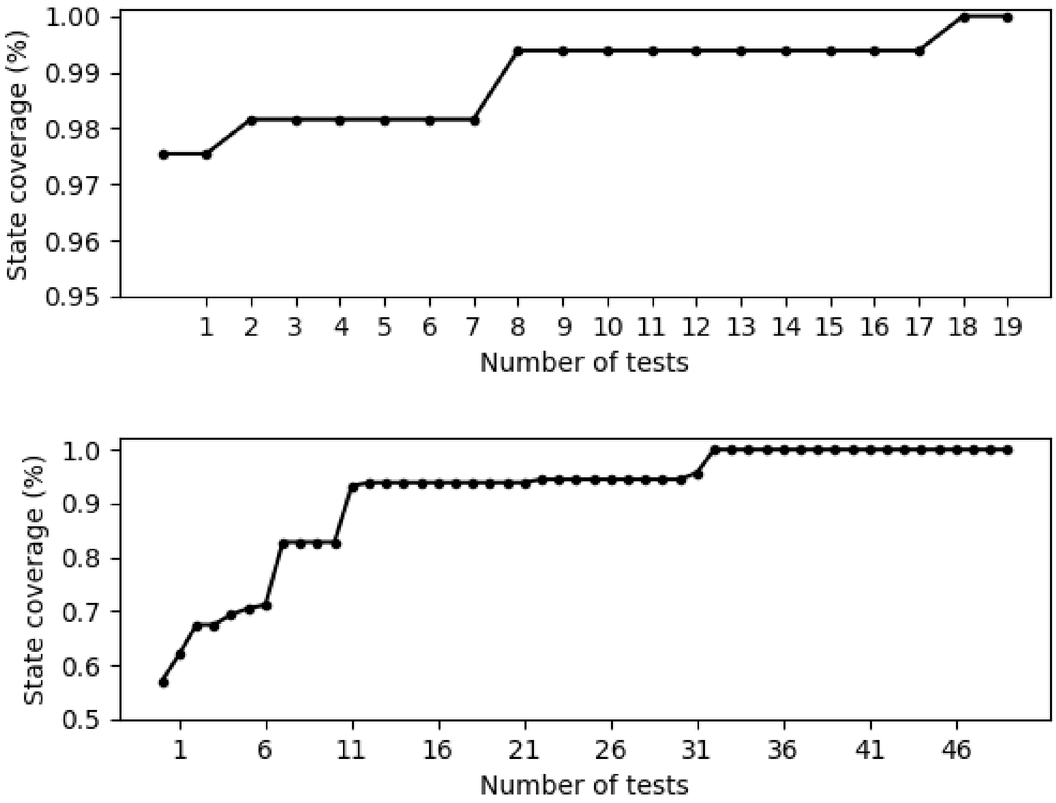


Figure 7.7: State coverage of the fuzzing programs for well-typed (above) and ill-typed (below) tests. With the random program generation procedure, we quickly convergence on full state coverage, even with a large number (160+) of states. Fuzzing revealed three states that were unreachable in the design and could be safely removed.

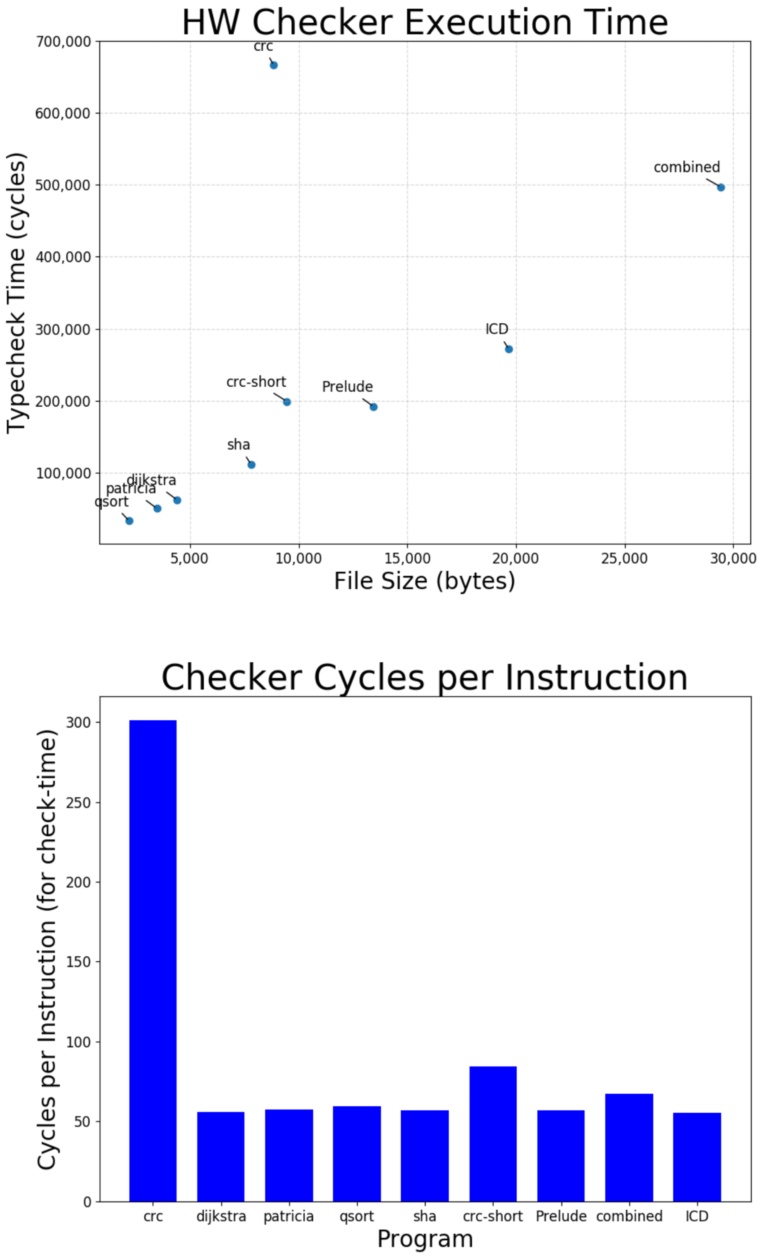


Figure 7.8: BEU evaluation for a set of sample programs drawn from MiBench, an embedded benchmark suite. For most programs, complete binary checking will take 150-160 cycles per instruction. LEFT: Time for hardware checker to complete, in cycles, as a function of the input program's file size. RIGHT: The same checking time, divided over the number of instructions in each program. Though the stock CRC32 has the longest typecheck time, an automatic procedure can modify the program to lower the checking time while preserving program semantics, noted as CRC-short.

struction to check. This was due to a single, large function that constructs the program's lookup table. The time to check a binary will scale roughly quadratically with function length. However, because function environments are pure and immutable, a simple automatic procedure is able to break the large function up into smaller ones, producing an equivalent program with a much better check time. Even if one chose not to automatically break up their long functions, a program of 100,000 lines with worst-case performance can still check in under one second.

To understand if real-world programs can be efficiently typed and checked with our system, we implement a subset of the benchmarks from MiBench [116]. These tended to be much longer and more complex programs when compared to the randomly-generated ones. While the fuzzer's programs averaged 50-65 instructions per program, the embedded benchmarks range from 500 to over 7,000 and represent code structures drawn from real-world applications, such as hashes, error detection, sorting, and IP lookup. In addition to the MiBench programs, a standard library of functions was checked, as well as a synthetic program combining all the other programs (to see the characteristics of longer programs).

Figure 7.8 shows how long typechecking took for the benchmark programs as a function of their code size. A linear trend is clearly visible for most of the programs, but one stands out from the pack: the CRC32 error detection function. The default CRC32 implementation is, in fact, a pathological case for our checking method as it is dominated by a single large function in the program. This function constructs a lookup table used elsewhere and is fully unrolled in the code. No other benchmark had a function nearly as large. The typecheck algorithm, while linear in program length (it checks in a single pass), is quadratic in function length and type complexity[5]. This insight not only explains the anomalous behavior of the initial CRC32 program, but provides a clear solution: break up the large function.

We test this hypothesis by breaking up CRC32 and re-checking it. While the task of break-

[5] "Type complexity" refers to how many base types are in a type; i.e., the length of its member types, recursively.

ing up a function in a traditional imperative programming language is complicated by the large amounts of global and implicit state, and would be even harder to perform at level of assembly, in a pure functional environment every piece of state is explicit. This makes the process not only easier, but even possible to fully automate. When we look at CRC32 specifically, the state, passed directly from one instruction to the next for table composition, can be captured in a single argument. We perform this transformation on our CRC32 program to break table construction across 26 single-argument functions, producing the CRC-short data point in the graphs in Figure 7.8. It still stands slightly above average because the table-construction functions are still above the average function length; recursively applying the breaking procedure could easily reduce the gap further.

While function length is an important aspect of checking time, with some care it can be effectively managed, and in the end all of the programs examined can be statically analyzed in hardware at a rate greater than 1 instruction per 100 cycles. This rate is more than fast enough to allow checking to happen at software-update-time, and could perhaps even be used at load-time, depending on the sensitivity of the application to startup latency.

7.7.2 Practical Application to an ICD

In addition to the benchmarks described above, we additionally provide results for a complete embedded medical application that was typed and checked; specifically, an ICD, or implantable cardioverter-defibrillator[6]. The ICD code was the largest single program examined (only the synthetic, combined program was larger). Its complexity required the use of multiple cooperating coroutines, managed by a small microkernel that handled scheduling and communication. Despite its length and complexity, it had the best typecheck characteristics of any of our test programs, with its cycles-per-instruction figure falling just below the average at 55.2.

[6] An ICD is a device planted in a patient's chest cavity, which monitors the heart for life-threatening arrhythmias. In the case one is detected, a series of pacing shocks are administered to the heart to restore a safe rhythm.

The process of adding types to the application was relatively simple, taking approximately 2 hours by hand.

Attempted Attack	**Result**
Binary that reads past the end of an object to access arbitrary memory	Hardware refuses to load binary due to type error ”field count mismatch”
Binary that passes an argument to a function of the wrong type to cause unexpected behavior	Hardware refuses to load binary due to type error ”not expected type”
Binary that writes past the end of an object to corrupt memory	Hardware refuses to load binary due to “application on non-function type”
Binary that passes too few arguments to a function to attempt to corrupt the stack	Hardware refuses to load binary due to ”undersaturated call”
Binary that uses an invalid branch head to try and make arbitrary jump	Hardware refuses to load binary due to type error ”branch type mismatch”
Binary that jumps past the end of a case statement to enable creation of ROP gadgets	Hardware refuses to load binary due to ”invalid branch target”
Jump past the end of a function to create ROP gadgets	Hardware refuses to load binary due ”invalid branch target”

Table 7.3: A list of some of the erroneous code that may be present in a binary (tested in our ICD application) and how the BEU identifies it as an error. Some of these errors, such as reading off the end of an object, writing beyond the end of an object, and jumping to arbitrary code points, are sufficient to thwart common attacks, like buffer overflow and ROP.

Since the ICD represents the largest and most complex program, as well as the exact type of program the BEU is designed to protect, we attempt to introduce a set of errors in the program to demonstrate the ability of the BEU to ensure integrity and security. Some of the errors are designed to crash the program; some are designed to hijack control flow; others are designed to read privileged data. The list of attempted attacks and how the BEU caught them are shown in Table 7.3.

In an unchecked system, passing an invalid function argument, writing past the end of an object, and passing an invalid number of function arguments could all lead to undefined behavior or system crashes. While past work could establish that a specific code would not do these things independent of the device, this work establishes these properties for the device itself, applying to all programs that can potentially execute — it is simply impossible to load a binary that will allow these errors to manifest. To establish that this was indeed the case, Table 7.3 shows the result of our attempts produce these behaviors: a type error, a function application error, and an undersaturated call error, respectively. Reading past the end of an object in an attempt to snoop privileged data was thwarted by detecting a type error dealing with field count mismatches. Control-flow hijacks, like using an invalid branch head, jumping past the end of a case statement, and jumping past the end of a function, were caught by a type mismatch in the first case and the detection of an invalid branch target in the latter two.

Though not exhaustive, these attacks show the resilience of the system to injected errors when compared to an unchecked alternative and demonstrate its practicality in the face of real errors and attempted attacks.

7.7.3 Bugs Found

The addition of types to the binaries and creation of many typed test cases revealed several bugs in existing programs, as well as the assembler. In typing the Zarf standard library of functions and data structures, four separate errors were found in recursive calls in the functions belonging to a set of data structures. In addition, a name shadowing bug was revealed when the standard library binary was typed: the assembler made a faulty assumption about the result of a name collision, so no error existed in higher-level code when the binary in fact contained a fatal error. This error would go undiagnosed without binary-level checking.

The randomly-generated programs in the fuzzing process found four assembler bugs. In one, instructions were omitted from the binary in rare situations, resulting in a runnable but incorrect program. In another, incorrect type variable numbers were emitted in cases where constructors of the same datatype used type variables in different orders in their declarations. In the third, the assembler crashed when a program used a function stored in a constructor field inside of nested case statements. The fourth occurred in only 0.2% of the generated tests, and had to do with needing different type variable numberings in different usages of the same constructor.

We note anecdotally the importance of a high number of test cases. In the final rounds of testing and debugging, there was a bug that up showed up in .009% of test cases — 18 out of 200k tests. With fewer than several thousand tests, it would easily have gone undetected.

7.7.4 Synthesis Results

Synthesized with Yosys, the hardware typechecker logic uses 21,285 cells (of which 829 are D Flip Flops, the equivalent of approximately 26 32-bit registers). Mapped to the open-source VSC 130nm library, it is .131 mm^2, with a clock rate of 140.8 MHz. Scaled to 32nm, it is approximately .0079 mm^2. As an addition to an embedded system or SoC, it provides only a tiny increase in chip area, and requires no power at run-time (having already checked the loaded program).

Assuming the checker can use the system memory, it requires no additional memory blocks; if not, it needs a memory space at least as large as the input binary type information, and space linear in the size of the program's functions.

The worst-case checking rate was 301 cycles per instruction for a pathological program; even a program of 450,000 lines with worst-case checking performance can be checked in under a second at the computed clock speed of 140 MHz on 130nm.

Chapter 8

Conclusion and Future Work

Critical systems are all around us. They help drive our cars, fly our planes, and monitor our hearts. For these types of systems, the actual costs of development can be dwarfed by the cost of verification. Unlike our traditional domains where power and performance are the only things we think about, correctness is a primary concern. Realizing when failure is intolerable and statically ensuring things behave as intended is key as we move forward.

Software analysis provides many avenues to help ensure various functionalities of a program, but only in binary analysis do we see all functionalities made apparent. Furthermore, for embedded systems, a high-level language and analysis may not be suitable — the software stack may not exist for the hardware platform, it may be too large for the limited resource, and it may be too slow. What is needed is a platform that is both binary-level (actual hardware) and easier to analyze.

As computing continues to automate and improve the control of life-critical systems, new techniques which ease the development of formally trustworthy systems are sorely needed. The system approach demonstrated in this work shows that deep and *composable* reasoning directly on machine instructions is possible when the architecture is amenable to such reasoning.

The range of and variance in processors and digital systems today is astounding. Modern cars have hundreds of microprocessors that handle everything from steering to monitoring tire pressure. The rise of “smart homes” means greater connectivity as more devices become IoT devices. Not only do many homes now come equipped with smart thermostats, but internet-enabled light bulbs are being sold today.

We don’t give much thought to each new device equipped with digital and internet functionality, but every new digital system has responsibilities — and room for failure. Each new connected device creates new attack surfaces. For example, hackers recently gained access to a casino’s systems by first hacking into a thermometer in their lobby’s fish tank [117], gaining access to their private network from there. Realizing that most security vulnerabilities are correctness problems, many attacks can be prevented by properly vetting code. Of course, as code becomes more and more complex, manually finding errors becomes increasingly challenging.

Zarf helps address both of these concerns. The pure, functional interface makes new and old automated and manual analyses easier and more tractable. Meanwhile, the automated hardware typechecker ensures that programs loaded onto light bulbs, fish tank thermometers, etc. are well-formed, crash-free, and more secure.

As system complexity grows, we need new methods of analysis to keep up with the explosion in software and functionality. If we do not make correctness and analyzability primary concerns, they will be left behind.

Despite Zarf being larger and slower than an optimized imperative microprocessor — for the lazy implementation, 2-4x slower on microbenchmarks and 20x slower in a worst-case analysis with GC on a real application, and approximately 2x larger — in an absolute sense, Zarf is still competitive. We see this in the real example ICD application: though the Zarf program takes much longer than the C version of the application, it still finishes in 180 μs on an FPGA — 25 time faster than the required 5 ms real-time deadline to operate at the necessary 200 Hz. Similarly, the entire Zarf lazy implementation logic fills only 7% of a common Artix-7 FPGA; other than power, there is no benefit to using fewer of the FPGA resources for a complete design, so we lose nothing by taking a few more percentage points of LUTs.

As we move to increasingly diverse systems on chip, heterogeneity in semantic complexity is an interesting new dimension to consider. The Zarf processor, along with it static analysis program checker, together make up only .405 mm^2 — less than half a square millimeter — even on the out-dated 130 nm tech node. A very small core supporting highly critical workloads might help ameliorate critical bugs, vulnerabilities, and/or excessive high-assurance costs. On a modern SoC, with billions of transistors and sub-20nm tech node, a core executing the Zarf ISA would take up roughly 0.006% of the total chip area.

8.1 Future Work

There are many interesting future directions to consider for the Zarf platform.

Multiprogramming

Zarf was designed as an embedded processor, meant to run a single, statically-loaded program. Embedded devices simply don't often have the resources or need to support context switching and multiple processes/programs/threads. However, to support a wider variety of applications, a larger Zarf processor that does support these things would be an interesting research direction. Many questions arise, however. How does inter-process communication work in a pure-functional environment? The hardware must manage context switches, since there is no architecturally visible state; how does software cooperate in this? Zarf is garbage collected (managed by the hardware), but what should GC look like in a multi-process environment? Can you collect one process at a time? What about shared objects? Should shared objects even be supported, or just serially-transmitted integers? Lastly, what sort of software paradigms work well with the Zarf multi-process abstraction? Where do we have to compromise the abstractions to make the paradigms possible?

Performant Hardware

Because the current incarnations of the Zarf hardware are research prototypes, there is little thought for performance in their designs. If we apply traditional architectural performance-optimizing techniques, what happens on Zarf? If we implement pipelining — or even out-of-order processing and superscalar dispatch — do we reap the same kind of performance benefits as on an imperative machine? Is the transformation for these optimizations as easy, or more challenging? To cooperate with these optimizations, perhaps the right direction is to implement Zarf as a micro-code decode layer on top of a traditional microprocessor; something like a "firmware runtime." If we can make Zarf execute functional code as fast an imperative processor executes imperative code, then it has the added benefit of being a "functional code accelerator."

Caching is a frequently-used architectural optimization. One clear advantage with an optimized version of Zarf is that there is no need for cache coherence protocols. Even in a shared, multi-core environment, because objects can never be mutated, once they are shared, they never change. Thus, cache lines never become dirty and never need emergency write-backs. This obviates the need for snoop-based cache architectures, greatly simplifying the cache design.

GC: Concurrent and Provably Correct

Garbage collection remains one of the most complex parts of the Zarf execution hardware. Many in the language run-time space would be uncomfortable with the idea of a hardware garbage collector, because different GC algorithms suit different needs; being able to swap out your garbage collector to one better suited for the current application is a clear advantage of software. However, there are two main enhancements we can make to the Zarf hardware garbage collector that make it a clearly better choice over software collection.

The first is to make GC fully concurrent. Concurrent GC has been the subject of much research, and is acknowledged to be a hard, messy problem. However, working on Zarf, most of the problems go away: the environment is pure, functional, and immutable. The key principle that makes immutability worthwhile in this context is that one can copy and relocate an object *while it is being used* and software cannot know the different. The GC engine can update addresses one at a time as it encounters them to point to the new location, and no piece of software can distinguish if it is using the new or the old copy.

The second enhancement is that, with a simple, gate-level GC implementation, we have the opportunity to prove GC functionally correct. There are two phases to achieving this: one is verifying the algorithm has the correct properties (i.e., it never collects a live object, and every dead objects is eventually collected); the other is verifying that our gate-level implementation implements the algorithm. The first challenge is the same whether we use software or hardware GC, but is made easier by the fact that we choose a simple GC algorithm (to make it amenable to hardware implementation). The second task is made possible by the low-level implementation of the algorithm. Normally, to verify an implementation correct, one needs a complete semantics for the language in which the implementation exists; these tend to be large, messy, and hard to work with (that's one of the main motives for using Zarf for software analysis). Our implementation, however, is at the gate level; if we have a verified algorithm in the form of a state machine, using Z3 [115] to verify that each state behaves correctly is just a set of logical propositions. Put another way, our "language" of implementation is logical gates, which have a small and clear semantics.

Verified Hardware

Zarf is meant as a platform for critical systems, to help alleviate the high cost of software assurance and verification. But the Zarf implementation itself is not beyond scrutiny. There is no standard at present for verified hardware, but if one were to verify the implementation of a hardware platform, Zarf is a clear choice. With a proof that the gate-level implementation is correct (i.e., correctly implements the operational semantics), when one verifies a property of their software on the Zarf semantics, that guarantee now extends down to the gate level. Verifying the Zarf platform, as opposed to an imperative core, has its challenges. The semantics presented by most imperative cores is low-level and poor for reasoning; it is, however, closer to what the hardware execution looks like. Thus, for a simple, single-cycle machine, the ISA spec gives a useful semantics to verify against. For Zarf, we have a high-level semantics, so we have to bridge the gap from Boolean logic to operational semantics. From the bottom, if we have a state machine implementation of the semantics, we can verify that a multicycle implementation of Zarf correctly implements the state machine with Z3 (as mentioned for verified GC). From the top, we can try to use a theorem prover like Coq to verify that the operational semantics can be equivalent to the state machine.

Compromising Purity

The abstractions to which Zarf adheres were chosen to maximize the analysis potential, without regard to their implementation difficulty or performance. This leaves a new avenue for optimization: compromise on some of the Zarf abstractions to enhance performance, potentially at the expense of some analyzability. For example, Zarf has no mutable arrays; Zarf programs must resort to using linked lists, binary trees, or other structures amenable to an immutable environment. Introducing a hardware-supported mutable array type could greatly speed up certain programs — such as our CRC32 program, which uses a lookup table extensively, or the SHA-1 and Dijkstra programs, which use mutable arrays. Running SHA-1 on one input, strict Zarf executes 7.65 million instructions; of these, 5.25 million (68%) were devoted to reading/writing binary trees. If those 5 million instructions could be replaced with built-in mutable arrays, SHA-1 (and other programs) would see a huge performance improvement. There are potentially other efficient compromises to be made; moving forward in this direction would require a fairly extensive survey of the possible compromises, and a large benchmark suite, in order to decide which are worthwhile. Gauging the cost to reasoning is much harder, but would also need to be done.

Compilation Tool-chain

Though not necessarily a “research” direction, a working compiler is essential for the health of the Zarf ecosystem. Having one enables other research: software surveys, new benchmarks, better productivity on the platform, etc. Haskell is the obvious choice for ease of use, but there are other interesting directions to take: a verified extraction from Coq, for example, would let you write a proof and then extract a executable Zarf program.

Compiling Zarf to Imperative

There are several opportunities to combine Zarf and non-Zarf code in interesting ways. One is using Zarf as an intermediate language — something like a “proof llvm” — where programs are safe, typed, and easy to reason about, but then they can be compiled to RISC-V, x86, or ARM for execution. The last compilation step could even be verified, but it would be difficult. Another interesting direction would be to compile to one of the aforementioned ISAs, then prove properties on the compiled code; for example, given a RISC-V binary that was compiled from a Zarf program, can we show strict non-interference between functions (i.e., that purity is respected)? It’s trivial to show at the Zarf level, but clearly valuable for an imperative program.

Sandboxing

This idea can take two forms: an imperative bubble within a Zarf environment, or Zarf bubbles within an imperative environment. Or, perhaps, even a shared core where imperative and Zarf code are “equal.” When the two operate in tandem, it is critical to show that the imperative code cannot mutate or modify things in the Zarf world (except via allowed communication channels). Even if the imperative code were not safe internally, it cannot be allowed to cause the Zarf code to be unsafe, or all our carefully crafted guarantees would collapse.

The first idea, of some imperative code in a Zarf environment, was partially implemented in first Zarf paper [6]. There, an imperative processor and a Zarf processor operated in tandem, but the Zarf processor was “in charge” and didn’t depend on the imperative one for operation. This allowed for safety problems, errors, and crashes on the imperative side, without affecting the critical Zarf code. In a broader sense, an imperative bubble would potentially allow one to factor away performance critical sections, or code that depends on imperative paradigms (like mutable arrays), into an imperative section of code within the broader Zarf program. However, because the imperative code is alone, it would be written in an essentially-functional style, so it would already be pretty similar to how a Zarf program would have handled the task.

The other idea, which is potentially more useful, is for Zarf sandboxes. This idea is related to the notion of Trusted Execution Environments (TEE), or “enclaves.” We have a mostly-imperative core, but a piece of code can be trusted, signed, and functional, which causes the processor to enter a sandboxed Zarf mode. The hardware provides the normal guarantees for isolation and integrity (as well as attestation), and the Zarf setting lets one reason about their critical components more easily. By cooperating with an imperative setting, however, the majority of the application does not have to be ported to Zarf. This set up (as with standard enclaves) would be vulnerable to denial-of-service attacks, but otherwise could provide an interesting and useful platform for partially-verified applications.

Appendix A

Progress and Preservation of Zarf Typesystem

Preservation

"Preservation" means that every step of computation preserves well-typedness. That is, if a term is well-typed, performing the next step of computation according to the semantic inference rules will result in a term that is also well-typed.

Lemma 1 (ApplyType)**.** *Calling the helper function* `applyType`$(\tau_i, \vec{\tau}_a, C, \alpha)$ *returns the principal type of an application of a function to zero or more arguments.*

Proof. `applyType` begins by generating a constraint for each parameter and argument until the list of arguments $\vec{\tau}_a$ is exhausted. Then it unifies these constraints to produce a substitution, which is used to generate the principal type of the application. As the algorithms of constraint generation, unification, and substitution are well-known and followed in the standard form in the machine, it suffices to look at standard proofs of principal types for proof of this lemma. The formalized algorithm for Zarf is found in Chapter 7.2.3. □

Lemma 2 (Preservation of `let`)**.** *If e is a* `let` *expression, $\vdash e : \tau$, and $e \rightarrow e'$, then $\vdash e' : \tau$.*

Proof. There are 11 possibilities for a well-formed `let` expression, corresponding to the 11 small-step inference rules.

First we handle the special case of applying on an integer. According to the inference rule LET-PRIMINT, we map x to n and proceed with the next expression. The typing rule LET-INT says that this addition to Γ is sufficient to type the next expression.

LET-SAT handles application on a static function with the right number of arguments. According to the LET-VAR typing rule, the types of the given sequence of arguments must match the expected sequence. With those values in Γ', we have all of the state (along with Σ, which does not change) that can be references at the start of e'. Since those arguments are guaranteed to be the correct types, by induction e' will be typable.

LET-UNDERSAT handles application on a static function with too few arguments given. As before, `applyType` in LET-VAR guarantees that the arguments are well-typed and the closure we create is well-formed. According to LET-VAR, the addition of the closure to Γ is all that is required to type e.

LET-OVERSAT handles application on a static function with too many arguments given. The proof for LET-SAT applies here: mapping our arguments to Γ is sufficient to show, by induction, that e' is typable. However, we also note that the screening of `applyType` further guarantees that the arguments placed in the argsK continuation will be consumed by the return value of f.

LET-PRIM performs a primitive operation and maps x to the result in Γ. LET-VAR tells us this addition to Γ is sufficient to type e, and `applyType` guarantees that we received the expected number of integer arguments for the primitive operation.

LET-CON proceeds the same as LET-UNDERSAT, but instead of building a closure, we build a constructor (which `applyType` guarantees is well-typed). Inductively, we can type e.

The rules LETCLO-PRIM, LETCLO-SAT, LETCLO-UNDERSAT, LETCLO-OVERSAT, and LETCLO-CON follow similar proofs, but need to deal with the closure being read. As noted in the proof for LET-UNDERSAT, `applyType` on the previous application will guarantee the closure is well-formed.

applyType on this application guarantees that the additional arguments are of the expected type as well. Concatenating the closure's arguments with the one's provided will either result in a sequence too short, too long, or exactly right; the rules cover all three cases. □

Lemma 3 (Preservation of case)**.** *If e is a* case *expression,* $\vdash e : \tau$*, and* $e \rightarrow e'$*, then* $\vdash e' : \tau$*.*

Proof. There are three inference rules that deal with the execution of case instructions.

CASE-PAT handles matches against constructors. According to the CASE-CON typing rule, allConsPres must be satisfied for the case instruction to be well-typed, so we know we have a match in the sequence of branches. In addition, branchTypes will guarantee that each branch is well-typed, requiring the mapping of the field names to the dynamic constructor field types in Γ; therefore, mapping each field name to the appropriate constructor field in CASE-PAT will allow the expression to be typable. branchTypes further guarantees that each branch terminates in a result instruction of the same principal type as this function's return type (as guaranteed by principalType). Thus, we have as a premise that, no matter which branch matches, the expression for the branch will can be inductively typed and return a value of the appropriate type.

CASE-LIT handles matches against integers where we did find a match. As a premise in the CASE-INT typing rule, each branch's expression e_i is typable as τ_i and the principal type (as checked by principalType) of the sequence $\overrightarrow{\tau_i}$ is τ, which must match the return type of this function. Therefore, we can inductively type the matching branch's expression e.

CASE-LIT-ELSE is similar, handling matches against integers where we did not find a match. The proof is the same as for CASE-LIT, noting that τ_e, the type of the else branch, is also checked by principalType. □

Lemma 4 (Preservation of result)**.** *If e is a* result *expression,* $\vdash e : \tau$*, and* $e \rightarrow e'$*, then* $\vdash e' : \tau$*.*

Proof. The inference rules RESULT-VAR and RESULT-CLO handle `result` instructions. They are typed rather trivially by the RESULT typing rule, which just looks up the returned type. In Lemma 8, we make a proof that the continuation stack is appropriately aligned for the two possible inference rules, which each expect something different on top of the stack. On top of that, we look at the typing rule LET-VAR, which says that typing the next expression is made possible by mapping the given name to the result of the `let` call. With the `result` instruction, we finish the `let` call and resume execution with the saved next expression. RESULT-VAR contains the explicit premise that we map the saved name to the result of the call (the value being returned). Therefore, RESULT-VAR respects preservation. RESULT-CLO creates a new pair of instructions to handle the various cases of under/oversaturation for its closure call. However (as covered in Lemma 8), we know the `let` instruction that first spawned this call and created the closure was well-typed. This means the closure has well-formed arguments in it, and the additional arguments being fed in are expected and also well-formed. Therefore, the `let` expression created is well-formed. We additionally know that the `result` instruction is well-typed, because the original `let` operation was well-typed; according to LET-VAR, it expected a type of τ_1 as the result of the operation. Our new variable z is strictly a refinement of the original application; by induction, its refined version will have the same type and returning z is correct. □

Theorem 5 (Preservation of Expressions)**.** *If* $\vdash e : \tau$ *and* $e \to e'$*, then* $\vdash e' : \tau$*.*

Proof. By Lemmas 2, 3, and 4, it is true by completeness. □

Progress

“Progress” means that well-typed terms do not get stuck. This means that if a term is well-typed, it is either a value, or there is a valid step of computation it can take — where a “valid step” is one of our small-step inference rules (Figures 3.5 and 3.6). We will refer to those and the typing inference rules from Figure 7.1.

Lemma 6 (Progress of `let`)**.** *If e is a* `let` *expression and* $\vdash e : \tau$*, then* $e \rightarrow e'$*, for some* e'*.*

Proof. There are 11 possibilities for a well-formed `let` expression, corresponding to the 11 small-step inference rules.

First we handle the special case of applying on an integer. This is typed by the typing rule LET-INT. The expression is well-typed, so no arguments can be passed in. LET-PRIMINT says we map x to n and, trivially, continue execution with e.

If applying on a function and the right number of arguments are passed, LET-SAT is used. We're applying on a static function, which by construction must be in Σ, giving us the expression e'. We continue execution with e'.

If applying on a function and too few arguments are passed, LET-UNDERSAT is used. A closure is constructed to hold f and the indicated arguments. Execution proceeds trivially with the next expression e.

If applying on a function and too many arguments are passed, LET-OVERSAT is used. f is used to look up e' as before. The excess arguments are saved on the continuation stack. We continue execution with e'.

The previous three cases all have version where on applies on a closure (LETCLO-SAT, LETCLO-UNDERSAT, and LETCLO-OVERSAT). The proofs for these proceed in the same way, except that some of the arguments are drawn from the closure. By the premise that the application was well-typed, these arguments are also well-typed, and the proof proceeds the same.

If applying the correct number of arguments to a primitive operation, LET-PRIM is used. The result of the operation is stored in the environemnt and execution proceeds trivially with the next expression e.

If applying the correct number of arguments to a constructor, LET-CON is used. The constructor is built and saved in the environment, and execution proceeds with the next expression e.

As with before, there is a version of the previous two expressions where on applies on a closure instead of a function. As before, the action is only different in drawing some arguments from a closure and some from the expression; the proof proceeds the same. This covers LETCLO-PRIM and LETCLO-CON.

These 11 possibilities form a complete set. Any other action in a `let` expression would be ill-typed. Thus, if well-typed, the `let` expression can always make progress. □

Lemma 7 (Progress of `case`). *If e is a* `case` *expression and $\vdash e : \tau$, then $e \rightarrow e'$, for some e'.*

Proof. If casing on an integer, according to our typing rule CASE-INT, x must map to an integer in the environment. Then, either CASE-LIT or CASE-LIT-ELSE will be used for evaluation. If CASE-LIT, it is a premise that one branch matches, and thus execution continues with the corresponding expression. Otherwise, CASE-LIT-ELSE says that there was no match, and we continue execution with the expression from the `else` clause.

If casing on a constructor (the only other options, as casing on a closure is ill-typed), the rule CASE-PAT is used. According to our CASE-CON typing rule, it is a premise of being well-typed that all constructors are present (checked by `allConsPres`). This means that the function f to which the constructor x references must have a match someone in the sequence of branches. We choose the matching expression, and continue execution. □

Lemma 8 (Progress of `result`). *If e is a* `result` *expression and $\vdash e : \tau$, then $e \rightarrow e'$, for some e'.*

Proof. First we consider the RESULT-VAR inference rule, where an integer or constructor is returned. Either the continuation stack is empty, and execution halts, or there is a continuation on the stack. Since we are returning a value, not a closure which expects additional arguments, we know via the LET-VAR typing rule that the `let` expression to which we are returning had no excess arguments — otherwise it would have been ill-typed. Since it did not have excess arguments, the continuation on the stack could not have been placed there by the LET-OVERSAT (or LETCLO-OVERSAT rule. Similarly, the LET-UNDERSAT (and LETCLO-UNDERSAT) rule does not place a continuation on the stack. The only option remaining is that the continuation on the stack was placed by LET-SAT or LETCLO-SAT. In that case, it is a letK and not an argsK. The letK continuation contains an expression e, which we use to continue execution.

For the RESULT-CLO rule, there must be an argsK continuation on top of the continuation stack. This follows again from the typing rule LET-VAR: we are returning a closure expecting additional arguments, so to be well-typed those arguments must have been passed in and processed by either LET-OVERSAT or LETCLO-OVERSAT. This means an argsK continuation must be on the stack. This satisfies our premise, and we can continue execution with the indicated `let` and `result` expressions. □

Lemma 9 (Progress of Expressions)**.** *If* $\vdash e : \tau$*, then either* e *is a value or else* $e \rightarrow e'$*, for some* e'*.*

Proof. By Lemmas 6, 7, and 8, it is true by completeness. □

Lemma 10 (Progress of Functions)**.** *Assuming the correct arguments are given, executing the body of a well typed function* $\textbf{fun}\, fn\ \overrightarrow{x : \tau}\ \tau\ \textbf{=}\ e$ *produces a value of type* τ*, when the body terminates.*

Proof. By rule FUNC-PARAMS, execution of the function body e begins in an environment Γ where each parameter in the list $\vec{x}$ is mapped to those parameters' declared types $\vec{\tau}$. Any type variables in those parameter types and the return type are replaced with fresh rigid type variables, implying that those parameters cannot be specialized within the function body. Thus, type variables declared in the parameters are universally quantified across the entire function. Rule FUNC-RET follows similarly, except that the initial environment used to type check the body e is empty. We will use Γ to represent the starting environment in both cases.

We must show that $\Gamma \vdash e : \tau$, that is, that the function body evaluates to a non-error value of type τ. By Lemma 9, e is either a value or $e \rightarrow e'$. Because the function was well-typed, `result` instructions cannot return non-value expressions. □

Theorem 11 (Progress of Programs)**.** *Let P be a well typed program composed of a list of datatypes $\overrightarrow{data}$ and functions $\overrightarrow{func}$. Let (**fun** `main` $\overrightarrow{x : \tau}$ τ = e) $\in \overrightarrow{func}$ be the entry point to P where execution begins. Then P either halts and returns a value of type $\tau \neq$ `Error`, or it continues execution indefinitely.*

Proof. By Lemma 10 and rule FUNC-PARAMS, we know that **fun** `main` $\overrightarrow{x : \tau}$ τ = e has type τ (similarly for functions without parameters, using rule FUNC-RET). Since a hardware error value of type `Error` is created when the machine encounters an invalid state during evaluation, and Lemma 10 says that a well typed function does not lead to an invalid state, P returns a value of type $\tau \neq$ `Error` when it terminates. □

Appendix B

Integrity Typing

The material in this appendix was done in collaboration with Michael Christensen, but the majority is the result of his efforts. It appears with permission, and is included in this thesis for completeness of the work done on the Zarf platform.

B.1 Typesystem

The integrity type system is a set of typing rules that can be applied to an annotated Zarf assembly program, which will tell if the program is well-typed with regard to the system. If it is well-typed, we know that we have integrity of the indicated values; more preciesely, we know that no value of integrity level trusted (**T**) is ever influenced anywhere in the program by a value of integrity level untrusted (**U**).

The integrity typesystem rules can be found in Figure B.1, the joining type rules in Figure B.2, and the subtyping rules in Figure B.3.

$$\ell, \textsc{pc} \in \mathit{Label} ::= \textbf{T} \mid \textbf{U} \quad \tau \in \mathit{Type} ::= \textbf{num}^{\ell} \mid (cn, \vec{\tau}) \mid (\vec{\tau} \to \tau) \quad \Gamma \in \mathit{Env} = \mathit{Identifier} \to \mathit{Type}$$

$$\frac{\tau_1 = \Gamma(id) \quad \overrightarrow{\tau_2} = \Gamma(\overrightarrow{arg}) \quad \tau_3 = \texttt{applyType}(\tau_1, \overrightarrow{\tau_2}) \quad \Gamma[x \mapsto \tau_3] \vdash e : \tau_4}{\Gamma \vdash \textbf{let } x \textbf{ = } id\ \overrightarrow{arg} \textbf{ in } e : \tau_4}\ (\textsc{let})$$

$$\frac{(cn, \vec{\tau}_1) = \Gamma(arg) \quad (cn\ \vec{x}\ \Rightarrow\ e_1) \in \overrightarrow{br} \quad \Gamma[\vec{x} \mapsto \vec{\tau}_1] \vdash e_1 : \tau_2}{\Gamma \vdash \textbf{case } arg \textbf{ of } \overrightarrow{br} \textbf{ else } e_0 : \tau_2}\ (\textsc{case-cons})$$

$$\frac{\textbf{num}^{\ell} = \Gamma(arg) \quad \Gamma \vdash e_1 : \tau_1\ \ldots\ \Gamma \vdash e_n : \tau_n \quad \Gamma \vdash e_0 : \tau_0 \quad \tau = (\tau_0 \sqcup \tau_1 \sqcup \ldots \sqcup \tau_n) \bullet \ell}{\Gamma \vdash \textbf{case } arg \textbf{ of } n_1 \Rightarrow e_1\ \ldots\ n_n \Rightarrow e_n \textbf{ else } e_0 : \tau}\ (\textsc{case-lit})$$

$$\frac{\Gamma[x \mapsto \textbf{num}^{\textbf{T}}] \vdash e : \tau}{\Gamma \vdash \textbf{let } x \textbf{ = getint } n \textbf{ in } e : \tau}\ (\textsc{getint}) \qquad \frac{\textbf{num}^{\ell} = \Gamma(arg) \quad \Gamma[x \mapsto \textbf{num}^{\ell}] \vdash e : \tau}{\Gamma \vdash \textbf{let } x \textbf{ = putint } n\ arg \textbf{ in } e : \tau}\ (\textsc{putint})$$

$$\frac{\tau = \Gamma(arg)}{\Gamma \vdash \textbf{result } arg : \tau}\ (\textsc{result})$$

$$\texttt{applyType}(\vec{\tau}_1 \to \tau)\vec{\tau}_2 = \begin{cases} \tau & \text{if } |\vec{\tau}_1| = 0 \text{ and } |\vec{\tau}_2| = 0 \\ (\vec{\tau}_1 \to \tau) & \text{if } |\vec{\tau}_1| > 0 \text{ and } |\vec{\tau}_2| = 0 \\ \texttt{applyType}\vec{\tau}_3 \to \tau\vec{\tau}_4 & \text{if } \vec{\tau}_1 = \tau_1 :: \vec{\tau}_3,\ \vec{\tau}_2 = \tau_2 :: \vec{\tau}_4, \text{ and } \tau_2 \leq \tau_1 \\ \texttt{applyType}\vec{\tau}_3 \to \tau_4\vec{\tau}_2 & \text{if } |\vec{\tau}_1| = 0,\ |\vec{\tau}_2| > 0, \text{ and } \tau = (\vec{\tau}_3 \to \tau_4) \end{cases}$$

$$\Gamma(\mathrm{n})\ = \textbf{num}^{\textsc{pc}} \quad \Gamma(id\,) \ = \begin{cases} (\vec{\tau} \to (cn, \vec{\tau})) & \text{if } id = cn \text{ and } \textbf{con}\ cn\ \vec{x}{:}\vec{\tau} \in \overrightarrow{decl} \\ (\vec{\tau} \to \tau) & \text{if } id = fn \text{ and } \textbf{fun}\ fn\ \vec{x}{:}\vec{\tau}\ {:}\,\tau \textbf{ = } e \in \overrightarrow{decl} \\ \tau & \text{if } id = x \text{ and } (x \mapsto \tau) \in \Gamma \end{cases}$$

Figure B.1: Integrity Typing Rules. A type is inductively defined as either a labeled number, a singleton constructor, or a function constructed of these types. The type environment maps variables, function, and constructor names to types. Since all functions are annotated with their types, type checking proceeds by ensuring that the return type of a function is the same as the type deduced by checking the function's body expression with the function's parameter types added to the type environment. ⊔ denotes the join of two types, and • denotes the joining of a type's integrity label with another. Γ is a helper function that gets the type of an argument, and `applyType` applies a function type to argument types. Applying a helper function that takes one argument to a list of arguments is shorthand for mapping that function over the list.

$$\textbf{num}^{\ell} \sqcup \textbf{num}^{\ell'} = \textbf{num}^{\ell \sqcup \ell'}$$
$$(cn, \vec{\tau}) \sqcup (cn, \vec{\tau}) = (cn, \vec{\tau})$$
$$(\vec{\tau} \rightarrow \tau) \sqcup (\vec{\tau'} \rightarrow \tau') = (\vec{\tau} \sqcup \vec{\tau'} \rightarrow \tau \sqcup \tau')$$
$$\textbf{num}^{\ell} \bullet \ell' = \textbf{num}^{\ell \sqcup \ell'}$$

Figure B.2: Joining Two Types. The • operator is used to join a type's label with another label; if the type that the label is being joined with is not a **num**, the label will be joined with each of the type's inner types until a base **num** is reached. Joining two lists of types is equal to the pairwise join of their elements. Constructor join is trivial because constructors are singletons whose type never changes, and only equal constructors can be compared.

$$\frac{\ell \sqsubseteq \ell'}{\textbf{num}^{\ell} \leq \textbf{num}^{\ell'}}\ (\textsc{num}) \qquad \frac{\vec{\tau} \leq \vec{\tau'} \quad \tau \leq \tau'}{(\vec{\tau} \rightarrow \tau) \leq (\vec{\tau'} \rightarrow \tau')}\ (\textsc{func})$$

$$\frac{}{(cn, \vec{\tau}) \leq (cn, \vec{\tau})}\ (\textsc{cons}) \qquad \frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ (\textsc{tran})$$

Figure B.3: Subtyping Rules. One type is a subtype of another if their base types are equal and, in the case of the base num type, the first's label is lower in the integrity lattice than the other. A list is a subtype of another if pairwise each element of the first is a subtype of the corresponding element in the other list.

B.2 Proof of Soundness

The soundness proof of the integrity type system proceeds by cases on the three forms of expressions.

Lemma 12 (Case Expression Soundness)**.** *If* $\forall\ x\ e_0\ e_1\ arg_1\ arg_2\ \tau_0\ \tau_1\ cn_1\ \vec{\tau}_1\ v_1\ v_2\ \rho\ \Gamma$,

1. e_0 **= (*case*** arg_1 ***of*** $\overrightarrow{br}$ ***else*** e_1*)*

2. $\Gamma \vdash e_0 : \tau_0 \ \wedge\ \rho \vdash e_0 \Downarrow v_1 \ \wedge\ \rho\,(x)$ **=** arg_1

3. $\Gamma \vdash arg_2 : \tau_1 \ \wedge\ \tau_1 > \tau_0 \wedge \rho \vdash arg_2 \Downarrow\ v_2$

then $\rho[x \mapsto v_2] \vdash e_0 \Downarrow v_1$.

Proof. By cases on arg_1:

1. If $\Gamma \vdash arg_1 : \mathbf{num}^{\ell}$, then $\forall\ n\ e_2\ \ldots\ e_m\ , e_0\ =\ ($**case** n **of** $n_2 \Rightarrow e_2\ \ldots\ n_m \Rightarrow e_m$ **else** $e_1)$, and either $\ell = \mathbf{T}$ or $\ell = \mathbf{U}$. We show that regardless of arg_2's level when it is of type **num**, it cannot be changed and therefore e_0's value doesn't change.

   (a) If $\exists\ \ell_1\ \in \tau_0$ s.t. $\ell_1 = \mathbf{T}$, then by typing rule CASE-LIT and the rule for join, n's integrity label is **T**. Therefore, arg_1 cannot both equal n and be arbitrarily changed to some expression arg_2 because it is not an expression whose type label is less trusted than the type of the entire expression (i.e. $\mathbf{num}^{\mathbf{T}} \not\geq \tau_0$). Thus we cannot replace arg_1 with arg_2, so in this case the value of e_0 remains the same, as desired. Since e_1 through e_{m+1} are expressions whose soundness with respect to the type system can be considered separately through Lemmas 12, 13, and 14, we do not consider them here.

   (b) If $\exists\ \ell_1\ \in \tau_0$ s.t. $\ell_1 = \mathbf{U}$, then by our definition of the $\mathbf{T} - \mathbf{U}$ integrity lattice, there can be no values whose type is greater than τ_0 (arg_1 included) that we can change. Therefore, e_0 remains unchanged, satisfying our conclusion.

2. If $\Gamma \vdash arg_1 : (cn, \vec{\tau_1})$, then $\forall\, cn_3\ \ldots\ cn_n\ \vec{x_3}\ \ldots\ \vec{x_n}\ e_3\ \ldots\ e_n, e_0 = (\textbf{case}\ (cn, \vec{\tau_1})\ \textbf{of}\ cn_3\ \vec{x_3} \Rightarrow e_3\ \ldots\ cn_n\ \vec{x_n} \Rightarrow e_n\ \textbf{else}\ e_1)$.

    - We know by the operational semantics (restricted to accommodate this type system, with singleton constructor types) that which branch we case on is determined entirely by the constructor that arg_1 evaluates to, and **not** the values contained within that constructor. Therefore, changing the expressions within any constructor will result in the same branch being taken, such that e_0 evaluates to the branch's right-hand-side expression. Therefore, we cannot choose to replace arg_1 with another arbitrary arg_2 when $\Gamma \vdash arg_1 : (cn, \vec{\tau_1})$.
    - Let $(cn_3\ \vec{x_3} \Rightarrow e_3)$ be the matching branch (where $cn = cn_3$). Based on the previous bullet point, we know that changing the expressions of any other branches will not change the value of the entire case expression, so we focus on this particular branch as an example. We must show that $\forall \tau_3,\ \exists\ x_3 \in \vec{x_3}$ s.t. $\Gamma \vdash x_3 : \tau_3 > \tau_0$, changing the value that x_3 maps to in ρ does not change the value that e_3 evaluates to; that is, $\rho[x_3 \mapsto v_3] : e_3 \Downarrow v$, where $\rho(arg_2) = v_3$. Since e_3 is an expression, its soundness is either covered by Lemma 12 (by induction) or Lemmas 13 or 14.

□

Lemma 13 (Return Expression Soundness)**.** *If* $\forall\ x\ e_0\ arg_1\ arg_2\ \tau_1\ \tau_2\ v_1\ \rho\ \Gamma$,

1. $e_0\ =\ (\textbf{result}\ arg_1) \wedge \rho \vdash e_0 \Downarrow v_1$
2. $\Gamma \vdash arg_1 : \tau_1 \wedge arg_2 = \rho(arg_1)$
3. $\rho \vdash arg_2 : \tau_2 \wedge \tau_2 > \tau_1 \wedge \rho \vdash arg_2 \Downarrow v_2$

then $\rho[(x \mapsto v_2)] \vdash e_0 \Downarrow v_1$

Proof. The **result** expression is used for wrapping a value into a single expression containing that value. Therefore, changing the value of arg_1 to arg_2 would change the resultant value v_1 that e_0 is given, contradicting our result. As another point, by the typing rule RESULT, **result**'s type is precisely the type of arg_1, meaning there are no values within e_0 to change that would not cause us to violate (3) above. Therefore, the value of arg_1 must equal the value of arg_2 such that value of e_0 cannot change. □

Lemma 14 (Let Expression Soundness)**.** *If* $\forall\ e_0\ e_1\ x\ id\ arg_1\ arg_2\ \overrightarrow{arg_3}\ \overrightarrow{arg}_4\ \tau_1\ \tau_2\ v_1\ v_2\ v_3\ \vec{v}_3\ \vec{v}_4\ v_5\ \rho\ \Gamma$,

1. $e_0 \;=\; (\textbf{let}\ x\ \textbf{=}\ id\ \overrightarrow{arg}_3\ \textbf{in}\ e_1)$
2. $\Gamma \vdash e_0 : \tau_1 \;\wedge\; \rho \vdash e_0 \Downarrow v_1$
3. $\Gamma \vdash id : \vec{\tau} \to \tau$
4. $\rho\,(\overrightarrow{arg}_3) = \vec{v}_3$
5. $\Gamma\,(arg_1) = \tau_2 \;\wedge\; \tau_2 > \tau_1$
6. $arg_0 \in \overrightarrow{arg}_3 \;\wedge\; \overrightarrow{arg}_4 = \overrightarrow{arg}_3 - arg_0 + arg_1$
7. $\rho\,(\overrightarrow{arg}_4) = \vec{v}_4$
8. $(id \in \overrightarrow{cons} \wedge \mathtt{applyCn}(id, \vec{v}_3) = v_2) \vee (\mathtt{applyFn}(id, \vec{v}_3,\ ) = v_2)$
9. $(id \in \overrightarrow{cons} \wedge \mathtt{applyCn}(id, \vec{v}_4) = v_3) \vee (\mathtt{applyFn}(id, \vec{v}_4,\ ) = v_3)$
10. $v_2 = v_3$
11. $\rho[x \mapsto v_3] \vdash e_1 \Downarrow v_5$

then $v_1 = v_5$.

Proof. By cases on id:

1. If *id* is a primitive function (add, multiply, etc.), then $v_2 \neq v_3 \iff \overrightarrow{arg}_3 \neq \overrightarrow{arg}_4$. By the typing rule of primitives, the type τ that the function returns is the least upper bound of all of its arguments, including arg_1, meaning by definition, both the value and type of the primitive operation are entirely dependent on all arguments. Therefore, there cannot exist an arg_2 that allows us to substitute it for arg_1 whose type is less trusted than τ without changing the entire value v_1.

2. If *id* is a constructor, then *id* has the type $\overrightarrow{\tau} \rightarrow (cn, \vec{\tau})$. *id*'s return type is determined statically and does not change throughout program execution. Therefore, there does not exist a subexpression in $\overrightarrow{arg}_3$, or more generally, in e_0, that can changed without changing the type of the constructor, which would contradict our having the same values after evaluation.

3. If *id* is a non-recursive function composed solely of case and result expressions and applications of primitive functions and constructors used in let expressions, then by (1), (2), Lemmas 12 and 13 and induction on Lemma 14, we know *id* must be sound. By extension, if *id* calls a function that fulfills these requirements, one can unfold the called function's contents in order to see that the resultant value v_2 satisfies this case.

4. If *id* is a recursive function or calls a function which calls *id* (i.e. mutual recursion), it is possible that the function call never terminates and therefore never results in a single value. The soundness of e_0 must then be guaranteed via induction on possible expressions, proven in the previous lemmas. We know statically that the type of *id* is of the form $\vec{\tau} \rightarrow \tau$, so we are guaranteed via simplification rules in the `apply` helper functions that types of $\overrightarrow{arg}_3$ must be equal to or subtypes of $\vec{\tau}$, or otherwise our operational semantics would get stuck. By induction, any recursive calls made in e_1 must also satisfy this lemma, meaning that the actual arguments $\overrightarrow{arg}_3$ are used properly, otherwise e_0 wouldn't type check to type τ_1 by getting stuck.

By proving that v_2's value does not change when less-trusted values change, we can safely continue with the evaluation of e_1, which will be a **case**, **result**, or **let**, all of which are handled in Lemma 12, Lemma 13, and Lemma 14, respectively. □

Theorem 15 (Integrity Type System Soundness)**.** *Our integrity type system is sound if, given some expression* ***e*** *of type τ which evaluates to some value* ***v****, we can show that we can arbitrarily change any (or all) expressions in* ***e*** *which are less trusted than τ so that* ***e*** *still evaluates to* ***v****; i.e., untrusted data does not affect trusted data.*

Formally, if $\forall e_1\ e_2\ e_3\ e_4\ \tau_1\ \tau_2\ \tau_3\ v\ \rho\ \Gamma$,

1. $\rho \vdash e_1 \Downarrow v \ \wedge\ \Gamma \vdash e_1 : \tau_1$

2. $e_2 \in \texttt{subexprs(}e_1\texttt{)} \ \wedge\ \Gamma \vdash e_2 : \tau_2 \ \wedge\ \tau_2 > \tau_1$

3. $\Gamma \vdash e_3 : \tau_3 \ \wedge\ \tau_3 \geq \tau_2$

then $\rho \vdash e_1[e_3/e_2] \Downarrow v$

Proof. There are just three types of expressions: **let**, **case**, and **result**. By Lemma 12, we show that **case** expressions (the vehicle for control-flow) are sound. By Lemma 13, we show that **result** expressions are sound. Likewise, by Lemma 14, we show that **let** expressions (the vehicle for function application) are sound. Thus, we have exhaustively shown soundness of all expressions. Furthermore, we can see that when these expressions are composed according to the abstract syntax, with the additional typing annotations and a few restrictions, any well-typed Zarf program has the property of non-interference with respect to integrity. □

Bibliography

[1] A. Fox and M. O. Myreen, *A trustworthy monadic formalization of the armv7 instruction set architecture*, in *Proceedings of the First International Conference on Interactive Theorem Proving*, ITP'10, (Berlin, Heidelberg), pp. 243–258, Springer-Verlag, 2010, DOI.

[2] G. M. Amdahl, G. A. Blaauw and F. P. Brooks, *Architecture of the ibm system/360*, *IBM Journal of Research and Development* **8** (1964) 87.

[3] A. Kennedy, N. Benton, J. B. Jensen and P.-E. Dagand, *Coq: the world's best macro assembler?*, in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, pp. 13–24, ACM, 2013.

[4] A. Chlipala, *Mostly-automated verification of low-level programs in computational separation logic*, in *ACM sigplan notices*, vol. 46, pp. 234–245, ACM, 2011.

[5] http://lambda-the-ultimate.org/node/1633.

[6] J. McMahan, M. Christensen, L. Nichols, J. Roesch, S.-Y. Guo, B. Hardekopf et al., *An architecture supporting formal and compositional binary analysis*, in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), pp. 177–191, ACM, 2017, DOI.

[7] J. McMahan, M. Christensen, K. Dewey, B. Hardekopf and T. Sherwood, *Bouncer: Static program analysis in hardware*, in *International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), ACM, 2019.

[8] N. Heintze and J. G. Riecke, *The slam calculus: Programming with secrecy and integrity*, in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, (New York, NY, USA), pp. 365–377, ACM, 1998, DOI.

[9] M. Abadi, A. Banerjee, N. Heintze and J. G. Riecke, *A core calculus of dependency*, in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, (New York, NY, USA), pp. 147–160, ACM, 1999, DOI.

[10] D. Volpano, C. Irvine and G. Smith, *A sound type system for secure flow analysis*, *J. Comput. Secur.* **4** (1996) 167.

[11] D. E. Denning and P. J. Denning, *Certification of programs for secure information flow*, *Commun. ACM* **20** (1977) 504.

[12] J. A. Goguen and J. Meseguer, *Security policies and security models*, in *Security and Privacy, 1982 IEEE Symposium on*, pp. 11–11, April, 1982, DOI.

[13] F. Pottier and V. Simonet, *Information flow inference for ml*, *ACM Trans. Program. Lang. Syst.* **25** (2003) 117.

[14] A. Sabelfeld and A. C. Myers, *Language-based information-flow security*, *IEEE J.Sel. A. Commun.* **21** (2006) 5.

[15] D. Terei, S. Marlow, S. Peyton Jones and D. Mazières, *Safe haskell*, in *Proceedings of the 2012 Haskell Symposium*, Haskell '12, (New York, NY, USA), pp. 137–148, ACM, 2012, DOI.

[16] D. Yu, N. A. Hamid and Z. Shao, *Building certified libraries for pcc: dynamic storage allocation*, in *Proceedings of the 12th European conference on Programming*, pp. 363–379, Springer-Verlag, 2003.

[17] A. Chlipala, *Mostly-automated verification of low-level programs in computational separation logic*, in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 234–245, ACM, 2011, DOI.

[18] R. S. Boyer and Y. Yu, *Automated correctness proofs of machine code programs for a commercial microprocessor*, in *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, (London, UK, UK), pp. 416–430, Springer-Verlag, 1992, http://dl.acm.org/citation.cfm?id=648230.752650.

[19] N. G. Michael and A. W. Appel, *Machine instruction syntax and semantics in higher order logic*, in *Proceedings of the 17th International Conference on Automated Deduction*, CADE-17, (London, UK, UK), pp. 7–24, Springer-Verlag, 2000, http://dl.acm.org/citation.cfm?id=648236.761384.

[20] A. Kennedy, N. Benton, J. B. Jensen and P.-E. Dagand, *Coq: The world's best macro assembler?*, in *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, (New York, NY, USA), pp. 13–24, ACM, 2013, DOI.

[21] J. S. Moore, *A mechanically verified language implementation*, *Journal of Automated Reasoning* **5** (1989) 461.

[22] W. A. Hunt Jr, *Microprocessor design verification*, *Journal of Automated Reasoning* **5** (1989) 429.

[23] *Journal of Automated Reasoning* **30** (2003) .

[24] G. C. Necula, *Proof-carrying code. design and implementation*. Springer, 2002.

[25] A. W. Appel, *Foundational proof-carrying code*, in *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, (Washington, DC, USA), pp. 247–, IEEE Computer Society, 2001, http://dl.acm.org/citation.cfm?id=871816.871860.

[26] J. Yang and C. Hawblitzel, *Safe to the last instruction: Automated verification of a type-safe operating system*, in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, (New York, NY, USA), pp. 99–110, ACM, 2010, DOI.

[27] T. Maeda and A. Yonezawa, *Typed assembly language for implementing os kernels in smp/multi-core environments with interrupts*, in *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2010, http://dl.acm.org/citation.cfm?id=1929004.1929005.

[28] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, *Boogie: A modular reusable verifier for object-oriented programs*, in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, (Berlin, Heidelberg), pp. 364–387, Springer-Verlag, 2006, DOI.

[29] H. Xi and R. Harper, *A dependently typed assembly language*, in *ACM SIGPLAN Notices*, vol. 36, pp. 169–180, ACM, 2001.

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin et al., *sel4: Formal verification of an os kernel*, in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, ACM, 2009.

[31] H. Tuch, G. Klein and M. Norrish, *Types, bytes, and separation logic*, in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Martin Hofmann and Matthias Felleisen, ed., (Nice, France), pp. 97–108, ACM, jan, 2007.

[32] A. Chlipala, *A verified compiler for an impure functional language*, in *ACM Sigplan Notices*, vol. 45, pp. 93–106, ACM, 2010.

[33] G. C. Necula, *Translation validation for an optimizing compiler*, in *ACM sigplan notices*, vol. 35, pp. 83–94, ACM, 2000.

[34] P. Curzon and P. Curzon, *A verified compiler for a structured assembly language*, in *In proceedings of the 1991 international workshop on the HOL theorem Proving System and its applications. IEEE Computer*, 1991.

[35] M. Strecker, *Formal verification of a java compiler in isabelle*, in *Automated DeductionCADE-18*, pp. 63–77, Springer, (2002).

[36] X. Leroy, *A formally verified compiler back-end*, *Journal of Automated Reasoning* **43** (2009) 363.

[37] A. W. Appel, *Verified software toolchain*, in *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, (Berlin, Heidelberg), pp. 1–17, Springer-Verlag, 2011, http://dl.acm.org/citation.cfm?id=1987211.1987212.

[38] G. Neis, C.-K. Hur, J.-O. Kaiser, C. McLaughlin, D. Dreyer and V. Vafeiadis, *Pilsner: A compositionally verified compiler for a higher-order imperative language*, in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, (New York, NY, USA), pp. 166–178, ACM, 2015, DOI.

[39] T. Ramananandro, Z. Shao, S.-C. Weng, J. Koenig and Y. Fu, *A compositional semantics for verified separate compilation and linking*, in *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, (New York, NY, USA), pp. 3–14, ACM, 2015, DOI.

[40] Z. Jiang, M. Pajic and R. Mangharam, *Cyber-physical modeling of implantable cardiac medical devices*, *Proceedings of the IEEE* **100** (2012) 122.

[41] A. O. Gomes and M. V. M. Oliveira, *Formal Specification of a Cardiac Pacing System*, pp. 692–707. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. 10.1007/978-3-642-05089-3_44.

[42] T. Chen, M. Diciolla, M. Kwiatkowska and A. Mereacre, *Quantitative verification of implantable cardiac pacemakers*, in *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pp. 263–272, IEEE, 2012.

[43] L. C. Cordeiro, B. Fischer, H. Chen and J. Marques-Silva, *Semiformal verification of embedded software in medical devices considering stringent hardware constraints*, in *ICESS*, 2009.

[44] P. J. Landin, *The Mechanical Evaluation of Expressions*, *The Computer Journal* **6** (1964) 308.

[45] B. Graham, *Secd: Design issues*, tech. rep., University of Calgary, 1989.

[46] T. J. Clarke, P. J. Gladstone, C. D. MacLean and A. C. Norman, *Skim - the s, k, i reduction machine*, in *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, (New York, NY, USA), pp. 128–135, ACM, 1980, DOI.

[47] L. P. Deutsch, *A lisp machine with very compact programs*, in *Proceedings of the 3rd international joint conference on Artificial intelligence*, pp. 697–703, Morgan Kaufmann Publishers Inc., 1973.

[48] P. M. Kogge, *"The Architecture of Symbolic Computers"*. McGraw-Hill, Inc., New York, New York, 1991.

[49] T. F. Knight, *Implementation of a list processing machine*, Ph.D. thesis, Massachusetts Institute of Technology, 1979.

[50] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter and H. Isozaki, *Flicker: An execution infrastructure for tcb minimization*, *SIGOPS Oper. Syst. Rev.* **42** (2008) 315.

[51] E. Keller, J. Szefer, J. Rexford and R. B. Lee, *Nohype: Virtualized cloud infrastructure without the virtualization*, *SIGARCH Comput. Archit. News* **38** (2010) 350.

[52] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan et al., *Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses*, in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 129–142, IEEE, 2008.

[53] S. Gollakota, H. Hassanieh, B. Ransford, D. Katabi and K. Fu, *They can hear your heartbeats: non-invasive security for implantable medical devices*, in *Proc. ACM Conf. SIGCOMM*, pp. 2–13, 2011.

[54] T. Denning, K. Fu and T. Kohno, *Absence makes the heart grow fonder: New directions for implantable medical device security.*, in *HotSec*, 2008.

[55] G. Morrisett, D. Walker, K. Crary and N. Glew, *From System F to Typed Assembly Language*, *ACM Trans. Program. Lang. Syst.* **21** (1999) 527.

[56] K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker et al., *TALx86: A realistic typed assembly language*, in *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, pp. 25–35, 1999, http://www.cis.upenn.edu/ stevez/papers/MCGG99.pdf.

[57] H. Xi and R. Harper, *A Dependently Typed Assembly Language*, in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, (New York, NY, USA), pp. 169–180, ACM, 2001, DOI.

[58] G. Morrisett, K. Crary, N. Glew and D. Walker, *Stack-based typed assembly language*, *Journal of Functional Programming* **12** (2002) .

[59] K. Crary, *Toward a Foundational Typed Assembly Language*, in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, (New York, NY, USA), pp. 198–212, ACM, 2003, DOI.

[60] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. HriÅ£cu, D. Pichardie et al., *A Verified Information-flow Architecture*, in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), pp. 165–178, ACM, 2014, DOI.

[61] G. Balakrishnan, R. Gruian, T. Reps and T. Teitelbaum, *CodeSurfer/x86A Platform for Analyzing x86 Executables*, in *Compiler Construction*, Lecture Notes in Computer Science, pp. 250–254, Springer, Berlin, Heidelberg, Apr., 2005, DOI.

[62] J. Lee, T. Avgerinos and D. Brumley, *Tie: Principled reverse engineering of types in binary programs*, in *In Proceedings of the Network and Distributed System Security Symposium*, 2011.

[63] M. Noonan, A. Loginov and D. Cok, *Polymorphic Type Inference for Machine Code*, *arXiv:1603.05495 [cs]* (2016) .

[64] J. Caballero and Z. Lin, *Type Inference on Executables*, *ACM Comput. Surv.* **48** (2016) 65:1.

[65] Z. Chen, *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional, 2000.

[66] W. Coekaerts, "The java typesystem is broken." `http://wouter.coekaerts.be/2018/java-type-system-broken`.

[67] N. Amin and R. Tate, *Java and scala's type systems are unsound: The existential crisis of null pointers*, in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 838–848, ACM, 2016, DOI.

[68] R. Grigore, *Java generics are turing complete*, *CoRR* **abs/1605.05274** (2016) [`1605.05274`].

[69] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim and S. A. Edwards, *Hardware synthesis from a recursive functional language*, in *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, CODES '15, (Piscataway, NJ, USA), pp. 83–93, IEEE Press, 2015, http://dl.acm.org/citation.cfm?id=2830840.2830850.

[70] R. Townsend, M. A. Kim and S. A. Edwards, *From functional programs to pipelined dataflow circuits*, in *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, (New York, NY, USA), pp. 76–86, ACM, 2017, DOI.

[71] S. Nagarakatte, J. Zhao, M. M. Martin and S. Zdancewic, *Softbound: Highly compatible and complete spatial memory safety for c*, in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 245–258, ACM, 2009, DOI.

[72] J. Devietti, C. Blundell, M. M. K. Martin and S. Zdancewic, *Hardbound: Architectural support for spatial safety of the c programming language*, in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (New York, NY, USA), pp. 103–114, ACM, 2008, DOI.

[73] S. Nagarakatte, M. M. K. Martin and S. Zdancewic, *Watchdog: Hardware for safe and secure manual memory management and full memory safety*, in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 189–200, IEEE Computer Society, 2012, http://dl.acm.org/citation.cfm?id=2337159.2337181.

[74] B. Hardekopf and C. Lin, *Flow-sensitive pointer analysis for millions of lines of code*, in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, (Washington, DC, USA), pp. 289–298, IEEE Computer Society, 2011, http://dl.acm.org/citation.cfm?id=2190025.2190075.

[75] E. Moggi, *Notions of computation and monads*, *Information and computation* **93** (1991) 55.

[76] P. Hudak, J. Hughes, S. Peyton Jones and P. Wadler, *A history of haskell: being lazy with class*, in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 12–1, ACM, 2007.

[77] R. Hindley, *The principal type-scheme of an object in combinatory logic*, *Transactions of the American Mathematical Society* **146** (1969) 29.

[78] R. Milner, *A theory of type polymorphism in programming*, *Journal of Computer and System Sciences* **17** (1978) 348.

[79] "Smallstep small-step operational semantics." http://flint.cs.yale.edu/cs430/lectureNotes/terse/Smallstep.html.

[80] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu et al., *The gem5 simulator*, *SIGARCH Comput. Archit. News* **39** (2011) 1.

[81] "The Coq proof assistant: https://coq.inria.fr."

[82] M. Madhavan and P. A. Friedman, *Optimal programming of implantable cardiac-defibrillators*, *Circulation* **128** (2013) 659 [`http://circ.ahajournals.org/content/128/6/659.full.pdf`].

[83] R. Mangharam, H. Abbas, M. Behl, K. Jang, M. Pajic and Z. Jiang, *Three challenges in cyber-physical systems*, in *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*, pp. 1–8, Jan, 2016, DOI.

[84] S. Shuja, S. K. Srinivasan, S. Jabeen and D. Nawarathna, *A formal verification methodology for ddd mode pacemaker control programs*, *Journal of Electrical and Computer Engineering* (2015) .

[85] S. J. Connolly, M. Gent, R. S. Roberts, P. Dorian, D. Roy, R. S. Sheldon et al., *Canadian implantable defibrillator study (cids)*, *Circulation* **101** (2000) 1297 [`http://circ.ahajournals.org/content/101/11/1297.full.pdf`].

[86] T. A. versus Implantable Defibrillators (AVID) Investigators, *A comparison of antiarrhythmic-drug therapy with implantable defibrillators in patients resuscitated from near-fatal ventricular arrhythmias*, *New England Journal of Medicine* **337** (1997) 1576 [`http://dx.doi.org/10.1056/NEJM199711273372202`].

[87] J. Siebels, K.-H. Kuck and C. Investigators, *Implantable cardioverter defibrillator compared with antiarrhythmic drug treatment in cardiac arrest survivors (the cardiac arrest study hamburg)*, *American Heart Journal* **127** (1994) 1139.

[88] "Living with your implantable cardioverter defibrillator (ICD)." `http://www.heart.org/HEARTORG/Conditions/Arrhythmia/PreventionTreatmentofArrhythmia/Living-With-Your-Implantable-Cardioverter-Defibrillator-ICD_UCM_448462_Article.jsp`.

[89] "How many people have ICDs?." `http://asktheicd.com/tile/106/english-implantable-cardioverter-defibrillator-icd/how-many-people-have-icds/`.

[90] J. Pan and W. J. Tompkins, *A real-time qrs detection algorithm*, *IEEE Transactions on Biomedical Engineering* **BME-32** (1985) 230.

[91] M. M. Cruz-Cunha, J. Varaj£o, H. Krcmar, R. Martinho, R. A. lvarez, A. J. M. Penn et al., *Centeris 2013 - conference on enterprise information systems / projman 2013 - international conference on project management/ hcist 2013 - international conference on health and social care information systems and technologies a comparison of three qrs detection algorithms over a public database*, *Procedia Technology* **9** (2013) 1159 .

[92] "Open source ECG analysis software." `http://www.eplimited.com/confirmation.htm`.

[93] M. S. Wathen, P. J. DeGroot, M. O. Sweeney, A. J. Stark, M. F. Otterness, W. O. Adkisson et al., *Prospective randomized multicenter trial of empirical antitachycardia pacing versus shocks for spontaneous rapid ventricular tachycardia in patients with implantable cardioverter-defibrillators*, *Circulation* **110** (2004) 2591 [`http://circ.ahajournals.org/content/110/17/2591.full.pdf`].

[94] M. E. Conway, *Design of a separable transition-diagram compiler*, *Commun. ACM* **6** (1963) 396.

[95] A. L. D. Moura and R. Ierusalimschy, *Revisiting coroutines*, *ACM Trans. Program. Lang. Syst.* **31** (2009) 6:1.

[96] V. Kashyap, B. Wiedermann and B. Hardekopf, *Timing- and termination-sensitive secure information flow: Exploring a new approach*, in *2011 IEEE Symposium on Security and Privacy*, pp. 413–428, May, 2011, DOI.

[97] V. Simonet, *Fine-grained information flow analysis for a λ calculus with sum types*, in *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, CSFW '02, (Washington, DC, USA), pp. 223–, IEEE Computer Society, 2002, http://dl.acm.org/citation.cfm?id=794201.795163.

[98] Z. Cutlip, "Dlink dir-815 upnp command injection." `http://shadow-file.blogspot.com/2013/02/dlink-dir-815-upnp-command-injection.html`, February, 2013.

[99] A. Cui, M. Costello and S. J. Stolfo, *When firmware modification attack: A case study of embedded exploitation*, in *NDSS Symposium '13*, 2013.

[100] G. Hernandez, O. Arias, D. Buentello and Y. Jin, *Smart nest thermostat: A smart spy in your home*, in *Black Hat Briefings*, 2014.

[101] E. Buchanan, R. Roemer and S. Savage, "Return-oriented programming: Exploits without code injection." https://hovav.net/ucsd/talks/blackhat08.html.

[102] B. P. Miller, L. Fredriksen and B. So, *An empirical study of the reliability of unix utilities*, *Commun. ACM* **33** (1990) 32.

[103] X. Yang, Y. Chen, E. Eide and J. Regehr, *Finding and understanding bugs in c compilers*, in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, (New York, NY, USA), pp. 283–294, ACM, 2011, DOI.

[104] C. Lidbury, A. Lascu, N. Chong and A. F. Donaldson, *Many-core compiler fuzzing*, in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, (New York, NY, USA), pp. 65–76, ACM, 2015, DOI.

[105] J. Ruderman, *Introducing jsfunfuzz*, 2007.

[106] C. Holler, K. Herzig and A. Zeller, *Fuzzing with code fragments*, in *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, (Berkeley, CA, USA), pp. 38–38, USENIX Association, 2012, http://dl.acm.org/citation.cfm?id=2362793.2362831.

[107] R. Brummayer and A. Biere, *Fuzzing and delta-debugging smt solvers*, in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, (New York, NY, USA), pp. 1–5, ACM, 2009, DOI.

[108] R. Brummayer and M. JÄrvisalo, *Testing and debugging techniques for answer set solver development*, *Theory Pract. Log. Program.* **10** (2010) 741.

[109] R. Brummayer, F. Lonsing and A. Biere, *Automated testing and debugging of sat and qbf solvers*, in *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, SAT'10, (Berlin, Heidelberg), pp. 44–57, Springer-Verlag, 2010, DOI.

[110] K. Dewey, J. Roesch and B. Hardekopf, *Fuzzing the rust typechecker using clp*, in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, (Washington, DC, USA), pp. 482–493, IEEE Computer Society, 2015, DOI.

[111] J. Jaffar and J.-L. Lassez, *Constraint logic programming*, in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, (New York, NY, USA), pp. 111–119, ACM, 1987, DOI.

[112] J. Jaffar and M. J. Maher, *Constraint logic programming: A survey*, *Journal of Logic Programming* **19** (1994) 503.

[113] J. Wielemaker, T. Schrijvers, M. Triska and T. Lager, *SWI-Prolog*, *Theory and Practice of Logic Programming* **12** (2012) 67.

[114] D. Diaz and P. Codognet, *The gnu prolog system and its implementation*, in *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, SAC '00, (New York, NY, USA), pp. 728–732, ACM, 2000, DOI.

[115] L. De Moura and N. Bjørner, *Z3: An efficient smt solver*, in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008, http://dl.acm.org/citation.cfm?id=1792734.1792766.

[116] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, *Mibench: A free, commercially representative embedded benchmark suite*, in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001, DOI.

[117] A. Schiffer, *How a fish tank helped hack a casino*, July, 2017.