University of Oxford

Termination Analysis of λ-calculus and a subset of core ML

by

William Blum

Lady Margaret Hall

Dissertation submitted in partial fulfilment of the degree of
Master of Science in Computer Science

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD

September 1, 2004

Abstract

Termination analysis is a very important component of software verification: it is futile trying to prove a property on a program result if the program does not terminate and therefore never returns the result. Turing showed that termination is an undecidable property. However in [6], Lee, Jones and Ben-Amram introduced “size-change termination”, a decidable property strictly stronger than termination. They proposed a method called the “size-change principle” to analyze it.
Size-change analysis relies on a finite approximation of the program computational behavior. A call semantics is defined such that the presence of infinite call sequences characterizes non-termination. Since the approximated computational space is finite, infinite call sequences must contain loops. Deciding the size-change property then amounts to analyze loops of the program through the use of “size-change graphs” describing program calls.

We first explain the size-change principle in the first-order case ([6]) and its adaptation to the untyped λ-calculus ([5]). My implementation provides some improvements over the original method: it avoids variable renaming and generates a more accurate approximation. Finally we extend the size-change principle to a subset of ML featuring ground type values, higher-order type values and recursively defined functions. Compared to other works, this is the first time that the size-change principle is applied to a higher-order functional language. In a first attempt, the ML program is converted into a λ-calculus expression, by means of Church numerals and the Y combinator, and analyzed using the algorithm of [5]. Implementing numbers with church numerals has two important drawbacks: the size of the converted program increases proportionally to the integer values used in its definition. Secondly, since the decrease in integer values is not properly reflected by church numerals, most recursively defined functions operating on numbers are not recognized as terminating! In the second approach, being inspired by [5] we redefine from scratch an algorithm for the core ML case which handles natively `if-then-else` and `let rec` structures with no conversion. This algorithm produces the same result as [5] for higher-order values but can also analyze the size of ground type values. This enhances the scope of the termination analyzer to some recursively defined function operating on numbers.

An electronic version of this thesis as well as OCaml sources are available at the following address: `http://www.famille-blum.org/~william/mscthesis/`

Acknowledgements

My sincere thank goes to Professor Luke Ong, who supervised me on this project, for his encouragement and his corrections of the final draft.

Many thanks to the friends met in Lady Margaret Hall who made my stay in Oxford so enjoyable and in particular to my girlfriend Lily Lin (林浅寒).

I am very grateful to my family. Especially to my dear parents Bianca and Dominique and my grand-parents Anita and Luigi Bombardier for their spiritual and financial support throughout this year in Oxford.

Contents

List of Tables

Chapter 1

Introduction

1.1 History

One of the most challenging problems in computer science was to find an algorithm which can tell whether a given computer program terminates or not. This is known as the Halting problem, an important problem in computer science since it is the first problem that has been proved undecidable. Other problems have been proved undecidable and usually this has been achieved by proving that they reduce to the Halting problem ([9]).

The undecidability of the halting problem implies the unsolvability of the Entscheidungs problem (determining if a given first order logic statement is valid or not). Another consequence is Rice's theorem which says that the truth of a non-trivial statement about a function defined by an algorithm is undecidable. As a consequence, the problem "this algorithm halts for the input 0" is undecidable.

Turing proved the undecidability of the Halting problem. This famous proof proceeds by reductio ad absurdum: we suppose that a function exists such that it returns yes when a particular program terminates and no if it does not and then we build a new function which uses the first one and causes a contradiction.

The original proof of Alan Turing is based on a formalization of the concept of algorithm using Turing machines. However, his result is still valid in other models of computation computationally equivalent to Turing machines such as Lambda Calculus, the base "language" used in this project.

Turing's result shows that there is no general method to answer the questions of the type: "Does this particular program terminate for all input value?". But particular instances of the halting problem may be solved. Given a specific program, it is sometimes possible to prove that for any input it will halt. The difficulty is that every proof requires new arguments which cannot be guessed in a mechanical way.

The undecidability of the halting problem relies on the fact that programs have potentially infinite memory. In practice, the amount of memory used by existing computers is limited. In that case, the halting problem for the constrained case of limited memory computers is solvable by a general algorithm which is however inefficient ([9]).

1.2 Selection of particular question for study

The halting problem is undecidable, however there exist properties stronger than termination which are decidable. This suggests that there could be mechanical ways to prove termination in some particular cases.

In this project, we are interested in the study of the *size-change termination* property. *Size-change termination* is strictly stronger than termination (*size-change termination* implies termination but not all terminating programs are *size-change terminating*) but it is decidable.

The *size-change principle* is aimed at analyzing this property through the use of special graphs named "*size-change graphs*". These graphs describe the calls occurring in a program.

The aim of this MSc project is to implement and extend the size-change principle for termination analysis of programs expressed in λ-calculus or in a purely functional language such as the functional core part of ML.

1.3 Proposed method

The method which I will use is explained by Neil Jones and Nina Bohr in [5] and based on the size-change principle introduced by Neil Jones *et al.* in [6].

In [6], the size-change principle is used to determine whether a first-order functional program with well-founded parameter values halts.

In [5], the method is adapted to the case of λ-calculus. The algorithm explained can tell whether the call-by-value evaluation of a closed untyped λ-term terminates.

The method is sound: when it returns yes, the input program is guaranteed to terminate. However it is also incomplete: when it returns no, the input program may or may not terminate.

1.4 Description of the project

The work for this MSc project consisted in:

- understanding the size-change principle and the generation of the size-change graphs,
- constructing an implementation of the termination certifier for the call-by-value λ-calculus.

As an optional part, the following directions have then been explored:

- implementing a parser for λ-calculus expressions
- extending the method to a subset of the Core ML language with basic arithmetic constants and recursively defined function (based on the semantics given in [7]).

- implementing a parser for the Core ML language

Objective Caml ([4]) is the language I used to implement all the algorithms. See [3] for the complete implementation sources.

1.5 Timetable

I give here the timetable followed during this project:

- From April, 26th to May, 15th: Background readings ([6], [5], [7]).
- From May, 16th to May, 31st: Implementation of the algorithm given in [5].
- From June, 1st to June, 30th: Implementation of an OCaml parser and integration with the algorithms developed before. In parallel: extension of the size-change principle to a subset of core ML (first approach)
- From July, 1st to July, 31th: Extension to a subset of core ML ([7]) and extension of the size-change principle with a second approach.
- From August, 1st to August, 31th: Dissertation writing.

1.6 Link between the project and the taught part of the course

There is an obvious strong link between this project and the Lambda Calculus course (Hilary term). λ-calculus is the language on which we first concentrate for the termination analysis of programs.

Moreover, the ML extension of the principle required knowledge acquired during the Lambda Calculus course such as techniques for proving theorems by induction and case analysis and for defining the semantics of a language.

Finally, the techniques learnt during the Compilers course (Michaelmas term) gave me the skills necessary to build a parser for lambda calculus expression and a subset of Core ML language using the tools `ocamllex` and `ocamlyacc`.

1.7 Structure of the dissertation

There are basically three main parts in the dissertation. Each of them deals with a different application of the size-change principle and the first one also introduces the principle. The following is an outline of the dissertation:

- The next chapter (2) presents in details the size-change principle. The first-order functional programming language introduced in [6] is used as an example throughout this chapter. Results from [6] are recapitulated in an original way. Some of the definitions extracted from [6] have been slightly modified to make them more general in order to reuse them in the following chapters.

  The general algorithm for size-change termination analysis is explained in this chapter.

- Chapter 3 deals with the higher-order case. The language used is the untyped λ-calculus. This chapter starts by introducing the method explained in [5] for applying the size-change principle to the λ-calculus. It is based on the results of chapter 2. Only the results of [5] which are specific to the λ-calculus case are stated.

  It then gives details about my improvements over the method explained in [5].

  The chapter concludes with implementation details and practical results obtained.

- Chapter 4 is an account of the work I have carried out in order to extend the size-change principle to a more complicated functional language. The language used is a subset of core ML.

  We will see how to deal with recursively defined function (with and without the use of combinators), ground types values and if-then-else structure.

  The two approaches that I have tried are explained. The first one proceeds by conversion from the core ML language to the λ-calculus. The second one consists in redefining from scratch an appropriated size-change principle for the core ML language. We will see that the latter approach is more powerful than the first.

  The resulting algorithm is powerful enough to prove termination of higher-order and first order programs.

  Implementation details are discussed as well as practical results.

- The last chapter is the conclusion of the dissertation.

- The Appendix contains proofs of the important lemma and theorems stated in chapter 4.

Chapter 2

The size-change principle for first-order programs

This chapter introduces some basic notions and explains in details the size-change principle introduced in [6]. Definitions and theorems are illustrated with examples. For simplicity, these examples are all based on first order programs (in contrast with higher-order programs where a function can be passed as a parameter to another function).

In the next chapter we deal with higher-order programs: the size-change principle is applied to the simply typed lambda calculus.

Most of the definitions and theorems given in this chapter will be used in the next two chapters.

2.1 Basic concepts: well-founded set

[9] We recall here some basic definitions and properties on well-founded set used in mathematics.

Let $(X, \leq)$ be a partially ordered set.

Definition 2.1.1 (Well-founded set)**.** *We say that the relation $\leq$ is well-founded on X if every non-empty subset E of X has a minimal element for $\leq$ (an element $m \in E$ such that $\forall e \in E \cdot \neg(e \leq m)$).*

X is then said to be a well-founded set.

Well-founded sets are particulary adapted for the induction principle. Indeed, to prove that a particular property P holds for all elements of a well-founded set $(X, \leq)$, it suffices to show the following property:

$$[\forall y \cdot y \leq x \implies P(y)] \text{ implies that } P(x) \text{ holds.}$$

In particular for any minimal elements m, $P(m)$ must hold.

Definition 2.1.2 (Chain)**.** *A chain is a totally ordered subset of a partially order set.*

A chain is said to be infinitely descending if it has no minimal element.

We can now prove the following proposition which characterizes a well-founded set by its chains.

Proposition 2.1.1 (Well-founded set characterization)**.** *A partially order set is well-founded if and only if it contains no infinite descending chain.*

The implication of this proposition will be used in the following section to justify that a program verifying the size-change termination condition must terminate.

2.2 The language $\mathcal{L}$ for first-order programs

In this chapter we consider first-order programs. The programming language considered is an untyped functional language which supports recursion, if-then-else, and primitive operator calls. It is defined by Neil D. Jones in the article introducing the size-change principle ([6]). $\mathcal{L}$ denotes this first-order language. Its syntax and semantics are recalled below.

Loop structures (like `for`, `while` and `repeat` loops) are not present in the language. This simplifies the size-change principle (indeed dealing with loops structure would require to define a special notion of function call).

Definition 2.2.1 (Syntax of $\mathcal{L}$)**.**

$$\begin{array}{rcl}
\mathtt{P} \in \mathit{Prog} & ::= & \mathtt{def}_1 \ldots \mathtt{def}_\mathtt{m} \\
\mathtt{def} \in \mathit{Def} & ::= & \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}) = \mathtt{e}^\mathtt{f} \\
\mathtt{e} \in \mathit{Expr} & ::= & x \\
 & | & \mathtt{e}_1 \ = \ \mathtt{e}_2 \\
 & | & \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \\
 & | & \mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_\mathtt{n}) \\
 & | & \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_\mathtt{n}) \\
\mathtt{x} \in \mathit{Parameter} & ::= & \mathit{identifier} \\
\mathtt{f} \in \mathit{FcnName} & ::= & \mathit{identifier\ not\ in\ Parameter} \\
\mathtt{op} \in \mathit{Op} & ::= & \mathit{primitive\ operator}
\end{array}$$

The first function defined in the list of definitions of the program is denoted $\mathtt{f}_{initial}$*. This is the function which initializes the program computation (i.e. the first function called).*

If `f` *is a function then:*

- $\mathtt{e}^\mathtt{f}$ *denotes the body of the function in its definition.*
- $Param(\mathtt{f})$ *denotes the set of* `f`*'s parameters*
- $\mathtt{f}^{(i)}$ *denotes the* i^{th} *parameter of* `f`*.*

Op is the set of operators, for instance `pred` and `succ` are two operators (the predecessor and successor for numbers).

The semantics used to interpret $\mathcal{L}$ is the call-by-value evaluation semantics (see [8]) given in definition 2.2.2.

$\mathcal{E}$ denotes the semantic operator: $\mathcal{E}[\![\mathtt{e}]\!]\ \vec{v}$ is the value of expression `e` in the environment $\vec{v}$. An environment is a tuple containing the value of the parameter of `f`.

$\mathcal{E}$ is a function of type $\mathit{Expr} \rightarrow Value^* \rightarrow Value^\#$ where $Value^*$ is the set of finite sequences of $Value$ elements and $Value^\# = Value \cup \{\bot, \mathit{Err}\}$. $\bot$ represents non-termination and *Err* represents runtime error (i.e. exception). *Err* is the result of `pred 0` for instance.

Primitive operators like `pred` are interpreted using another semantic operator: $\mathcal{O} : \mathit{Op} \rightarrow Value^* \rightarrow Value^\#$. We assume that primitives always terminate therefore $\forall op \in \mathit{Op}, \vec{v} \in Value^* : \mathcal{O}[\![op]\!](\vec{v}) \neq \bot$. However operators may cause errors.

The complete semantics of this programming language is given by the following definition:

Definition 2.2.2 (Call-by-value semantics of $\mathcal{L}$)**.** *See paragraph 1.4 and figure 5.1 of [6] for more details.*

Domains

$$\begin{array}{ll} v \in Value & \textit{with the special value } True \in Value \\ u, w \in Value^\# = Value \cup \{\bot, \mathit{Err}\}, & \textit{where } \bot \sqsubseteq w \textit{ for all } w. \end{array}$$

Types

$$\begin{array}{rcl} \mathcal{E} & : & \mathit{Expr} \rightarrow Value^* \rightarrow Value^\# \\ \mathcal{O} & : & \mathit{Op} \rightarrow Value^* \rightarrow Value^\# \\ lift & : & Value \rightarrow Value^\# \\ \mathit{strictapply} & : & (Value^* \rightarrow Value^\#) \rightarrow (Value^\#)^* \rightarrow Value^\# \end{array}$$

Semantic operator

$$\begin{array}{rcl} \mathcal{E}[\![\mathtt{f}^{(\mathtt{i})}]\!](v_1, ..., v_n) & = & lift\ v_i \\ \mathcal{E}[\![\mathtt{if\ e\ then\ e_1\ else\ e_2}]\!]\ \vec{v} & = & \begin{cases} \mathcal{E}[\![\mathtt{e}]\!]\ \vec{v} & \textit{if } \mathcal{E}[\![\mathtt{e}]\!]\ \vec{v} \in \{\bot, \mathit{Err}\} \\ \mathcal{E}[\![\mathtt{e_1}]\!]\ \vec{v} & \textit{if } \mathcal{E}[\![\mathtt{e}]\!]\ \vec{v} = True \\ \mathcal{E}[\![\mathtt{e_2}]\!]\ \vec{v} & \textit{elsewhere.} \end{cases} \\ \mathcal{E}[\![\mathtt{op(e_1, \ldots, e_n)}]\!]\ \vec{v} & = & \mathit{strictapply}(\mathcal{O}[\![\mathtt{op}]\!])(\mathcal{E}[\![\mathtt{e_1}]\!]\ \vec{v}, \ldots \mathcal{E}[\![\mathtt{e_n}]\!]\ \vec{v}) \\ \mathcal{E}[\![\mathtt{f(e_1, \ldots, e_n)}]\!]\ \vec{v} & = & \mathit{strictapply}(\mathcal{E}[\![\mathtt{e^f}]\!])(\mathcal{E}[\![\mathtt{e_1}]\!]\ \vec{v}, \ldots \mathcal{E}[\![\mathtt{e_n}]\!]\ \vec{v}) \end{array}$$

Auxiliary operation *The function strictapply implements the mechanism of exception in programming languages:*

$$
\textit{strictapply}\ \psi(w_1, \dots w_n) \;=\; \begin{cases} \psi(v_1, \dots v_n) & \textit{if } \forall i \in 1 \dots n : w_i \notin \{\bot, Err\} \\ & \textit{and } w_i = lift\ v_i; \\ & \\ w_i & \textit{elswhere, where } i \textit{ is the least index} \\ & \textit{such that } w_i \in \{\bot, Err\} \end{cases}
$$

Assumption

$$\mathcal{O}[\![op]\!]\ \vec{v} \neq \bot$$

2.3 Control flow graph and state transition in $\mathcal{L}$ programs

The computational behavior of the input program is represented by a call-graph. The vertices of the call-graph are the *program control points*. They correspond to the calls made in the evaluation of the program. An arc from one call to another signifies that the latter call is caused directly by the former.

The notion of program points, calls and control flow graph are defined formally in the following paragraphs:

2.3.1 Program points

Program points are possible positions in the program where calls can occur. They characterize the caller of a call as well as the callee. $\mathcal{P}$ denotes the set of all program points in a program.

For instance, for first order programs (studied in [6]), we can define program points to be the function names of the program.

In the following example:

```
plus(x,y) = x + y
f = plus(3,2)
```

the set of program points is $\mathcal{P} = \{\texttt{plus}, \texttt{f}\}$.

In order to apply the size-change principle, a requirement is for the set $\mathcal{P}$ to be finite (this implies the presence of loops in infinite call sequences). $\mathcal{P}$ must be a finite approximation of the infinite set of possible states in the program.

2.3.2 Calls

Definition 2.3.1 (Call)**.** *Let $p_1, p_2 \in \mathcal{P}$. We write $p_1 \xrightarrow{c} p_2$ to denote a **call** to program point p_2 occurring at program point p_1 and labeled with the name c.*

In the first-order case, program points are the function names. A call is therefore defined by the name of the caller function and the name of the function called. It is represented by an arrow in the control flow graph of a program. In the example of section 2.3.1, the call from function `f` to function `plus` is denoted $\mathtt{f} \rightarrow \mathtt{plus}$.

Definition 2.3.2 (Transitive call)**.**

1. *A* ***call sequence*** *is a finite or infinte sequence of calls:* $cs = \langle c_1 c_2 \ldots \rangle$
2. *A call sequence is* ***well-formed*** *for* `P` *if there are functions* $\mathtt{f}_1, \mathtt{f}_2, \ldots$ *such that* $\mathtt{f}_1 \xrightarrow{c_1} \mathtt{f}_2$*,* $\mathtt{f}_2 \xrightarrow{c_2} \mathtt{f}_3$*, ...*
3. *If cs is finite then* $cs = \langle c_1 c_2 \ldots c_k \rangle$ *and we use the notation*

$$\mathtt{f} \xrightarrow{cs} \mathtt{f}_{\mathtt{k+1}} \triangleq \left[\mathtt{f}_1 \xrightarrow{c_1} \mathtt{f}_2, \mathtt{f}_2 \xrightarrow{c_2} \mathtt{f}_3, \ldots, \mathtt{f}_{\mathtt{k}} \xrightarrow{c_k} \mathtt{f}_{\mathtt{k+1}}\right]$$

   to denote the ***transitive call*** *from* `f` *to* $\mathtt{f}_{\mathtt{k+1}}$*.*

4. $\mathtt{f} \rightarrow^* \mathtt{f}_{\mathtt{k+1}}$ *means that* $\mathtt{f} \xrightarrow{cs} \mathtt{f}_{\mathtt{k+1}}$ *for some call sequence cs.*
5. *A call sequence from a program point to itself is a* ***recursive transitive call*** *(*cs_{rec} *is a recursive transitive call if* $\mathtt{f} \xrightarrow{cs_{rec}} \mathtt{f}$ *for some* $\mathtt{f} \in \mathcal{P}$*).*

We say that a call $f \xrightarrow{c} g$ is **activable** if there is a program point reachable in `P` where the call can be made, in other words if we have $\mathtt{f}_{initial} \rightarrow^* \mathtt{f} \xrightarrow{c} \mathtt{g}$.

If a call is activable then its corresponding arc in the control flow graph belongs to the connected component of the control flow graph containing $\mathtt{f}_{initial}$. Note that the reciprocal is false since the connected component containing $\mathtt{f}_{initial}$ can contain an arc representing a dead code call.

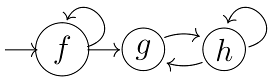
2.3.3 Control flow

The *control flow* of a first-order program is a graph which nodes correspond to the functions of the program and which edges correspond to possible calls from one function to another.

Example 2.3.1
The control flow of the following program:

```
f(n) = if n mod 2 = 0 then g(2*n) else 5 + f(n-1)
g(n) = if n = 0 then 0 else h(n-1)
h(n) = if n mod 2 = 0 then 2 * g(n-1) else 1 + h(n-1)
```

is



The control flow graph is statically determined. This means that one does not need to run the program in order to find the edges of the control flow graph: they can be found just by looking at the code defining the program.

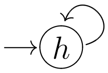
Each computation of `P` can be represented as a path in the control flow graph starting at node $\mathtt{f}_{initial}$. This path may be infinite if the computation does not terminate.

Example 2.3.2
Consider the program given in last example. The call sequence of the computation of `f` on input `7` is of the form $\langle f, f, g, h, \ldots \rangle$.

Not all infinite paths of the control flow graph correspond to a possible computation. For instance, consider the following program:

```
h(n) = if false then h(n+1) else 1
```

Its control graph is:



The control graph of h contains an infinite path corresponding to the call sequence $\langle h, h, \ldots, h, \ldots \rangle$. But there is no such possible execution of the program h since the recursive call in the definition of h is part of a dead-code block.

2.3.4 State transition

Call sequences do not describe completely a computation of a program. They just give information on the functions beeing called during the execution of the program. In order to describe completely the computations, state transition sequences have been introduced in [6]. The formal definition is given below:

Definition 2.3.3 (State transition sequence)**.**

- *A state is a pair in* $FcnName \times Value^*$.
- *A state transition* $(\mathtt{f}, \overrightarrow{v}) \to (\mathtt{g}, \overrightarrow{u})$ *is a pair of states connected by a call of type* $\mathtt{g}(\mathtt{e_1}, \ldots, \mathtt{e_n})$ *occuring in* $\mathtt{e}^{\mathtt{f}}$*, the body of* `f`*, such that* $\overrightarrow{u} = (u_1 \ldots u_n)$ *and* $\mathcal{E}[\![e_k]\!]\ \overrightarrow{v} = lift(u_k)$ *for* $k \in 1..n$.
- *A state transition sequence is a sequence (possibly infinite) of form:*

$$sts = (\mathtt{f}_0, \overrightarrow{v_0}) \to (\mathtt{f}_1, \overrightarrow{v_1}) \to (\mathtt{f}_3, \overrightarrow{v_3}) \to \ldots$$

  where $(\mathtt{f_t}, \overrightarrow{v_t})$ *are state transitions.*
- *The state transition sequence containing all function calls occurring during the computation of* $\mathtt{f}_{initial}$ *on the input* $\overrightarrow{x} \in Value^*$ *is noted:*

$$sts(\mathtt{P}, \overrightarrow{x}) = (\mathtt{f}_{initial}, \overrightarrow{x}) \to \ldots$$

- *We note $Sts(\mathtt{P})$ the set of all state transition sequences of computation starting at function $\mathtt{f}_{initial}$:*
$$Sts(\mathtt{P}) = \{sts(\mathtt{P}, \overrightarrow{x}) \mid \overrightarrow{x} \in Value^*\}$$

Note that a state transition sequence corresponding to a computation is finite if and only if the corresponding call sequence is finite.

2.4 Termination of first-order programs

First-order programs are interpreted by the call-by-value evaluation semantics given in definition 2.2.2. This semantics model defines formally the intuitive concept of termination for a first-order program. An important property required by this semantics is that all primitive operators (like addition on numbers) terminate.

We use the following notations:

- $\mathtt{P}\ \lightning \quad \triangleq \quad \forall x \in Value^* : \mathcal{E}[\![\mathtt{f}_{initial}]\!]\ \overrightarrow{x} \neq \bot$ (program P terminates on all input values),
- $\neg(\mathtt{P}\lightning) \quad \triangleq$ P does not terminate on all input values.

Example 2.4.1
The program h terminates although there is an infinite path in its control flow graph.

We already noticed that not all infinite paths in the flow graph correspond to real computation of the program. But if one of these infinite paths corresponds to a real computation then the corresponding computation does not terminate.

Conversely, a finite path represents a terminating computation. This is true since:

- we assumed that all primitive operators terminate,
- the language does not contain loop structures hence preventing the presence of infinite loop in the body of a function.

Hence, non-termination of a first-order program results from an infinite sequence of calls during the computation:

Proposition 2.4.1. *A program P terminates on all value of its inputs if and only if all state transition sequences starting from $\mathtt{f}_{initial}$ are finite. In other words:*

$$[\forall sts \in Sts(\mathtt{P}) \cdot |sts| < \infty] \quad \Longleftrightarrow \quad \mathtt{P}\lightning$$

The proof proceeds by analysis of the call-by-value semantics. See [6] for the details.

We now deduce the following corollary which gives a sufficient condition for a program to terminate:

Corollary 2.4.2. *Let P be a program. If none of the infintite paths in the control flow graph of* `P` *correspond to a valid computation then* `P` *terminates on all input values.*

Proof If none of the infinite path in its control flow graph correspond to a possible computation then all possible call sequences are finite. Therefore all possible state transition sequences are finite. Hence by proposition 2.4.1 the program terminates on all input value. ■

2.5 Size-change principle

2.5.1 Idea

We suppose that we are dealing with programs which function parameters belong to a well-founded set (in the first order case, the set $Value$ has to be well-founded).

We say that `P` satisfies the size-change termination condition (SCT) if every infinite call sequences following the control flow of `P` would cause an infinite descent in some of the program's data value.

Since infinite descents are not possible in well-founded set (proposition 2.1.1), this implies that no infinite path in the flow graph corresponds to a possible computation. By proposition 2.4.1, this implies that the program terminates on all input values.

Hence SCT guarantees termination.

The size-change principle is based on a theorem (2.5.2 stated in the next section), which states that if the set of calls in the flow graph of the program verifies a certain property (all loops are descending), then the program verifies the size-change termination property and therefore it terminates for all input values.

Before establishing this theorem, we need to define a new kind of objects: size-change graphs. A size-change graph describes a call in a program. Among all the size-change graphs describing a call, we are interested in those which give us accurate information about the call. We say that these graphs *safely* describes the call.

This section gives formal definitions for size-change graphs, the safety property and the size-change termination condition.

2.5.2 Size-change graphs

We use the definition given in [5] (the one given in [6] is less general than this one):

Definition 2.5.1 (Size-change graph).

- *A size-change graph $A \xrightarrow{G} B$ consists of a source set A, a target set B and a set of labeled arcs G:*

$$A \xrightarrow{G} B = (A, B, G)$$

$$G \subset A \times \{\downarrow, =\} \times B$$

where G does not contain two arrows with same endings and different labels.

The graph is identified with its arc set G when A and B are clear from context.

- *An arc $(x, =, y) \in G$ is noted $x \overset{=}{\to} y$*
- *An arc $(x, \downarrow, y) \in G$ is noted $x \overset{\downarrow}{\to} y$*
- *A size-change graph containing at least one arc of type $x \overset{\downarrow}{\to} x$ is said to be* ***descending****.*

We now relate calls and size-change graphs (see definition 12 of [5]):

Definition 2.5.2 (SCG describing a call)**.**

1. *For $p \in \mathcal{P}$ we associate $gb(p)$, the* ***graph-basis*** *of the program point p.*
2. *A size-change graph* ***describes a call*** *$p_1 \to p_2$ (or a transitive call $p_1 \to^* p_2$) if its source set is $gb(p_1)$ and its target set is $gb(p_2)$.*

To understand these definitions, let us consider the case of first-order programs. The definition of a size-change graph for a first-order program given in [6] can be restated as follow:

Definition 2.5.3 (First order size-change graph)**.** *Let* `f`, `g` *be function names in program* `P`.

A size-change graph from `f` *to* `g` *written $Param(\mathtt{f}) \xrightarrow{G} Param(\mathtt{g})$ describes a call from function* `f` *to function* `g`*. It is a bipartite graph relating the parameters of* `f` *to those of* `g` *with labeled-arc set*

$$G \subset Param(\mathtt{f}) \times \{\downarrow, =\} \times Param(\mathtt{g})$$

where G does not contain two arrows with same endings and different labels.
The graph is identified with its arc set G when `f` *and* `g` *are clear from context.*

We remark that this definition for first-order programs is equivalent to definition 2.5.1 if we take:

- $\mathcal{P}$ = set of function names in `P`
- and for $\mathtt{f} \in \mathcal{P}$, $gb(\mathtt{f}) = Param(\mathtt{f})$

Example 2.5.1 gcd
The following program computes the greatest common divisor of two numbers using Euclid algorithm.

```
gcd(x,y) = if y = 0 then x else gcd(y, (x mod y))
```

We have $gb(\texttt{gcd}) = \{\texttt{x}, \texttt{y}\}$.

Each of the following size-change graphs describes the recursive call occuring in `gcd`:

$$\left(gb(\texttt{gcd}), gb(\texttt{gcd}), \left\{y \overset{=}{\to} x, y \overset{\downarrow}{\to} y\right\}\right)$$

$$\left(gb(\texttt{gcd}), gb(\texttt{gcd}), \left\{y \overset{\downarrow}{\to} x, y \overset{\downarrow}{\to} y\right\}\right)$$

However we will see later that the second one does not *safely* describes the call (the safety property is defined in section 2.5.4).

2.5.3 Composition of size-change graphs

If G represents the call $f \to g$ and G' the call from $g \to h$ then we want to be able to construct a graph representing the transitive call from f to h. The graph we are looking for is the composition of G by G' noted $G; G'$ and defined as follow: (our definition is equivalent to the definition 14 of [6] and definition 1 of [5])

Definition 2.5.4 (Size-change graph composition)**.**

1. *Size-change graphs $A \overset{G_1}{\to} B$ and $C \overset{G_1}{\to} D$ are composible if $B = C$.*

2. *The sequential composition of size-change graphs $A \overset{G_1}{\to} B$ and $B \overset{G_1}{\to} D$ is $A \overset{G_1;G_2}{\longrightarrow} D$ where*

$$\begin{aligned} G_1; G_2 &= \{x \overset{\downarrow}{\to} z \mid \exists y \cdot x \overset{\downarrow}{\to} y \overset{r}{\to} z \quad \vee \quad x \overset{r}{\to} y \overset{\downarrow}{\to} z\} \\ &\cup \{x \overset{=}{\to} z \mid \exists y \cdot x \overset{=}{\to} y \overset{=}{\to} z \quad \wedge \quad (x \overset{\downarrow}{\to} z) \notin G'\} \end{aligned}$$

using the following notation: $x \overset{r}{\to} y \overset{r'}{\to} z \triangleq \left[x \overset{r}{\to} y \in G_1 \text{ and } y \overset{r'}{\to} z \in G_2\right]$

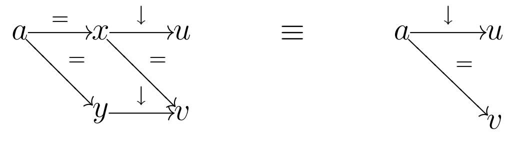
Example 2.5.2 Graph composition for first order programs

Suppose that $G_1 = \{a \overset{=}{\to} x, a \overset{=}{\to} y\}$ describes the call $f \overset{c_1}{\to} g$ and $G_2 = \{x \overset{=}{\to} v, x \overset{\downarrow}{\to} u, y \overset{\downarrow}{\to} v\}$ describes $g \overset{c_2}{\to} h$ then

$$G_1; G_2 = \{a \overset{=}{\to} v, a \overset{\downarrow}{\to} u\}$$

describes the transitive call $f \overset{\langle c_1 c_2 \rangle}{\longrightarrow} h$.

The following diagrams illustrate how the composition is computed:



Definition 2.5.5 (Size-change graphs describing a program)**.** *A set of size-change graphs $\mathcal{G}$ describes a program* `P` *if*

$$\mathcal{G} = \{G_c \mid c \textit{ is an activable call in } \texttt{P}\}$$

and if for every activable call c in `P` *the graph $G_c \in \mathcal{G}$ describes c.*

Definition 2.5.6 (Size-change graph describing a loop)**.** *A size-change graph G* ***describes a loop*** *(or is a loop size-change graph) if G describes a recursive transitive call (definiton 2.3.2).*

If moreover $G; G = G$, we say that G ***describes asymptotically*** *the loop.*

The condition $G; G = G$ in the definition 2.5.6 means that the description of the call is still valid after after any number of iterations of the loop: suppose that G describes $\mathtt{f} \overset{cs}{\rightarrow} \mathtt{f}$ then it describes the size-relation between elements of $Param(\mathtt{f})$ after any number of repetitions of the call sequence cs.

Definition 2.5.7 (Closure by composition)**.** *Let $\mathcal{G}$ be a set of size-change graphs. The closure by composition of $\mathcal{G}$ noted $\overline{\mathcal{G}}$ is defined as follow:*

$$\overline{\mathcal{G}} = \bigvee \left\{ \mathcal{H} \textit{ such that } \mathcal{H} \supseteq \mathcal{G} \textit{ and } \begin{pmatrix} \mathtt{f} \overset{G_{cs_1}}{\rightarrow} \mathtt{g} \in \mathcal{H} \\ \mathtt{g} \overset{G_{cs_2}}{\rightarrow} \mathtt{h} \in \mathcal{H} \end{pmatrix} \Longrightarrow \mathtt{f} \xrightarrow{G_{cs_1 \hat{}\, cs_2}} \mathtt{h} \in \mathcal{H} \right\}$$

where $G_{c_1 \hat{}\, c_2} = G_{c_1}; G_{c_2}$

2.5.4 Safe size-change graphs

We expect size-change graphs to safely describe what really happens during a call in the program.

This means that each arc $a \overset{=/\downarrow}{\rightarrow} b$ in the graph should provide a sound characterization of the data-size relation between object a and object b during the evaluation of the program. An arrow labeled ↓ should mean that at running time, the size of the value represented by b is strictly less than the size of the value represented by a. Likewise, an arrow labeled = should mean that there is no increase in the data-size. If these conditions are realized then we say that the arc **safely describes the call**.

In the case of first-order programs, a possible definition could be:

Definition 2.5.8 (Arc safely describing a call in the first-order case)**.** *An arc $a = x \overset{=/\downarrow}{\rightarrow} y$ safely describes a call $f \rightarrow g$ if*

- *$a = x \overset{\downarrow}{\rightarrow} y$ and the value used as parameter y in the call to function g is always strictly less than the value of the input parameter x of f.*

- *or $a = x \overset{=}{\rightarrow} y$ and the value used as parameter y of g is never greater than the value of the input parameter x of f.*

From now on, we assume that the notion of safety for a size-change graph arc has been chosen. Based on the definition of arc safety, we then define the safety property for a size-change graph and for a set of size-change graphs:

Definition 2.5.9 (Safety properties)**.** *Consider the following set of size-change graphs describing* `P`*:*

$$\mathcal{G} = \{G_c \mid c \textit{ is an activable call in } \texttt{P}\}$$

Then:

1. *the size-change graph $A \overset{G_c}{\rightarrow} B$ is **safe for** c if every arc in G_c safely describes the size relation in the call,*

2. *the set $\mathcal{G}$ is a **safe description of program** `P` if for every activable call c, G_c is safe for c.*

3. *By extension we say that $A \overset{G}{\rightarrow} B$ is **safe for a well formed sequence of calls** $cs = \langle c_0 c_1 \ldots c_k \rangle$ if every arc in G safely describes the size relation in the transitive call $\cdot \overset{c_0}{\rightarrow} \cdot \overset{c_1}{\rightarrow} \ldots \overset{c_k}{\rightarrow} \cdot$.*

Note that the safety property for a set of size-change graphs requires only activable calls to be decribed (this definition is therefore similar to definition 12 in [5] but different from definition 9 of [6] where all calls in `P` need to be described).

Example 2.5.3 gcd (example 2.5.1 continued)
The following size-change graphs safely describe the recursive call occuring in `gcd` :

$$G_1 = \{y \overset{=}{\rightarrow} x, y \overset{\downarrow}{\rightarrow} y\} \quad G_2 = \{y \overset{=}{\rightarrow} x, y \overset{=}{\rightarrow} y\}$$
$$G_3 = \{y \overset{=}{\rightarrow} x\} \qquad G_4 = \{y \overset{=}{\rightarrow} y\} \qquad G_5 = \emptyset$$

G_1 is a safe description because $x \mod y < y$.

While all these graphs safely describe the call, G_1 is the most accurate one (it gives more information about the call).

For $i \in \{1..5\}$, the set $\mathcal{G}_i = \{G_i\}$ of size-change graphs is a safe description of program `gcd`.

2.5.5 Size-change termination condition

Definition 2.5.10 (Multipath and thread)**.**

- *a **multipath** is a finite or infinite sequence of size-change graphs where consecutive graphs are compatible for composition. This sequence can be viewed as a concatenation (possibly infinte) of graphs.*

- *a **thread** in a multipath is a connected path of arcs (of any length) in the multipath.*

- *a thread is **descending** if at least one of the arc in the sequence is labelled with* $\downarrow$

- *a thread is **infinitely descending** if it contains infinitely many arcs labelled with* $\downarrow$*.*

We define the size-change condition property relatively to a safe set of size-change graphs:

Definition 2.5.11 ($\mathcal{G}$-SCT)**.** *A program* `P` *is* $\mathcal{G}$*-SCT (for* $\mathcal{G}$*-size-change terminating) if*

- $\mathcal{G}$ *safely describes* `P`

- *for all infinite call sequences* $cs = \langle c_0 c_1 \ldots \rangle$*, the multipath* $G_{c_0} G_{c_1} \ldots$ *has an infinitely descending thread.*

Theorem 2.5.1 ($\mathcal{G}$-SCT and termination)**.**

$$\mathtt{P}\ \textit{is}\ \mathcal{G}\textit{-SCT} \implies \mathtt{P}\lightning$$

A proof of this theorem is given in [6].

2.5.6 Deciding SCT

We know that the SCT property can be used to prove that a program terminates on all input value. The problem is now to decide whether the SCT holds for a particular program. In this section, we will give an algorithm which decides SCT.

The following theorem gives a characterization of the $\mathcal{G}$-size-change termination condition:

Theorem 2.5.2 (Characterization of $\mathcal{G}$-SCT)**.** *Let* `P` *be a program,* $\mathcal{G}$ *a set of size-change graphs which **safely** describes* `P`*. Then* `P` *is **not*** $\mathcal{G}$*-size-change terminating if and only if* $\overline{\mathcal{G}}$ *contains a non-descending asymptotic loop size-change graph:*

$$\mathtt{P}\ \textit{is}\ \textbf{\textit{not}}\ \mathcal{G}\textit{-SCT} \iff \exists \mathtt{f} \xrightarrow{G} \mathtt{f} \in \overline{\mathcal{G}}\ \textit{such that}\ \begin{pmatrix} G;G = G \\ \forall \mathtt{x} \in gb(\mathtt{f}) : \mathtt{x} \xrightarrow{\downarrow} \mathtt{x} \notin G \end{pmatrix}$$

A proof of this theorem is given in [6].

This theorem leads us to the following algorithm for deciding $\mathcal{G}$-SCT.

Algorithm 2.5.1 (Size-change principle algorithm)**.** The following steps describes an algorithm which decides $\mathcal{G}$-SCT. We assume that a known algorithm, noted $\mathcal{A}_{safearcs}$, can generate a safe description of any call occurring in a program.

1. By syntax analysis, the flow graph of the program is determined. For each activable call in this flow graph, we associate the corresponding *size-change graphs* (SCG) using the algorithm $A_{safearcs}$ to generate its arcs. We obtain a set $\mathcal{G}$ of size-change graphs which safely describes the program `P`.

2. Build $\overline{\mathcal{G}}$, the closure of $\mathcal{G}$ by size-change graph composition.

3.
   ```
   If there is a graph $G \in \overline{\mathcal{G}}$:
     $G$ describes a recursive call $f \to f$
     and $G$ is not descending,
     and $G = G; G$ then
       return "P is not G-size-change terminating"
   else "P is G-size-change terminating"
   ```

Note that this algorithm does not decide the size-change termination condition in general but the $\mathcal{G}$-SCT condition.

The power of this new algorithm directly depends on our ability to generate accurate safe size-change graphs. The naive approach for $A_{safearc}$ consisting in generating size-change graphs with no arcs is clearly not satisfactory.

We want the algorithm $A_{safearc}$ to generate as many safe arcs $x \overset{\downarrow}{\to} y$ as possible. This way, we reduce the chance that the closure $\overline{\mathcal{G}}$ contains a non-descending size-change graph describing a loop.

Complexity of the algorithm The generation of the closure of the set $\mathcal{G}$ of size-change graphs is expensive in time (the number of possible compositions can be exponential in the size of the input program size).

The $\mathcal{G}$-SCT decision problem happens to be PSPACE complete. A proof of this is given in [6].

Chapter 3

The size-change principle in the untyped λ-calculus

The size-change principle described in the previous chapter can now be adapted to the untyped λ-calculus case.

Recall that the size-change principle method relies on definitions of the following notions:

1. the language (its syntax, its semantics)
2. termination of the program computation
3. a finite program points space $\mathcal{P}$
4. notion of call such that non-termination is charaterized by the presence of infinite call sequences
5. the graph-basis function $gb(p)$ for $p \in \mathcal{P}$
6. a size for the parameter values (a well-founded order).
7. definition of the safety property for arcs of size-change graphs.
8. the algorithm $A_{safearcs}$ used to generate arcs which safely describe calls in the program.

In the following sections we explain how each of these notions are defined in the λ-calculus case. Finally an algorithm will be derived for termination analysis in the λ-calculus.

3.1 The untyped λ-calculus

We assume that the reader is familiar with the basics notion of the untyped λ-calculus. A complete treatment can be found in the Bible of the λ-calculus by Henk Barendregt [1]. [2] is one possible introduction to λ-calculus.

We recall basic definitions:

Definition 3.1.1 (λ-calculus basic definitions)**.** *The set of λ-expressions is defined by the following grammar:*

$$\mathsf{e} ::= \mathtt{x} \mid \mathsf{e} \mathbin{@} \mathsf{e} \mid \lambda \mathtt{x}.\mathsf{e}$$

$$\mathtt{x} ::= \textit{variable name}$$

The set of free variable of an expression noted $fv(\mathsf{e})$ is defined by:

$$\begin{array}{rcl} fv(\mathtt{x}) & = & \{\mathtt{x}\} \\ fv(\mathsf{e} \mathbin{@} \mathsf{e}') & = & fv(\mathsf{e}) \cup fv(\mathsf{e}') \\ fv(\lambda\mathtt{x}.\mathsf{e}) & = & fv(\mathsf{e}) \setminus \{\mathtt{x}\} \end{array}$$

e *is a closed λ-expressions if $fv(\mathsf{e}) = \emptyset$.*
The set of subexpressions of a λ-expressions e is noted $subexp(\mathsf{e})$ and is defined by:

$$\begin{array}{rcl} subexp(\mathtt{x}) & = & \{\mathtt{x}\} \\ subexp(\mathsf{e} \mathbin{@} \mathsf{e}') & = & \{\mathsf{e} \mathbin{@} \mathsf{e}'\} \cup subexp(\mathsf{e}) \cup subexp(\mathsf{e}') \\ subexp(\lambda\mathtt{x}.\mathsf{e}) & = & \{\lambda\mathtt{x}.\mathsf{e}\} \cup subexp(\mathsf{e}) \end{array}$$

A program is a closed λ-expressions.

3.2 Termination in the untyped λ-calculus

Usually, non-termination in the untyped lambda calculus is expressed by the fact that a given expression contains no redex. Here we rely on the call-by-value semantics to define evaluation of a lambda expression and we say that a program does not terminate if there is no evaluation of it.

As we did for the first order case, we define the call-by-value semantics for the untyped λ-calculus. We use the judgement form $e \Downarrow v$ to denote that expression e evaluates to the value v.

Definition 3.2.1 (Call-by-value semantics)**.** *The call-by-value evaluation is defined by the following inference rules where ValueS is the set of all abstractions (expressions of type* $\lambda\mathtt{x}.\mathsf{e}$*) :*

$$(\text{ValueS}) \; \frac{}{v \Downarrow v} \; (\text{If } v \in \textit{ValueS}) \qquad\qquad (\text{ApplyS}) \; \frac{\mathsf{e}_1 \Downarrow \lambda\mathtt{x}.\mathsf{e}_0 \quad \mathsf{e}_2 \Downarrow v_2 \quad \mathsf{e}_0[v_2/\mathtt{x}] \Downarrow v}{\mathsf{e}_1 @ \mathsf{e}_2 \Downarrow v}$$

A consequence of the definition of rule (ValueS) is that any abstraction terminates (even if it contains redex).

As an example the term $\Omega = (\lambda x.xx)(\lambda x.xx)$ does not terminate while the term $\lambda x.\Omega$ does.

Definition 3.2.2 (Termination notation)**.** *We use the following abbreviations:*

$$\begin{array}{lcl} \mathtt{e} \Downarrow & \triangleq & \exists v \in ValueS \cdot \mathtt{e} \Downarrow \mathtt{v} \\ \mathtt{e} \not\Downarrow & \triangleq & \neg(\mathtt{e} \Downarrow) \end{array}$$

3.3 Program control points

In contrast with the previous chapter, we are dealing here with higher-order programs. Consequently, a function parameter can be itself a function. Hence a program can make a call to a function passed as a parameter! For this reason, the program's control points set $\mathcal{P}$ cannot be represented by function names as we did in the previous chapter.

In the λ-calculus case, program points are the subexpressions of the program expression (i.e. the control flow graph nodes are the subexpressions of $\mathtt{P}$):

$$\mathcal{P} = subexp(\mathtt{P})$$

3.4 Calls

The size-change principle relies on the analysis of calls occurring in a program. We need to define what a call is in the untyped λ-calculus.

In [5] the following notion of *call* is used for the untyped λ-calculus:

> *There is a call from expression* $\mathtt{e}$ *to expression* $\mathtt{e}'$ *(noted $e \to e'$) if in order to deduce $e \Downarrow v$ for some value v, it is necessary to first deduce $e' \Downarrow u$ for some u.*

The formal definition is given using inference rules as follow:

Definition 3.4.1 (The call relation for the untyped λ-calculus)**.** *We define the call relation on the set of λ-expression $\to \subset Exp \times Exp$. The call relation is $\to = \underset{r}{\to} \cup \underset{d}{\to} \cup \underset{c}{\to}$ where $\underset{r}{\to}, \underset{d}{\to}, \underset{c}{\to}$ are defined by the following inference rules:*

$$\text{(OperatorS)}\ \frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\to} \mathtt{e}_1} \qquad \text{(OperandS)}\ \frac{\mathtt{e}_1 \Downarrow v_1}{\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\to} \mathtt{e}_2} \qquad \text{(CallS)}\ \frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \qquad \mathtt{e}_2 \Downarrow v_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0[v_2/\mathtt{x}]}$$

Letters c, r and d stand respectively for Call, operatoR and operanD.

An important result of [5] is that non-termination in the untyped lambda-calculus is characterized by the presence of infinite call chains in the program. This property is a requirement for the size-change principle.

Lemma 3.4.1 (Nontermination Is Sequential [5]). *Let* P *be a program. Then:*

$$\mathtt{P} \not\Downarrow \quad \Longleftrightarrow \quad \mathtt{P} = \mathtt{e}_0 \rightarrow \mathtt{e}_1 \rightarrow \mathtt{e}_2 \rightarrow \ldots$$

This lemma is the counterpart of proposition 2.4.1 (in the first-order case) for the untyped λ-calculus.

3.5 Semantics describing the computation space

In order to describe the computation space Neil D. Jones proposed to replace the semantics for the untyped-lambda calculus by an equivalent one describing more precisely the computation space. This new semantics is based on the use of environments.

An environment records the value associated to each free variable of an expression. The use of environments permits us to keep subexpressions unmodified in the judgment forms. Every state is now described by a subexpression of P and by an environment (also called closure) which associate a value to each of the free variables of the subexpression. Recall that subexpressions are the control points, this is the reason why we want to keep them unmodified during the application of the rules. Only the environment part of the state will be updated.

The formal definition of states, values and environments are given in the following definition:

Definition 3.5.1 (State, Value, Environment). *The sets State, $Value$, Env are the smallest sets verifying the following equation:*

$$\begin{array}{rcl} State & = & \{e : \rho \mid e \in Exp, \rho \in Env, fv(e) \subseteq dom(\rho)\} \\ Value & = & \{e : \rho \mid \lambda x, e : \rho \in State\} \\ Env & = & \{p : X \rightarrow Value \mid X \textit{ finite set of variables}\} \end{array}$$

The empty environment with domain $X = \emptyset$ is written [].

Definition 3.5.2 (Environment-based semantics). *The judgement form are $s_1 \Downarrow v$ and $s_1 \rightarrow s_2$ where $s_1, s_2 \in State$ and $v \in Value$.*

The evaluation and call relations $\Downarrow, \rightarrow$ are defined by the following inference rules, where $\rightarrow\, = \underset{r}{\rightarrow} \cup \underset{d}{\rightarrow} \cup \underset{c}{\rightarrow}$.

$$\text{(Value)}\ \frac{}{v \Downarrow v}\ (\text{If } v \in Value) \qquad \text{(Var)}\ \frac{}{\mathtt{x}\!: \rho \Downarrow \rho(\mathtt{x})}$$

$$\text{(Operator)}\ \frac{}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{r}{\rightarrow} \mathtt{e}_1 : \rho} \qquad \text{(Operand)}\ \frac{\mathtt{e}_1 : \rho \Downarrow v_1}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{d}{\rightarrow} \mathtt{e}_2 : \rho}$$

$$\text{(Call)}\ \frac{\mathtt{e}_1 : \rho \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 : \rho_0 \qquad \mathtt{e}_2 : \rho \Downarrow v_2}{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2]} \qquad \text{(Apply)}\ \frac{\mathtt{e}_1@\mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}' : \rho' \qquad \mathtt{e}' : \rho' \Downarrow v}{\mathtt{e}_1@\mathtt{e}_2 : \rho \Downarrow v}$$

Note that the substitution occurring in the rules (ApplyS) has been now replaced by an update of the environment in the (Call) rule.

We can map each state to an expression with no free variables by replacing recursively the free variables by their associated expressions in the environment. This mapping is given in [5]: $F : \mathit{Exp} \times \mathit{Env} \rightarrow \mathit{Exp}$ defined as

$$F(\mathtt{e} : \rho) = \mathtt{e}[F(\rho(\mathtt{x}_1))/\mathtt{x}_1, ..., F(\rho(\mathtt{x}_k))/\mathtt{x}_k] \text{ where } \{\mathtt{x}_1, .., \mathtt{x}_k\} = \mathit{dom}(\rho) \cap fv(e)$$

As a result, the environment based semantics is equivalent to the standard one. Indeed $\mathtt{P} : [] \Downarrow v$ relatively to definition 3.5.2 if and only if $\mathtt{P} \Downarrow F(v)$ relatively to the standard semantics (see [5]).

Consequently, the lemma 3.4.1 is still valid with this new semantics:

Lemma 3.5.1. *Let* P *be a program. Then:*

$$\mathtt{P} : [] \not\Downarrow \quad \Longleftrightarrow \quad \mathtt{P} : [] = \mathtt{e}_0 : \rho_0 \rightarrow \mathtt{e}_1 : \rho_1 \rightarrow \mathtt{e}_2 : \rho_2 \rightarrow \ldots$$

This lemma is a key element in the size-change principle: the main theorem of the size-change principle (2.5.1) is based on it.

Remark 3.5.1. In section 3.3, we defined control points in the λ-calculus as beeing subexpressions of the program P. This choice is motivated by the following property of the environment semantics:

Proposition 3.5.2 (Subexpression property). *If* $\mathtt{P} : [] \Downarrow \lambda x.e : \rho$ *then* $\lambda x.e \in \mathit{subexp}(\mathtt{P})$

This is proved in [5] (by first introducing the notion of expression support).

Hence for the λ-calculus we choose:

$$\mathcal{P} = \mathit{subexp}(\mathtt{P})$$

The important fact is that $\mathit{subexp}(\mathtt{P})$ is finite: this allow us to use the size-change principle.

3.6 Size-change graphs

The program points space $\mathcal{P}$ has been defined in the previous section.

We define the graph-basis function as follow:

Definition 3.6.1 (Graph-basis in the λ-calculus)**.** *The graph basis of* $\mathtt{e} \in \mathcal{P} = \mathit{subexp}(\mathtt{P})$ *is*

$$gb(\mathtt{e}) = fv(\mathtt{e}) \cup \{\bullet\}$$

Remark 3.6.1. This graph-basis definition is the counterpart of the one used in the first-order case ($gb(\mathtt{f}) = \mathit{Input}(\mathtt{f})$). In the λ-calculus, function parameters have been replaced by free variables.

Note that an extra element $\bullet$ has been added. It represents the expression e itself. Later we will see how we use this element.

The next step consists in defining the notion of size:

Definition 3.6.2 (State support). *The support of a state $s = \mathtt{e} : \rho$ is*

$$support(\mathtt{e} : \rho) = \{\mathtt{e} : \rho\} \cup \bigcup_{\mathtt{x} \in fv(\mathtt{e})} support(\rho(\mathtt{x}))$$

Definition 3.6.3 (Size relation). *Suppose that $s_1 = e_1 : \rho_1$ and $s_2 = e_2 : \rho_2$ then*

$$s_1 \succeq s_2 \quad \Longleftrightarrow \quad \left\{ \begin{array}{ll} & support(s_1) \ni s_2 \\ or & subexp(e_1) \ni e_2 \textit{ and } \rho_1(x) = \rho_2(x) \textit{ for all } x \in fv(e_2) \end{array} \right.$$

We write $s_1 \succ s_2$ if $s_1 \succeq s_2$ and $s_1 \neq s_2$.

The size relation $\succ$ is an order and it is well-founded (on the set $State \times State$). In the next section we define the safety property by using this notion of size.

3.7 Safety property

We first define the valuation function which maps each graph-basis element to a state in the set *State*. The special element • is mapped to the expression component of the state itself and the free variables are mapped to their associated state in the environment. We say that an arc is safe if it measures the size change of these valuated elements during a call (relatively to the notion of size defined in definition 3.6.3):

Definition 3.7.1 (Safety property in the λ-calculus).

- *For every $s = \mathtt{e} : \rho \in State$, the **valuation** function for s noted $\overline{s} : gb(\mathtt{e}) \longrightarrow Value$, is defined as follow:*

$$\overline{s}(\bullet) = s \quad \text{and} \quad \overline{\mathtt{e} : \rho}(\mathtt{x}) = \rho(\mathtt{x})$$

- *An arc $a = x \overset{=/\downarrow}{\rightarrow} y \in G$ is **safe** for (s_1, s_2) if*
  - *$a = x \overset{=}{\rightarrow} y$ and $\overline{s_1}(x) \succeq \overline{s_2}(y)$*
  - *or $a = x \overset{\downarrow}{\rightarrow} y$ and $\overline{s_1}(x) \succ \overline{s_2}(y)$*

  By extension we say that an arc safely describes the call $s_1 \rightarrow s_2$ if it is safe for (s_1, s_2).

The definitions given in section 2.5.4 for safe size-change graphs, safe sets of size-change graphs and the $\mathcal{G}$-SCT condition (definition 2.5.11 and 2.5.9) are now used relatively to this definition of safe arcs.

Moreover we saw in lemma 3.4.1 that non-termination in the untyped lambda-calculus is characterized by the presence of infinite call chains in the program. Consequently, theorem 2.5.1 is still valid in the λ-calculus case.

The characterization theorem 2.5.2 is also valid in the λ-calculus case.

Hence we can apply the last two parts of algorithm 2.5.1 to detect termination in the λ-calculus programs. The remaining difficulty is to find how to generate a safe set of size-change graphs describing the calls of the program.

3.8 Graph generation (algorithm $A_{safearcs}$)

The environment-based semantics of the language given in definition 3.5.2 is now modified in order to generate safe graphs during the application of each rule.

Definition 3.8.1 (Environment-base semantics with graph generation). *The evaluation and call judgement forms are now* $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho', G$ *and* $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho', G$ *where* G *is the generated graph. The inference rules are:*

$$\text{(ValueG)} \quad \frac{}{\lambda \mathtt{x}.\mathtt{e} : \rho \Downarrow \lambda \mathtt{x}.\mathtt{e} : \rho, id^{=}_{\lambda \mathtt{x}.\mathtt{e}}}$$

$$\text{(VarG)} \quad \frac{}{\mathtt{x} : \rho \Downarrow \rho(\mathtt{x}), \{\mathtt{x} \overset{=}{\rightarrow} \bullet\} \cup \{\mathtt{x} \overset{\downarrow}{\rightarrow} \mathtt{y} \mid \mathtt{y} \in \mathit{fv}(\mathtt{e}')\}} \; (\text{If } \rho(\mathtt{x}) = \mathtt{e}' : \rho')$$

$$\text{(OperatorG)} \quad \frac{}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{r}{\rightarrow} \mathtt{e}_1 : \rho, id^{\downarrow}_{\mathtt{e}_1}} \qquad \text{(OperandG)} \quad \frac{\mathtt{e}_1 : \rho \Downarrow v_1}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{d}{\rightarrow} \mathtt{e}_2 : \rho, id^{\downarrow}_{\mathtt{e}_2}}$$

$$\text{(CallG)} \quad \frac{\mathtt{e}_1 : \rho \Downarrow \lambda \mathtt{x}.\mathtt{e}_0 : \rho_0, G_1 \qquad \mathtt{e}_2 : \rho \Downarrow v_2, G_2}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto v_2], G_1^{-\bullet} \cup G_2^{\bullet \mapsto \mathtt{x}}}$$

$$\text{(ApplyG)} \quad \frac{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \underset{c}{\rightarrow} \mathtt{e}' : \rho', G' \qquad \mathtt{e}' : \rho' \Downarrow v, G}{\mathtt{e}_1 @ \mathtt{e}_2 : \rho \Downarrow v, (G'; G)}$$

where

$$\begin{aligned}
id^{=}_{\mathtt{e}} &\triangleq \{\bullet \overset{=}{\rightarrow} \bullet\} \cup \{\mathtt{x} \overset{=}{\rightarrow} \mathtt{x} \mid \mathtt{x} \in \mathit{fv}(\mathtt{e})\} \\
id^{\downarrow}_{\mathtt{e}} &\triangleq \{\bullet \overset{\downarrow}{\rightarrow} \bullet\} \cup \{\mathtt{x} \overset{=}{\rightarrow} \mathtt{x} \mid \mathtt{x} \in \mathit{fv}(\mathtt{e})\} \\
G_1^{-\bullet} &\triangleq \{\ \mathtt{y} \overset{r}{\rightarrow} \mathtt{z} \mid\ \mathtt{y} \overset{r}{\rightarrow} \mathtt{z} \in G_1\} \cup \{\ \bullet \overset{\downarrow}{\rightarrow} \mathtt{z} \mid\ \bullet \overset{r}{\rightarrow} \mathtt{z} \in G_1\} \\
G_2^{\bullet \mapsto \mathtt{x}} &\triangleq \{\ \mathtt{y} \overset{r}{\rightarrow} \mathtt{x} \mid\ \mathtt{y} \overset{r}{\rightarrow} \bullet \in G_2\ \} \cup \{\ \bullet \overset{\downarrow}{\rightarrow} \mathtt{x} \mid\ \bullet \overset{r}{\rightarrow} \bullet \in G_2\ \}
\end{aligned}$$

Each graph extracted from this semantics is safe for its two associated program subexpressions (see theorem 2 in [5]).

3.9 Abstraction of the semantics (as in [5])

Remember that when we explained the size-change principle in the first order case, the computation was exactly described using state transition sequences (definition 2.3.3).

After removing the $Value$ component of states in transition sequences we obtain call sequences. Call sequences are paths in the control flow graph of the program (which is determined at the syntax level).

The effect of abstracting state transition sequences by call sequences is to transform an infinite graph into a finite one (indeed $\mathcal{P}$ is a finite set).

The size-change principle is based on the finiteness of this approximation. The fact that, in a finite graph, infinite paths must contain loops (see theorem 2.5.2) permits us to analyze the size-change condition in terms of presence of particular loops in the program.

We therefore need to find a similar approximation for the λ-calculus case.

The approximation proposed in [5] consists in removing the environment components in the semantics of the language. This reduces dramatically the number of possible judgment forms which can be generated by the semantics. Indeed, since there are finitely many subexpressions of the program expression, after removing the environment components there are only finitely many possible judgements of type $\mathtt{e} \to \mathtt{e}'$ or $\mathtt{e} \Downarrow \mathtt{e}'$.

The absence of environments forces us to change the way we deal with the variable look-up in the (Var) rule. In [5] this problem is solved by over-approximating the rule (Var) with a new rule (VarA). With this new rule, the variable $\mathtt{x}$ may now evaluate to several possible values.

Precisely, if the program expression contains an application $\mathtt{e}_1\ \mathtt{e}_2$ where $\mathtt{e}_1$ evaluates to $\lambda\mathtt{x}.\mathtt{e}$ and $\mathtt{e}_2$ evaluates to v_2 then we deduce that $\mathtt{x} \Downarrow v_2$.

The formal definition of the approximate semantics is given below:

Definition 3.9.1 (Approximate semantics)**.** *The judgement forms are* $\mathtt{e} \Downarrow \mathtt{e}'$ *and* $\mathtt{e} \to \mathtt{e}'$. *The inference rules are:*

(ValueA) $$\frac{}{\lambda\mathtt{x}.\mathtt{e} \Downarrow \lambda\mathtt{x}.\mathtt{e}}$$

(VarA) $$\frac{\mathtt{e}_1@\mathtt{e}_2 \in \mathit{subexp}(\mathtt{P}) \quad \mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{x} \Downarrow v_2}$$

(OperatorA) $$\frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\to} \mathtt{e}_1}$$

(OperandA) $$\frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\to} \mathtt{e}_2}$$

(CallA) $$\frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}_0}$$

(ApplyA) $$\frac{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\to} \mathtt{e}' \quad \mathtt{e}' \Downarrow v}{\mathtt{e}_1@\mathtt{e}_2 \Downarrow v}$$

Assuming that $\mathtt{P} : [] \to^* \mathtt{e} : \rho$ then $\mathtt{e} : \rho \to \mathtt{e}' : \rho$ implies $e \to e'$ and $\mathtt{e} : \rho \Downarrow \mathtt{e}' : \rho$ implies $e \Downarrow e'$.

This property justifies that the new rules are over-approximating the original semantics.

The approximate semantics of the language can now be extended in order to generate safe graphs during the application of each rule (as we did for the exact semantics in definition 3.8.1):

Definition 3.9.2 (Approximated semantics with graph generation)**.** *The judgement forms are now* $\mathtt{e} \to \mathtt{e}', G$ *and* $\mathtt{e} \Downarrow \mathtt{e}', G$.

$$\text{(ValueAG)}\ \frac{}{\lambda\mathtt{x}.\mathtt{e} \Downarrow \lambda\mathtt{x}.\mathtt{e}, id^{=}_{\lambda\mathtt{x}.\mathtt{e}}} \qquad \text{(VarAG)}\ \frac{\mathtt{e}_1@\mathtt{e}_2 \in \mathit{subexp}(\mathtt{P}) \quad \mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0, G_1 \qquad \mathtt{e}_2 \Downarrow v_2, G_2}{\mathtt{x} \Downarrow v_2, \{\mathtt{x} \overset{=}{\rightarrow} \bullet\} \cup \{\mathtt{x} \overset{\downarrow}{\rightarrow} \mathtt{y} \mid \mathtt{y} \in \mathit{fv}(v_2)\}}$$

$$\text{(OperatorAG)}\ \frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{r}{\rightarrow} \mathtt{e}_1, id^{\downarrow}_{\mathtt{e}_1}} \qquad \text{(OperandAG)}\ \frac{}{\mathtt{e}_1@\mathtt{e}_2 \underset{d}{\rightarrow} \mathtt{e}_2, id^{\downarrow}_{\mathtt{e}_2}}$$

$$\text{(CallAG)}\ \frac{\mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e}_0, G_1 \qquad \mathtt{e}_2 \Downarrow v_2, G_2}{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\rightarrow} \mathtt{e}_0, G_1^{-\bullet} \cup G_2^{\bullet\mapsto\mathtt{x}}} \qquad \text{(ApplyAG)}\ \frac{\mathtt{e}_1@\mathtt{e}_2 \underset{c}{\rightarrow} \mathtt{e}', G' \qquad \mathtt{e}' \Downarrow v, G}{\mathtt{e}_1@\mathtt{e}_2 \Downarrow v, G'; G}$$

3.10 Description of the algorithm

We finally give the description of the entire algorithm for detecting size-change termination. There are two steps: the generation of a safe set of size-change graphs for P and the decision of the SCT property.

1. Construct an over-approximation $\mathcal{G}$ that contains (at least) all size-change graphs that would be built during an exact evaluation of the λ-calculus expression:

   $$\mathcal{G} = \{\ G_j \mid j > 0 \wedge \exists \mathtt{e}_i, G_i (0 \leq i \leq j) : \\ \mathtt{P} = \mathtt{e}_0 \wedge (\mathtt{e}_0 \rightarrow \mathtt{e}_1, G_1) \wedge \ldots \wedge (\mathtt{e}_{j-1} \rightarrow \mathtt{e}_j, G_j)\ \}$$

   $\mathcal{G}$ can be computed by applying exhaustively the rules given in definition 3.9.2, starting with expression P until no new graphs or subexpressions are obtained. This process ends since P contains a finite number of subexpressions and possible size-change graphs.

   $\mathcal{G}$ is safe for for P (see theorem 3 in [5]).

2. Check whether the program satisfies the $\mathcal{G}$-size-change condition using the graph-based algorithm explained in section 2.5.6 and based on theorem 2.5.2:

   (a) build the set $\mathcal{S}$ containing the transitive closure of $\mathcal{G}$.

   (b) check that for all $G \in \mathcal{S}$ such that $G; G = G$, G has at least an arc of form $x \overset{\downarrow}{\rightarrow} x$. If it is the case then P is size-change terminating.

3.11 Improvement: a more accurate approximation

The over-approximation achieved by rules (VarAG) of definition 3.9.2 in the approximation semantics given in the previous section has a cost: the rule (VarAG) produces evaluation judgement forms of type $\mathtt{x} \Downarrow \mathtt{v}_2$ since the real scope of x is not taken into account.

Because the rules do not make use of the environment, at the next iteration of the exhaustive research, this judgement form will be reused as a premise to the rule (CallAG) or (ApplyAG) even if the context is different. As a result, new uninteresting judgement forms will be generated (this has been observed during experiments).

3.11.1 Variable renaming

One way to compensate for the absence of environment is to not reuse the same variable name in two different contexts in the λ-calculus program expression. This is what Neil D. Jones did in all of the examples of [5]. The drawback is that the code becomes less readable. For instance the variable s and z used in church numerals have to be renamed for each instance of a church numeral as we can see on the following code found in [5]:

Example 3.11.1

```
[λn.λx.  n                                          -- n --
         @  [λr.λa. 11: (r@ 13: (r@a))]             -- g --
         @  [λ k.λ s.λ z.(s@((k@s)@z))]             - succ-
         @  x ]                                     -- x --
@           [λs2.λz2.  (s2@(s2@(s2@z2))) ]          -- 3 --
@           [λs1.λz1.  (s1@(s1@(s1@(s1@z1))))]      -- 4 --
```

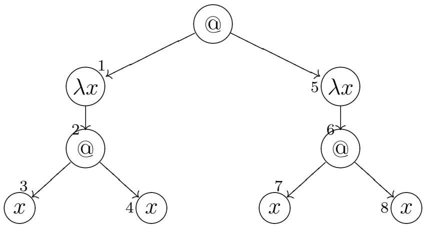
This is not an elegant way to solve the problem since it has an impact on the way the programmer has to write the code.

3.11.2 Another approach

I propose here another method which I have implemented. The effect of this method is the same as variable renaming : the approximation of the call and evaluation semantics is more accurate. This means that starting with identical input codes, the new method (like the variable renaming one) will detect size-change termination more often than the normal method with no renaming. Moreover having a more accurate approximation speeds-up the exhaustive application of approximation rules, since fewer judgment forms are generated.

The important change in this method is the way two subexpressions of `P` are distinguished. All the evaluation and call rules given until now were based on a structural comparison of the subexpression. This means that two subexpressions which are structurally equivalent are considered to be equal. I propose now to distinguish subexpressions according to their associated node numbers in the abstract tree.

Example 3.11.2
Consider the expression `P` $= (\lambda x.xx)(\lambda x.xx)$. Its abstract tree is:



By structural comparison, subexpressions 3 and 5 are equal. In the new method, subexpression 3 and 5 are considered to be different.

We will use the following notations:

- $node(i)$ represents the node number i in the abstract syntax tree,
- $\langle @, n_1, n_2\rangle$ represents a node labeled @ with two daughter nodes numbered n_1 and n_2,
- $\langle \lambda\mathtt{x}, n\rangle$ denotes a node labeled by $\lambda\mathtt{x}$ with a single daughter node with number n,
- $\langle\mathtt{x}\rangle$ represents a leaf of the tree (labeled with a variable name).

The set of program point $\mathcal{P}$ is now defined as the set of node numbers in the abstract tree of P (instead of the set of subexpressions of P):

$$\mathcal{P} = nodes(\mathtt{P})$$

Applying this definition to example 3.11.2 we have

$$\mathcal{P} = \{0..8\} \qquad \text{instead of} \qquad \mathcal{P} = \{\mathtt{x}, \mathtt{x@x}, \lambda\mathtt{x.x@x}, \mathtt{P}\}$$

The rules for the evaluation and call semantics are now interpreted relatively to this new definition. For instance, if we apply rule (OperatorS) to the subexpression number 2 in the example we obtain the judgement form $2 \underset{r}{\rightarrow} 3$. Where 2 represents expression x@x and 3 represents expression x. This judgement form differs from $6 \underset{r}{\rightarrow} 7$ even if expressions 2 and 3 are equivalent to expressions 6 and 7.

The rule (OperatorS) can be rewritten as follow:

$$(OperatorS)\frac{}{i \underset{r}{\rightarrow} i_1} \quad (node(i) = \langle @, i_1, i_2\rangle)$$

Consequently the number of possible judgement forms for a given program P increases.

The rules of definitions 3.2.1, 3.4.1, 3.5.2, 3.8.1, 3.9.1 and 3.9.2 can be all reinterpreted using the new notion of program points as we did for rule (OperatorS).

Rule (VarA)

Now, let us focus on the rule (VarA) of definition 3.9.1 in order to solve the over-approximation problem:

$$(VarA)\frac{\mathtt{e}_1\mathtt{@e}_2 \in subexp(\mathtt{P}) \quad \mathtt{e}_1 \Downarrow \lambda\mathtt{x.e}_0 \quad \mathtt{e}_2 \Downarrow v_2}{\mathtt{x} \Downarrow v_2}$$

We first rewrite the rule according to the new definition of $\mathcal{P}$:

$$(VarA)\frac{i_{\mathtt{e}_1\mathtt{@e}_2} \in nodes(\mathtt{P}) \quad i_1 \Downarrow i_{\lambda\mathtt{x.e}_0} \quad i_2 \Downarrow v_2}{i_\mathtt{x} \Downarrow v_2}$$

with the following side-conditions:

1. $node(i_{\mathtt{e}_1 @ \mathtt{e}_2}) = \langle @, i_1, i_2 \rangle$

2. $node(i_{\lambda \mathtt{x}.\mathtt{e}_0}) = \langle \lambda \mathtt{x}, i_0 \rangle$

3. $node(i_{\mathtt{x}}) = \langle \mathtt{x} \rangle$

The over-approximation is due to the fact that we ignore the scope of variable x. The solution that I propose consists in adding a side-condition to the rule (VarA) to limit the scope of the variable `x`. The side-condition is expressed as follow:

4. The node $i_{\mathtt{x}}$ belongs to the subtree rooted at node $i_{\mathtt{e}_0}$ and represents a free occurrence of variable `x`.

The rules (VarAG) of 3.9.2 can be modified in exactly the same manner. These modified rules are used in place of the original ones during the exhaustive search of judgement forms.

Remark 3.11.1. We now realize that the new definition of program points gives another advantage: suppose that the variable `x` bound in the abstraction $\lambda \mathtt{x}.\mathtt{e}_0$ does not occur as a subexpression of $\mathtt{e}_0$. Then the rule (VarAG) will not be applied and no useless judgement forms will be generated! This was not the case with the original definition of rule (VarAG) in [6].

Results

The effect of this change is the same as variable renaming.

It was easy to observe this experimentally: consider the original method noted M_{org}, the new one M_{new}, a program code `P` where variable identifiers are used several times in different context and the corresponding code $\mathtt{P}_{ren}$ where the variables have been renamed manually.

Then I was able to check that judgement forms obtained by running M_{org} on $\mathtt{P}_{ren}$ match exactly with the judgment forms obtained by running M_{new} on `P`.

Moreover, running M_{org} on `P` produces extra judgement forms which prevent the detection of size-change termination on all the examples given in [5].

3.12 Implementation

The termination analysis algorithm for the λ-calculus has been implemented in OCaml ([4]). We give here a presentation of the implementation details including a description of the data structures.

The object oriented features of OCaml have been used in order to develop reusable code. For instance the functions used for the size-change termination decision procedure are contained in a class defined in the separate module `Sct` which is then derived into other classes for the different flavors of termination analysis.

The implementation consist of:

- a common set of tools including the SCT decision procedure: 914 lines of commented code (34 kilobytes).
- modules specific to the λ-calculus case including a parser and lexer for the language: 641 lines of commented code (21 kilobytes).
- modules specific to the ML language including a parser and lexer (see next chapter): 1182 lines of commented code (36 kilobytes).

3.12.1 Data structures

The program first parses an input file containing the description of the λ-calculus expression and produces the correspdoning abstract syntax tree. The following OCaml type describes the data structure used to store this tree:

```
type ident = string

type lambda_expr =
    Var of ident
  | Abstr of ident * lambda_expr
  | Appl of lambda_expr * lambda_expr
```

λ-expressions

The only λ-expression involved in the termination analysis algorithm are the subexpressions of the program expression P. This suggested me to use another data structure for the program expression. the abstract syntax tree of the program is converted into an array. Each element of this array is a node or a leaf of the syntax tree. Nodes are abstractions and applications, leaves are variables. This way, any subexpression of the program can be just represented by a number: the index of the subexpression in the program expression array.

The type lmb_node is the type of element in the program expression array.

Program subexpressions are represented by the index number of the corresponding node in the expression array (type sub_expr):

```
type sub_expr = int

type lmd_node =
    VarN of sub_expr
  | AbstrN of sub_expr * sub_expr
  | ApplN of sub_expr * sub_expr
```

Size-change graphs

The following Caml code gives the types used for the element of size-change graphs basis (gb_element) and the arcs of size-change graphs (scg_arc):

```
type gb_element = Variable of sub_expr | Bullet type scg_arclabel =
ArrowDown | ArrowEqual type scg_arc = gb_element * scg_arclabel *
gb_element
```

An element of a graph basis is either a variable name or the special vertex • represented by the value Bullet in the type gb_element.

An arc in the graph is composed of a label and two graph basis elements.

Finally, a size-change graph G is represented by the following type:

```
type scgraph = sub_expr * sub_expr * scg_arc list
```

Instead of storing the list of graph basis elements in the structure of the size-change graphs, we just record the number of two subexpressions in the program (the first two sub_expr components). The graph basis of the graph are determined by the set of free variables of these two subexpressions. This trick speeds-up the generation of size-change graphs.

The third component is the list of arcs in the graph.

Judgement forms

Control points are the abstract syntax tree nodes:

```
type lmd_cpt = sub_expr
```

The following enumerated type gives the different flavor of judgement forms:

```
type lmd_jftype = Operator | Operand | FuncApp | Evaluation;;
```

Each judgment form has a component containing information generated during application of the rules. In the λ-calculus case, this is the set of arcs of a size-change graph:

```
type lmd_jfgen = scg_arc list
```

Finally the type used for judgment forms:

```
type lmd_jf = lmd_jftype * lmd_jfgen
```

3.12.2 Parser

I have developed a parser using `ocamllex` and `ocamlyacc`. This feature makes the program easier to test on different λ-calculus expressions.

The syntax recognized by the parser is based on the formal syntax given in definition 3.1.1. Expression can be typed in a convenient manner: optional brackets can be avoided using the standard convention used in λ-calculus, the application operator is optional and commentary can be added after the sequence of characters `//` or between `(*` and `*)`.

The following program is an example of a λ-expression correctly parsed by the program:

Example 3.12.1

```
(lambda n.lambda x.n                              // n
  ( lambda r.lambda a.r (r  a))                   // g
  ( lambda k.lambda s.lambda z.s ((k s) z))       // succ
  x )                                             // x
(lambda s2.lambda z2.s2 (s2 (s2 z2)))             // 3 (lambda s1.lambda
z1.s1 (s1 (s1 (s1 z1))))    // 4
```

3.12.3 LaTex output

Running the analysis with the command line argument `-latex` will produce a latex file which, once processed with LaTex, generates a graphical representation of the syntax tree of the λ-expression. This was particularly helpful during the debugging phase of the program. See figure 3.13.2 for an example.

3.13 Results

The program has been tested on the examples given in [5]. The results obtained with my implementation are identical to those given in [5].

The program is started at the shell prompt using the command `sct file.lmd` where `file.lmd` is the file containing the lambda expression to analyze.

If the λ-expression is size-change terminating then the program outputs the list of all loops with the corresponding size-change graphs certifying that they are descending.

If the λ-expression is not size-change terminating then the program outputs the list of all loops which are not descending.

Note that only loop graphs verifying the equation $G;G = G$ are printed out (contrary to the outputs presented in [5]).

Example 3.13.1
This is a possible output:

```
Program is size change terminating! All the loops are descending:
24->*24,[s>s p=p][22, 8, 12, 8, 12, 8]
42->*42,[s3>s3][8, 22, 8, 12, 8, 22, 8, 38, 40]
```

The first line tells us that the program is size-change terminating. Then each line of the output corresponds to a different loop in the program.

Consider the first line:

- `24->*24` means that the loop occurs at node 24 in the abstract syntax tree of the λ-expression.
- `[s>s, u=t]` is the list of arcs of the size-change graph describing the loop. `s>s` represents the arc $s \overset{\downarrow}{\rightarrow} s$ and `u=t` represents the arc $u \overset{=}{\rightarrow} t$.
- `[22, 8, 12, 8, 12, 8]` is a list of subexpressions numbers. This is the call path of the transitive call from subexpression 24 to itself.

  One of the steps in the algorithm consists in computing the composition closure of the set of size-change graphs. During that phase, when two graphs $G_1 : e \rightarrow f$ and $G_2 : f \rightarrow g$ are composed, a new graph $G_3 : e \rightarrow g$ is created. The number of the subexpression f is then recorded in this list for the graph G_3.

Remark 3.13.1. Note that the lists of program points stored in the third component can differ from one implementation to another depending on the order in which graphs are browsed during the closure computation.

For instance, my first implementation of the algorithm used Caml `List` data structures to store the set of judgment forms. This implementation gave exactly the same control points lists as the one obtained in [5]. The output printed in this report were obtained with a more recent implementation which stores the judgement forms in a matrix for fast access. This produces different control point lists. However this implementation is dramatically faster than the original one and permits to analyze long programs that were impossible to analyze before (see `min.chml` example in next chapter).

3.13.1 Omega

The lambda expression $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ is written as follow:

omega.lmd

```
(lambda x . x x )
  (lambda x . x x )
```

```
$ ./sct omega.lmd
Program is not size change terminating! The critical (ie. not descending) loops are:
6->*6,[x=x][]
```

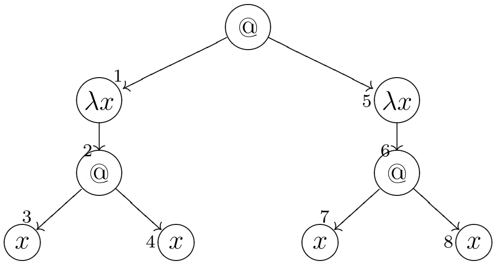


Table 3.1: Syntax tree generated with the command `sct -latex omega.lmd`.

3.13.2 Simple program from [5]

simple.lmd

```
(lambda s . lambda z . s (s z) )                // two
(lambda m . lambda s . lambda z . (m s) (s z))  // succ
(lambda s . lambda z . z)                       // zero
(lambda x . x)                                  // id1
(lambda x . x)                                  // id2
```

```
$ ./sct simple.lmd
Program is size change terminating! All the loops are descending:
14->*14,[m>m, s=s, z=z][]
```

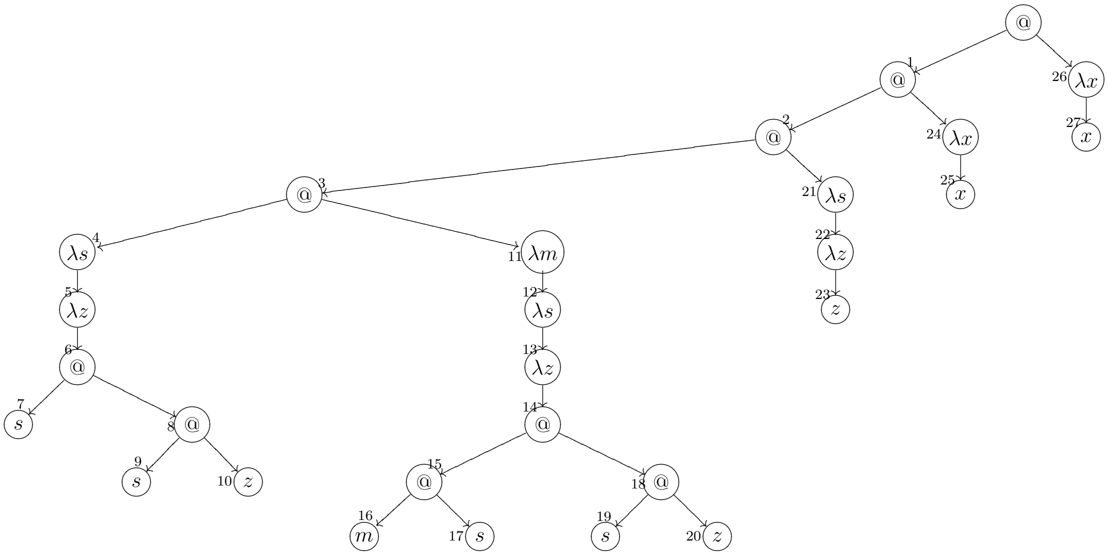


Table 3.2: Syntax tree generated with the command `sct -latex simple.lmd`.

3.13.3 Church numerals

churchnum.lmd

```
 (lambda n.lambda x.n                         // n
   ( lambda r.lambda a.r (r  a))              // g
   ( lambda k.lambda s.lambda z.s ((k s) z))   // succ
   x )                                         // x
 (lambda s.lambda z.s (s (s z)))               // 3
 (lambda s.lambda z.s (s (s (s z))))           // 4
```

```
$ ./sct churchnum.lmd
Program is size change terminating! All the loops are descending:
10->*10,[r>r, a=a][12]
10->*10,[r>r][]
12->*12,[r>r][10, 10]
12->*12,[r>r, a=a][10]
```

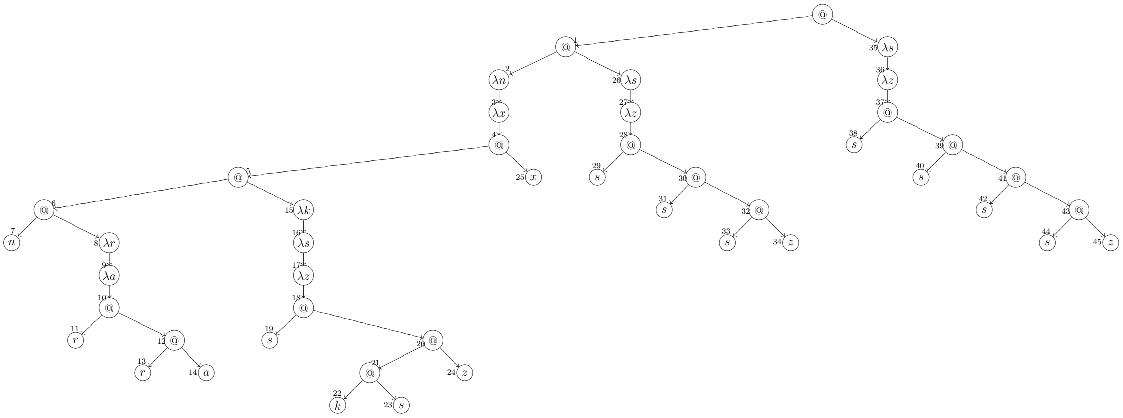


Table 3.3: Syntax tree generated with the command `sct -latex churchnum.lmd`.

3.13.4 Ackerman's function

ackerman.lmd

```
(lambda m. m
        // b
          ( lambda g. lambda n. n g (g  (lambda s.lambda z. s z) ) )
        // succ
          (lambda k.lambda s.lambda z. s (k s z))
)
(lambda s.lambda z. s (s z))         // 2
(lambda s.lambda z. s (s( s(z))) )  // 3
```

```
$ ./sct ackerman.lmd
Program is size change terminating! All the loops are descending:
8->*8,[g>g][16]
12->*12,[g>g][8]
16->*16,[s>s][8]
22->*22,[k>k, s=s, z=z][24]
22->*22,[s>s][8]
24->*24,[k>k, s=s, z=z][22]
```

```
24->*24,[s>s][16, 8, 12, 8, 22]
38->*38,[s>s][8] 40->*40,[s>s][8, 38]
42->*42,[s>s][8, 38, 40]
```

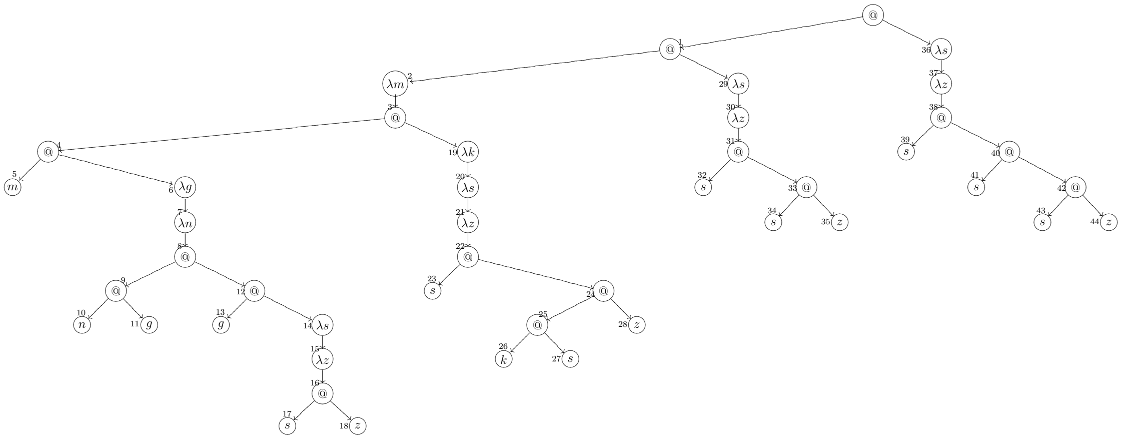


Table 3.4: Syntax tree generated with the command `sct -latex ackerman.lmd`.

3.13.5 Performance

Table 3.5 gives the times it takes to run the analysis on the different λ-expression examples using the natively compiled version of the Objective Caml program (these figures have been measured on a laptop computer equipped with a P3 2.4Ghz processor, 512Mb of RAM and running Windows XP).

λ-expression	Time
omega.lmd	0.00s
simple.lmd	0.00s
churchnum.lmd	0.03s
ackerman.lmd	0.04s

Table 3.5: Performance of λ-expression analysis

Chapter 4

Extension to core ML

In this chapter, we extend the size-change principle to the case of a more complex language. This language is a subset of the CoreML language based on the language defined in [7]. We use $\mathcal{L}_{ml}$ to refer to this language.

4.1 The language $\mathcal{L}_{ml}$

There are two ground types in the language, integers (`int`) and booleans (`bool`). Remember that the size-change principle requires us to work on well-founded data. We therefore restrict the ground type `int` to positive integers and we work on the well well-founded set $(\mathbb{N}, \leq)$. There are two operators which can be applied to numbers: predecessor (`prec`) and successor (`succ`).

The language supports `if-then-else` branching structure and equality test (`e1 = e2`)

4.1.1 Grammar of $\mathcal{L}_{ml}$

The following grammar defines expressions of the language $\mathcal{L}_{ml}$:

e ::=	x,f	value identifiers (x,f $\in Var$)
	`true`	boolean constants
	`false`	
	`if e then e else e`	conditional
	`n`	integer constants (n $\in \mathbb{N}$)
	`e = e`	integer equality
	`succ e`	successor
	`pred e`	predecessor
	`fun (x:ty) -> e`	function abstraction
	`fun f=(x:ty) -> e`	recursively defined function
	`e e`	function application
	`let x = e in e`	local definition

Var is a countably infinite sets of variables.

This syntax is similar to the Caml syntax but there are some differences. For instance `fun f = (x:ty) -> e` corresponds to the following Caml code:

```
let rec f = (fun (x:ty) -> e) in f
```

Other constructs can be implemented by adding some syntaxic sugar to the parser. For instance the parser that I have implemented recognizes the following structures:

- multiple local definitions: `let e = ...and...in...`
- recursion: `let rec f = ...in ...`

The set of subexpresssions of a $\mathcal{L}_{ml}$ expression e is noted $subexpr(e)$ and is defined by induction on the structure of e in the usual way.

Definition 4.1.1 (Free variables in $\mathcal{L}_{ml}$)**.** *The set of all variables occuring freely in the expression e is noted $fv(e)$ and is defined by induction on the structure of e. In particular we have:*

$$\begin{aligned} fv(\mathtt{x}) &\triangleq \{\mathtt{x}\} \\ fv(\mathtt{fun\ (x:ty)\ ->\ e}) &\triangleq fv(\mathtt{e}) - \{\mathtt{x}\} \\ fv(\mathtt{fun\ f=(x:ty)\ ->\ e}) &\triangleq fv(\mathtt{e}) - \{\mathtt{x}\} \\ fv(\mathtt{let\ x\ =\ e1\ in\ e2\ }) &\triangleq fv(\mathtt{e1}) \cup (fv(\mathtt{e2}) - \{\mathtt{x}\}) \end{aligned}$$

If $fv(e) = \emptyset$, we say that e is a closed expression.

A $\mathcal{L}_{ml}$ program is a closed expression.

4.1.2 Type assignment

(We rely on definitions given in [7].) A type can be assigned to every expression in $\mathcal{L}_{ml}$. The set of $\mathcal{L}_{ml}$ types is given by the following grammar:

ty ::=	bool	booleans
	int	positive integers
	ty → ty	functions

Type assignment relation is of the form $\Gamma \vdash e : ty$ where

- the typing context Γ is a function from a finite set $dom(\Gamma)$ of variables to types
- e is an expression
- ty is a type

This relation is built inductively by the following rules ($\Gamma[x \mapsto ty]$ denotes the typing context mapping x to ty and acting like Γ otherwise) :

Value identifiers:
$$\frac{x \in dom(\Gamma) \quad \Gamma(x) = ty}{\Gamma \vdash x : ty}$$

Boolean constants:
$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash b : bool}$$
Integer constants:
$$\frac{n \in \mathbb{N}}{\Gamma \vdash n : int}$$

Conditional:
$$\frac{\Gamma \vdash e_1 : ty \quad \Gamma \vdash e_2 : ty \quad \Gamma \vdash e : bool}{\Gamma \vdash (\texttt{if e then e}_1\texttt{ else e}_2) : \texttt{ty}}$$

Integer equality:
$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash (\texttt{e}_1 = \texttt{e}_2) : \texttt{bool}}$$

Function abstraction:
$$\frac{\Gamma[x \rightarrow ty_1] \vdash e : ty_2 \quad x \notin (\Gamma)}{\Gamma \vdash (\texttt{fun (x : ty}_1\texttt{)-> e}) : \texttt{ty}_1 \rightarrow \texttt{ty}_2}$$

Recursively defined function:
$$\frac{\Gamma[f \mapsto ty_1 \rightarrow ty_2][x \rightarrow ty_1] \vdash e : ty_2 \quad f, x \notin dom(\Gamma) \quad f \neq x}{\Gamma \vdash (\texttt{fun f } = \texttt{ (x : ty}_1\texttt{)-> e}) : \texttt{ty}_1 \rightarrow \texttt{ty}_2}$$

4.1.3 Canonical forms

We say that a $\mathcal{L}_{ml}$ expression is a canonical form if it is a constant (integer or boolean) or a function. The set of canonical forms (noted $Canon$) is given by the following grammar:

v ::=	true	
	false	
	n	any positive integer
	`fun (x:ty) -> e`	function
	`fun f=(x:ty) -> e`	recursive function

4.1.4 Semantics of $\mathcal{L}_{ml}$

The evaluation relation $e \Downarrow v$ expresses the fact that the closed expression e evaluates to the closed canonical form v. The notation $e \Downarrow$ means that $e \Downarrow v$ for some $v \in Value$ and $e \not\Downarrow$ is an abbreviation for $\neg(e \Downarrow)$. The rules of table 4.1 give the inductive definition of the evaluation relation.

The evaluation of `pred 0` causes an error. Any $\mathcal{L}_{ml}$ expression which involves the evaluation of `pred 0` during its own evaluation will also cause an error. We use $\texttt{e} \oslash$ to denote that an error will occur during the evaluation of the expression `e`. The error semantics is given in the table 4.2. It is straightforward to check the following lemma:

Lemma 4.1.1 (The predicates $\Downarrow$ and $\oslash$ are disjoint)**.**

$$\texttt{e} \oslash \implies \texttt{e} \not\Downarrow$$

A program $\mathtt{P}$ terminates, noted $\mathtt{P}\lightning$, if it can be evaluated or if an error occurs while trying to evaluate it:

$$\mathtt{P}\lightning \quad \triangleq \quad \mathtt{P}\Downarrow \quad \vee \quad \mathtt{P}\oslash$$

Canonical forms: $$\frac{}{\mathtt{v} \Downarrow \mathtt{v}}(\mathtt{v} \text{ in canonical form})$$

Conditional: $$\frac{\mathtt{e} \Downarrow \mathtt{true} \qquad \mathtt{e}_1 \Downarrow \mathtt{v}}{\mathtt{if\ e\ then\ e}_1\ \mathtt{else\ e}_2 \Downarrow \mathtt{v}} \qquad \frac{\mathtt{e} \Downarrow \mathtt{false} \qquad \mathtt{e}_2 \Downarrow \mathtt{v}}{\mathtt{if\ e\ then\ e}_1\ \mathtt{else\ e}_2 \Downarrow \mathtt{v}}$$

Integer equality: $$\frac{\mathtt{e}_1 \Downarrow \mathtt{n} \qquad \mathtt{e}_2 \Downarrow \mathtt{n}}{\mathtt{e}_1 = \mathtt{e}_2 \Downarrow \mathtt{true}} \qquad \frac{\mathtt{e}_1 \Downarrow \mathtt{n} \qquad \mathtt{e}_2 \Downarrow \mathtt{m} \qquad \mathtt{n} \neq \mathtt{m}}{\mathtt{e}_1 = \mathtt{e}_2 \Downarrow \mathtt{false}}$$

Operator: $$\frac{\mathtt{e} \Downarrow \mathtt{n}}{\mathtt{succ\ e} \Downarrow \mathtt{n} + 1} \qquad \frac{\mathtt{e} \Downarrow \mathtt{n} \qquad \mathtt{n} > 0}{\mathtt{pred\ e} \Downarrow \mathtt{n} - 1}$$

Function application: $$\frac{\mathtt{e}_1 \Downarrow \mathtt{fun(x : ty)\text{->}e}_0 \qquad \mathtt{e}_2 \Downarrow \mathtt{v}_2 \qquad \mathtt{e}_0[\mathtt{v2/x}] \Downarrow \mathtt{v}}{\mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{v}}$$

$$\frac{\mathtt{e}_1 \Downarrow \mathtt{v}_1 \equiv \mathtt{fun\ f} = \mathtt{(x : ty)\text{->}e}_0 \qquad \mathtt{e}_2 \Downarrow \mathtt{v}_2 \qquad \mathtt{e}_0[\mathtt{v}_2/\mathtt{x}, \mathtt{v}_1/\mathtt{f}] \Downarrow \mathtt{v}}{\mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{v}}$$

Local definition: $$\frac{\mathtt{e}_1 \Downarrow \mathtt{v}_1 \qquad \mathtt{e}_2[\mathtt{v}_1/\mathtt{x}] \Downarrow \mathtt{v}}{\mathtt{let\ x}\ =\ \mathtt{e}_1\ \mathtt{in\ e}_2 \Downarrow \mathtt{v}}$$

Table 4.1: $\mathcal{L}_{ml}$ evaluation relation

Lemma 4.1.2 (Determinism). *The relation $\Downarrow$ is deterministic:*

$$e \Downarrow v \wedge e \Downarrow v_2 \implies v = v_2$$

Operators:

$$(\text{ErrOp1})\frac{\texttt{e} \Downarrow \texttt{0}}{\texttt{pred e} \oslash} \qquad (\text{ErrOp2})\frac{\texttt{e} \oslash}{\texttt{pred e} \oslash} \qquad (\text{ErrOp3})\frac{\texttt{e} \oslash}{\texttt{succ e} \oslash}$$

Conditional:

$$(\text{ErrIf1})\frac{\texttt{e} \oslash}{\texttt{if e then e}_1\texttt{ else e}_2 \oslash}$$

$$(\text{ErrIf2})\frac{\texttt{e} \Downarrow \texttt{true} \qquad \texttt{e}_1 \oslash}{\texttt{if e then e}_1\texttt{ else e}_2 \oslash} \qquad (\text{ErrIf3})\frac{\texttt{e} \Downarrow \texttt{false} \qquad \texttt{e}_2 \oslash}{\texttt{if e then e}_1\texttt{ else e}_2 \oslash}$$

Integer equality:

$$(\text{ErrEq1})\frac{\texttt{e}_1 \oslash}{\texttt{e}_1 = \texttt{e}_2 \oslash} \qquad (\text{ErrEq2})\frac{\texttt{e}_2 \oslash}{\texttt{e}_1 = \texttt{e}_2 \oslash}$$

Function application:

$$(\text{ErrApp1})\ \frac{\texttt{e}_1 \oslash}{\texttt{e}_1\ \texttt{e}_2 \oslash} \qquad (\text{ErrApp2})\ \frac{\texttt{e}_1 \Downarrow \begin{cases} \texttt{fun f} = (\texttt{x} : \texttt{ty})\texttt{->e}_0 \\ \texttt{fun } (\texttt{x} : \texttt{ty})\texttt{->e}_0 \end{cases} \qquad \texttt{e}_2 \oslash}{\texttt{e}_1\ \texttt{e}_2 \oslash}$$

$$(\text{ErrApp3})\ \frac{\texttt{e}_1 \Downarrow \texttt{v}_1 \equiv \begin{cases} \texttt{fun f} = (\texttt{x} : \texttt{ty})\texttt{->e}_0 \\ \texttt{fun } (\texttt{x} : \texttt{ty})\texttt{->e}_0 \end{cases} \qquad \texttt{e}_2 \Downarrow \texttt{v}_2 \qquad \texttt{e}_0[\texttt{v}_2/\texttt{x}, \texttt{v}_1/\texttt{f}] \oslash}{\texttt{e}_1\ \texttt{e}_2 \oslash}$$

Local definition:

$$(\text{ErrLocDef1})\frac{\texttt{e}_1 \oslash}{\texttt{let x} = \texttt{e}_1\texttt{ in e}_2 \oslash} \qquad (\text{ErrLocDef2})\frac{\texttt{e}_1 \Downarrow \texttt{v}_1 \qquad \texttt{e}_2[\texttt{v}_1/\texttt{x}] \oslash}{\texttt{let x} = \texttt{e}_1\texttt{ in e}_2 \oslash}$$

Table 4.2: $\mathcal{L}_{ml}$ error semantics

4.2 First approach: Conversion from ML to λ-calculus

We would like to decide $\mathcal{G}$-size-change termination property for a given $\mathcal{L}_{ml}$ program.

The first approach consists in converting the $\mathcal{L}_{ml}$ program into a lambda calculus expression. Integers are expanded into church numerals, `if-then-else` structures ate implemented using appropriated λ-expressions and recursion is implemented using the Y combinator.

Provided that the conversion transposes isomorphically the termination property, we can apply the size-change principle explained in chapter 3 on the converted expression.

The conversion is done by syntactical analysis of the $\mathcal{L}_{ml}$ program expression:

- Function definition:

$$\lceil \mathtt{fun\ x_1\ x_2\ \ldots x_n\text{->}\ e} \rceil = \lambda x_1 x_2 \ldots x_n.\lceil e \rceil$$

- Application:

$$\lceil \mathtt{e_1\ e_2} \rceil = \lceil \mathtt{e_1} \rceil \lceil \mathtt{e_2} \rceil$$

- Numbers are implemented using Church numerals:

$$\lceil \mathtt{n} \rceil = \lambda sz.\underbrace{s(s(\ldots(s}_{n \text{ times}}\, z))\ldots)$$

  In particular we have $\lceil \mathtt{0} \rceil = \lambda sz.z$.

- The boolean value true and false are defined by:

$$\lceil \mathtt{true} \rceil = \mathbf{true} = \lambda xy.x$$

$$\lceil \mathtt{false} \rceil = \mathbf{false} = \lambda xy.y$$

- The `if-then-else` structure is implemented by using the constants true and false:

$$\lceil \mathtt{if\ e\ then\ e_1\ else\ e_2} \rceil = \ \mathtt{e\ e1\ e2} = \begin{cases} \mathtt{e_1} \text{ if } \mathtt{e} = \mathbf{true} \\ \mathtt{e_2} \text{ elsewhere.} \end{cases}$$

- The successor and predecessor operators are defined as follow:

$$\lceil \mathtt{succ} \rceil = \mathbf{succ} = \lambda ksz.s(ksz)$$

$$\lceil \mathtt{prec} \rceil = \mathbf{prec} = \lambda n.n(\lambda z.z\ \mathbf{i}(\mathbf{succ}\ z))(\lambda a\ b.\lceil 0 \rceil)$$

  where $\mathbf{i} = \lambda x.x$.

- Zero equality test:

  $$\lceil \texttt{n = 0} \rceil = \textbf{iszero}\ \lceil \texttt{n} \rceil = \lceil \texttt{n} \rceil\,(\lambda \mathrm{x}.\textbf{false})\textbf{true}$$

  where `n` is a number.

- Local definition:

  $$\lceil \texttt{let}\ \mathtt{x_1}\ =\ \mathtt{e_{x_1}}\ \texttt{and}\ \ldots\ \mathtt{x_n}\ =\ \mathtt{e_{x_n}}\ \texttt{in e} \rceil = (\lambda x_1 \ldots x_n.\, \lceil \texttt{e} \rceil)\, \lceil \mathtt{e_{x_1}} \rceil \ldots \lceil \mathtt{e_{x_n}} \rceil$$

  Remark 4.2.1. another approach consists in substituting in the body expression of the `let` structure all the occurences of the definition names by their corresponding value translated into lambda calculs:

  $$\lceil \texttt{let}\ \mathtt{x_1}\ =\ \mathtt{e_{x_1}}\ \texttt{and}\ \ldots\ \mathtt{x_n}\ =\ \mathtt{e_{x_n}}\ \texttt{in e} \rceil = \lceil \texttt{e} \rceil\, [\lceil \mathtt{e_{x_1}} \rceil / x_1, \ldots, \lceil \mathtt{e_{x_n}} \rceil / x_n]$$

  where $e[f/x]$ denotes the expression obtained after replacing every free occurence of x in expression e by the expression f.

  Unfortunately, this transformation does not preserve the termination property. Indeed, suppose that Ω is a valid well-typed $\mathcal{L}_{ml}$ expression which does not terminate, then `true[Ω/x] = true` terminates whereas `let x = Ω in true` does not.

- Recursion is implemented using the Y combinator defined as follow:

  $$\mathbf{Y} = \lambda p.(\lambda q.p(\lambda s.q(qs)))\ (\lambda t.p(\lambda u.t(tu)))$$

  $$\lceil \texttt{let rec f}\ =\ \mathtt{e_f}\ \texttt{in e} \rceil = (\lambda f.\, \lceil \texttt{e} \rceil)(\mathbf{Y}\ (\lambda f.\, \lceil \mathtt{e_f} \rceil))$$

4.2.1 Implementation

The implementation is straightforward: it consists in converting the $\mathcal{L}_{ml}$ expression into a λ-calculus expression by following the rules explained in the previous section.

Because of the improvement explained in section 3.11, variable renaming is unnecessary. This makes the expansion of church numerals much easier during the conversion process since all church numerals can now use the same variable names s and z.

The special parameter `-ml` indicates to the program that we want to analyze a $\mathcal{L}_{ml}$ expression, `-conv` indicates that we want it to be converted into a λ-calculus expression.

4.2.2 Results

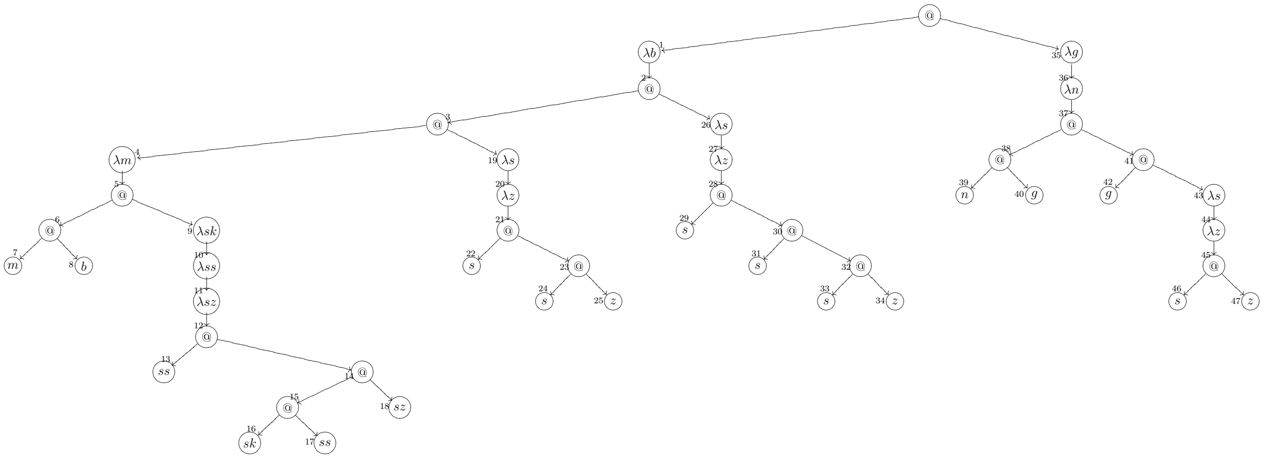
This method has first been tested on the same programs as in the λ-calculus case (note that using the syntax of $\mathcal{L}_{ml}$ these sample programs can be rewritten in a clearer way).

The analysis is very fast and the results given are similar to those obtained in the λ-calculus case:

- Ackerman's function

```
ackerman.chml
let b g n = n g (g 1)
in (fun m -> m b succ) 2 3 ;;
```

```
$ ./sct.opt -ml -conv ackerman.chml
Program is size change terminating! All the loops are descending:
12->*12,[ss>ss][37]
12->*12,[sk>sk, ss=ss, sz=sz][14]
14->*14,[ss>ss][12, 37, 12]
14->*14,[sk>sk, ss=ss, sz=sz][12]
28->*28,[s>s][37]
30->*30,[s>s][37, 12, 14, 12, 37, 28]
32->*32,[s>s][37, 12, 14, 12, 37, 28, 30]
37->*37,[g>g][12]
41->*41,[g>g][37]
45->*45,[s>s][37]
```

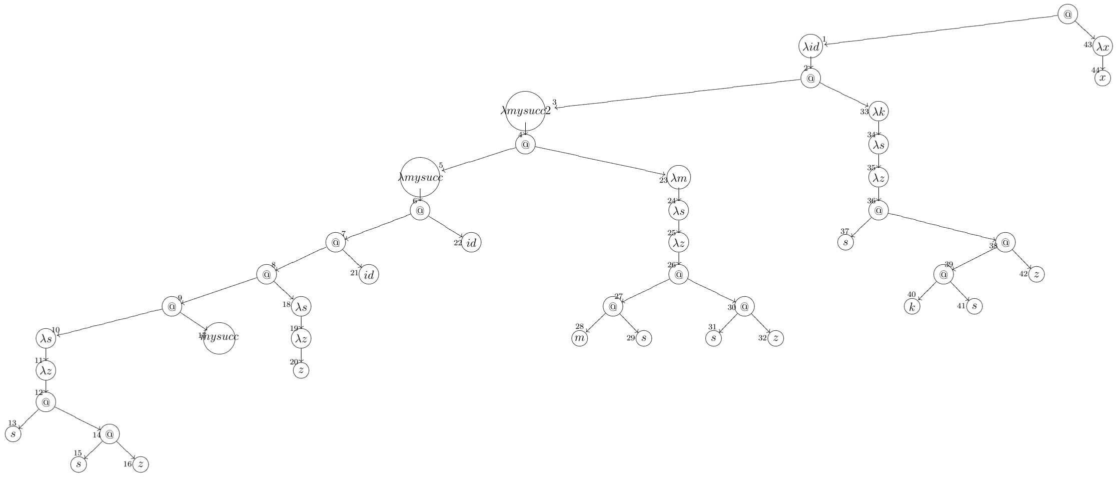


- Simple:

```
simple.chml
let mysucc m s z = (m s) (s z)
and mysucc2 k s z = s (k s z)
and id x = x
in

2 mysucc 0 id id
;;
```

```
$ ./sct.opt -ml -conv simple.chml
Program is size change terminating! All the loops are descending:
26->*26,[m>m, s=s, z=z][]
```



- Church numerals:

churchnum.chml

```
let g r a = r  (r  a)
in
 (fun n -> fun x ->  n g succ x ) 3 4
;;
```

```
$ ./sct.opt -ml -conv churchnum.chml
Program is size change terminating! All the loops are descending:
44->*44,[r>r, a=a][46]
44->*44,[r>r][]
46->*46,[r>r][44, 44]
46->*46,[r>r, a=a][44]
```

4.2.3 Performance

Table 4.3 gives the times it takes to run the analysis on the different $\mathcal{L}_{ml}$ expression examples (these figures have been measured on a laptop computer equipped with a P3 2.4Ghz processor, 512Mb of RAM and running Windows XP).

λ-expression	Time
omega.cml	0s
simple.chml	0.02s
churchnum.chml	0.02s
ackerman.chml	0.05s
min.chml	3.444s

Table 4.3: Performance of $\mathcal{L}_{ml}$ expressions analysis after conversion to λ-calculus

4.2.4 Limit of the approach

The following example has also been tested. It implements a recursively defined function for computing the minimum of two numbers.

min.chml

```
let rec min x y =
  if x = 0 then  0
  else
    if y = 0 then 0
    else
      succ (min (pred x) (pred y))
in
  min 2 5
;;
```

```
$ ./sct.opt -ml -conv min.chml
Program is not size change terminating! The critical (ie. not descending) loops are:
49->*49,[][67, 85, 96]
67->*67,[][85, 96, 49]
85->*85,[][96, 49, 67]
96->*96,[][49, 67, 85]
105->*105,[][]
106->*106,[][105, 105]
114->*114,[sz=sz][116]
114->*114,[][105]
116->*116,[sz=sz][114, 114]
116->*116,[][105, 114]
117->*117,[][105, 114, 116]
134->*134,[][]
135->*135,[][134, 134]
143->*143,[sz=sz][145]
143->*143,[][134]
145->*145,[sz=sz][143]
145->*145,[][134, 143]
146->*146,[][134, 143, 145]
```

Size-change termination is not detected by the algorithm! This particular example shows that our first approach is inefficient. This is because Church numerals are not adapted to the notion of size defined in definition 3.6.3. For instance the fact that

$$\underbrace{\lambda sz.s\ z}_{\lceil 1 \rceil} : [] \qquad \not\succ \qquad \underbrace{\lambda sz.z}_{\lceil 0 \rceil} : []$$

prevents us from detecting value decrease caused by the operator `pred`.

Moreover since numbers are expanded into church numerals, the size of the program expression becomes linear in the value of the integers used in the program expression. A program using the number x will have at least x nodes in its expression syntax tree. This causes bad performance.

These remarks motivate the new approach explained in the next section.

4.3 Second approach

The second approach consists in redefining from scratch the size-change principle for the language $\mathcal{L}_{ml}$. We follow the same steps as we did in the last chapter: define a well-founded notion of size, define call and evaluation semantics, extend these semantics with graph generation and finally exhibit the approximate semantics.

New difficulties arise in $\mathcal{L}_{ml}$: the presence of recursive definitions, the presence of ground types and errors.

The solution that I proposed combines two different size-change principles:

- The first one is based on an extension of the notion of size defined by Neil D. Jones for the higher-order case (λ-calculus) to the $\mathcal{L}_{ml}$ language. We will refer to it as SCP^+.
- The second one analyzes ground type values of type `int` and is based on the natural well-founded notion of size for positive integers: $(\mathbb{N}, \leq)$. We will refer to it as SCP^0.

These two principles will be applied in parallel. As a consequence, the rules for calls and evaluation semantics will be equipped with two graphs generation components: one for each of the two notions of size. Similarly, two different safe sets of size-change graphs will be generated.

The $\mathcal{L}_{ml}$ program will be terminating if it verifies the size-change termination condition for at least one of the two size-change principles SCP^+ and SCP^0.

We will assume that the $\mathcal{L}_{ml}$ program expression has already been type-checked.

4.3.1 Size-change graphs

Remember that size-change graphs are defined by the program control point set $\mathcal{P}$ and the graph-basis function gb.

In SCP^+ and SCP^0, program points and graph-basis are defined as follow:

- A program point is either a program subexpression or an integer or boolean value. Since the program point set $\mathcal{P}$ has to be finite for the size-change principle to work, we will represent integer values by the special symbol $?^{\texttt{int}}$ whose meaning is "any integer". Similarly we use the symbol $?^{\texttt{bool}}$ for undetermined boolean values:

$$\mathcal{P} = \mathit{subexp}(\texttt{P}) \cup \{?^{\texttt{int}}, ?^{\texttt{bool}}\}$$

- The graph-basis of a subexpression is the set of its free variables extended with the special $\bullet$ element. (definition 3.6.1). The graph-basis for the special element $?^{\texttt{int}}$ (and $?^{\texttt{bool}}$) is defined as the singleton $\{\bullet\}$. The use of these particular symbols is explained in section 4.3.5.

4.3.2 Environment based semantics

As we did for the λ-calculus, we define environments in order to describe the computation space.

The only difference is that $Value$ now contains ground type values as well as abstractions (i.e. all canonical form expressions).

The sets $State$, $Value$, Env are the smallest sets verifying the following equation:

$$\begin{array}{rcl} State &=& \{e : \rho \mid e \in Exp, \rho \in Env, fv(e) \subseteq dom(\rho)\} \\ Value &=& \{e : \rho \mid e : \rho \in State, e \text{ in } \mathit{canonical\ form}\} \\ Env &=& \{p : X \rightarrow Value \mid X \text{ finite set of variables}\} \end{array}$$

The evaluation and call semantics can now be expressed using environments. See tables 4.4 and 4.5 for a complete definition of the rules.

The judgement forms are $\mathtt{e} \Downarrow \mathtt{v}, G^0|G^+$ and $\mathtt{e} \rightarrow \mathtt{e}', G^0|G^+$ where $e, e', v \in Exp \times Env$ and G^0 and G^+ are the graph generation components (they can just be ignored for the moment).

The environment based semantics is equivalent to the standard one. The two definitions are related by the function $F : Exp \times Env \rightarrow Exp$ (see [8] and [5]) defined as:

$$F(\mathtt{e} : \rho) = \mathtt{e}[F(\rho(\mathtt{x}_1))/\mathtt{x}_1, ..., F(\rho(\mathtt{x}_k))/\mathtt{x}_k] \quad \text{where } \{\mathtt{x}_1, .., \mathtt{x}_k\} = dom(\rho) \cap fv(e)$$

It can be shown that $\mathtt{P} : [] \Downarrow \mathtt{v}$ (relatively to table 4.4) if and only if $\mathtt{P} \Downarrow F(v)$ (relatively to table 4.1).

We do not need to redefine an environment based error semantics, instead we adopt the notation $\mathtt{e} : \rho \oslash$ to mean $F(\mathtt{e} : \rho) \oslash$.

The notation for termination is $\mathtt{P}\lightning$ where the termination predicates is defined as follow:

$$\lightning = \Downarrow \cup \oslash$$

Non-termination is characterized by the presence of infinite call sequences:

Lemma 4.3.1 (NIS). *Let* P *be a program. Then:*

$$\neg(\mathtt{P}\lightning) \iff \mathtt{P} : [] = \mathtt{e}_0 : \rho_0 \rightarrow \mathtt{e}_1 : \rho_1 \rightarrow \mathtt{e}_2 : \rho_2 \rightarrow \ldots$$

The proof is in Appendix A.

4.3.3 Size and safe graphs

A different notion of size is used in SCP^0 and SCP^+:

- SCP^+

  Compared with the λ-calculus case, the syntax has changed but the function *subexp* and *support* can be defined similarly for the $\mathcal{L}_{ml}$ language. A counterpart of the size definition 3.6.3 can also be stated for $\mathcal{L}_{ml}$.

  Hence for the higher-order SCP^+, we use exactly the same notion of size as for the λ-calculus case.

- For SCP^0, we are only interested in the analysis of the size of expression of type `int` (*positive* integers). The notion of size used is based on the one underlying the well-founded set $(\mathbb{N}, \leq)$:

  Definition 4.3.1 (Size relation for SCP^0)**.** *We use the notation $\geq_{\texttt{int}}$ to denote the well-founded order on the integers and $\succeq_{\texttt{bool}}$ to denote any well-founded order on the boolean $\{true, false\}$.*

  Suppose that $s_1 = e_1 : \rho_1$ and $s_2 = e_2 : \rho_2$ then

$$s_1 \succeq_0 s_2 \qquad \triangleq \qquad \exists\ \Gamma\ s.t. \begin{array}{l} \left\{ \begin{array}{l} \Gamma \vdash e_1 : int, \qquad \Gamma \vdash e_2 : int \\ \quad and\ s_1 \Downarrow n_1 \wedge s_2 \Downarrow n_2 \implies n_1 \geq_{\texttt{int}} n_2 \end{array} \right. \\ or \\ \left\{ \begin{array}{ll} & \Gamma \vdash e_1 : bool, \qquad \Gamma \vdash e_2 : bool \\ and & s_1 \Downarrow b_1 \wedge s_2 \Downarrow b_2 \implies b_1 \succeq_{\texttt{bool}} b_2 \end{array} \right. \\ or \\ \Gamma \vdash e_1 : ty_1 \rightarrow ty_2 \quad and \quad \Gamma \vdash e_2 : ty_3 \rightarrow ty_4 \end{array}$$

  We write $s_1 \succ_0 s_2$ if $s_1 \succeq_0 s_2$ and $s_1 \neq s_2$.

  This size relation $\succeq_0 \subseteq State \times State$ is a well-founded order. Note that relatively to this order, all higher-order expressions are equal. This is because SCP^0 aims at analyzing ground type values only.

Safe size-change graphs for SCP^0 and SCP^+ are then defined relatively to their respective notion of size through the use of the valuation function as we did in definition 3.7.1 for the λ-calculus case. The definition of a safe set of size-change graphs remains the same.

The safety property is now defined and because of lemma 4.3.1, the two main theorems of the size-change principle (theorem 2.5.1 and 2.5.2) are valid for SCP^0 and SCP^+.

4.3.4 Semantics with graph generation

We know that the size-change principle can be used but we still need to provide a mechanism to generate a safe set of size-change graphs.

Again, we reused the technique developed for the λ-calculus case: we extend the semantics with a graph generation component in the judgement forms. In fact there are two graph components since we are dealing now with two size-change principles in parallel.

The rules of tables 4.4 and 4.5 give the evaluation and call semantics with graph generation. Each rule generates two size-change graphs: one for SCT^0 and one for SCT^+. The judgement forms are $\mathtt{e} \rightarrow \mathtt{e}', G|G^+$ or $\mathtt{e} \Downarrow \mathtt{v}, G|G^+$ where $s \in State$ and $v \in Value$. The special joker symbol _ means "any subexpression", "any graph" or "any environment" depending on the component in which it is used.

The sets of arcs G and G^+ are used as an abbreviation for $gb(e) \xrightarrow{G} gb(e')$ and $gb(e) \xrightarrow{G^+} gb(e')$ (the size change graphs themselves).

The tables 4.4 and 4.5 rely on the following definitions already given in the previous chapter:

$$\begin{array}{rcl}
id^{=}_{\mathtt{e}} & \triangleq & \{\bullet \xrightarrow{=} \bullet\} \cup \{\mathtt{x} \xrightarrow{=} \mathtt{x} \mid \mathtt{x} \in fv(\mathtt{e})\} \\
id^{\downarrow}_{\mathtt{e}} & \triangleq & \{\bullet \xrightarrow{\downarrow} \bullet\} \cup \{\mathtt{x} \xrightarrow{=} \mathtt{x} \mid \mathtt{x} \in fv(\mathtt{e})\} \\
G_1^{-\bullet} & \triangleq & \{\ \mathtt{y} \xrightarrow{r} \mathtt{z} \mid\ \mathtt{y} \xrightarrow{r} \mathtt{z} \in G_1\} \cup \{\ \bullet \xrightarrow{\downarrow} \mathtt{z} \mid\ \bullet \xrightarrow{r} \mathtt{z} \in G_1\} \\
G_2^{\bullet\mapsto\mathtt{x}} & \triangleq & \{\ \mathtt{y} \xrightarrow{r} \mathtt{x} \mid\ \mathtt{y} \xrightarrow{r} \bullet \in G_2\ \} \cup \{\ \bullet \xrightarrow{\downarrow} \mathtt{x} \mid\ \bullet \xrightarrow{r} \bullet \in G_2\ \}
\end{array}$$

Remark 4.3.1. We are only interested in generating size-change graphs for the evaluation and call judgment forms: there is no need to define graphs for the error semantics. Hence the tables 4.6 and 4.7 do not contain an environment based error semantics.

Remark 4.3.2. One may wonder why these tables do not contain the two following rules for conditional evaluation:

$$(\text{IfTrueG})\ \frac{\mathtt{e} : \rho \Downarrow \mathtt{true} : \_\ , \_\ |\_ \qquad \mathtt{e}_1 : \rho \Downarrow \mathtt{v}_1, G_1|G_1^+}{\mathtt{if\ e\ then\ e}_1\ \mathtt{else\ e}_2 : \rho \Downarrow \mathtt{v}_1, G_1|(id^{\downarrow}_{\mathtt{e}_1}; G_1^+)}$$

$$(\text{IfFalseG})\ \frac{\mathtt{e} : \rho \Downarrow \mathtt{false} : \_\ , \_\ |\_ \qquad \mathtt{e}_2 : \rho \Downarrow \mathtt{v}_2, G_2|G_2^+}{\mathtt{if\ e\ then\ e}_1\ \mathtt{else\ e}_2 : \rho \Downarrow \mathtt{v}_2, G_2|(id^{\downarrow}_{\mathtt{e}_2}; G_2^+)}$$

In fact, by looking carefully at the rules given in table 4.4 and 4.5, we realize that these two rules can be simulated by the rules (IfTrueCallG),(IfFalseCallG) and (ApplyG).

Remark 4.3.3. The definitions of graphs $LocalGr^0(x, G_1, e, G_2)$ and $LocalGr^+(x, G_1, e, G_2)$ generated by the rule (LocalG) look rather complicated. To understand them, let us rewrite the `let` structure of the $\mathcal{L}_{ml}$ language by an equivalent form:

$$\mathtt{let\ x\ =\ e}_1\ \mathtt{in\ e}_2 \quad \equiv \quad (\mathtt{fun\ (x:ty)\text{->}e}_2)\ \mathtt{e}_1$$

Hence, we can deduce the graph to be generated in rule (LocalG) by composing the graph generated in the rules (ValueG), (CallG) and (ApplyG):

Let us use the following abbreviation:

$$\begin{array}{rcl}
B & = & id_{\mathtt{fun\ (x:ty)\text{->}e}_2} = id^{=}_{\mathtt{e}} \setminus \{\mathtt{x} \xrightarrow{=} \mathtt{x}\} \\
A^0 & = & CallGr^0_x(B, G_1) \\
A^+ & = & CallGr^+_x(B, G_1^+)
\end{array}$$

By applying (ValueG) we have:

$$(ValueG)\frac{}{\mathtt{fun\ (x:ty)\text{->}e}_2 : \rho \Downarrow \mathtt{fun\ (x:ty)\text{->}e}_2 : \rho, B|B} \tag{4.3.1}$$

We now apply the rule (CallG) to 4.3.1 and the premise $\mathtt{e}_1 : \rho \Downarrow \mathtt{v}_1, G_1|G_1^+$ of (LocalG). We obtain:

$$(CallG)\frac{4.3.1 \qquad \mathtt{e}_1 : \rho \Downarrow \mathtt{v}_1, G_1|G_1^+}{(\mathtt{fun\ (x:ty)\text{->}e}_2)\ \mathtt{e}_1 : \rho \underset{c}{\rightarrow} \mathtt{e}_2 : \rho[\mathtt{x} \mapsto \mathtt{v}_1], A^0|A^+} \tag{4.3.2}$$

Finally we apply the rule (ApplyG) to 4.3.2 and the premise $\mathtt{e}_2 : \rho[\mathtt{x} \mapsto \mathtt{v}_1] \Downarrow \mathtt{v}_2, G_2|G_2^+$ of (LocalG). We obtain:

$$(ApplyG)\frac{4.3.2 \qquad \mathtt{e}_2 : \rho[\mathtt{x} \mapsto \mathtt{v}_1] \Downarrow \mathtt{v}_2, G_2|G_2^+}{\mathtt{(fun\ (x:ty)\text{->}e_2)\ e_1} : \rho \Downarrow \mathtt{v}_2, A^0; G_2|A^+; G_2^+}$$

This justifies the correctness of the graphs generated in the rule (LocalG): For $i \in \{0, +\}$:

$$LocalGr^i(\mathtt{x}, G_1, \mathtt{e}, G_2) \triangleq CallGr^i_{\mathtt{x}}(id_{\mathtt{e}} \setminus \{\mathtt{x} \overset{=}{\rightarrow} \mathtt{x}\}, G_1) \ ; G_2$$

Theorem 4.3.2 (Safe Graph Generation). *The size-change graphs generated in rules of table 4.4 and 4.5 are safe.*

The proof is in Appendix B.

4.3.5 Approximate semantics with graph generation

As we saw in section 4.3.1, the program points set $\mathcal{P}$ includes a new element: $?^{\mathtt{int}}$. This element was not used in the definition of the exact semantics (the exact integers and boolean where used instead). But remember that the size-change principle requires $\mathcal{P}$ to be finite. This is because we need to define an approximate semantics which generates a finite possible number of judgement forms.

The approximate semantics is given in table 4.6 and 4.7.

Dealing with indefinite integer or boolean values ($?^{\mathtt{int}}$ and $?^{\mathtt{bool}}$)

As you can see in table 4.6 and 4.7, the size-change graphs generation does not make use of the exact value of integers and boolean and therefore the use of $?^{\mathtt{int}}$ is particularly appropriated.

One may worries about the fact that the set $\mathbb{N}$ is reduced to the single symbol $?^{\mathtt{int}}$. More precisely, since two different integers are considered to be equal, the computation of the closure of the size-change graph could generate a graph for an impossible transitive call. But this is not a problem since first: it is permitted for the set of size-change graphs to be an over approximation of the real set of size-change graphs describing the program's calls. Secondly, the symbol $?^{\mathtt{int}}$ only appears on the right hand side of the judgment forms ($\_ \Downarrow ?^{\mathtt{int}}$), consequently, there will never be any call occurring at program point $?^{\mathtt{int}}$!

Finally, the use of $?^{\mathtt{int}}$ brings us a powerful feature: we can now verify that a particular program terminates for any value of a free variable of type `int`!

Soundness of the approximation

The following lemma states that the approximation of tables 4.6 and 4.7 is sound:

Lemma 4.3.3 (Approximation). *Suppose* $\mathtt{P} : [] \to^* e : \rho$ *then*

$$\begin{array}{llcl}
 & \textit{(exact semantics)} & & \textit{(approximation semantics)} \\[1ex]
\textit{for } e' \in \textit{subexp}(\mathtt{P}), & \mathtt{e} : \rho \to \mathtt{e'} : \rho', G^0|G^+ & \Longrightarrow & \mathtt{e} \to \mathtt{e'}, G^0|G^+ \\[1ex]
\textit{for } v \notin \mathbb{N} \cup \{\mathtt{true}, \mathtt{false}\}, & \mathtt{e} : \rho \Downarrow \mathtt{v} : \rho', G^0|G^+ & \Longrightarrow & \mathtt{e} \Downarrow \mathtt{v}, G^0|G^+ \\
\textit{for } n \in \mathbb{N} & \mathtt{e} : \rho \Downarrow \mathtt{n} : [], G^0|G^+ & \Longrightarrow & \mathtt{e} \Downarrow ?^{\mathtt{int}}, G^0|G^+ \\
\textit{for } b \in \{\mathtt{true}, \mathtt{false}\}, & \mathtt{e} : \rho \Downarrow \mathtt{b} : [], G^0|G^+ & \Longrightarrow & \mathtt{e} \Downarrow ?^{\mathtt{bool}}, G^0|G^+
\end{array}$$

The proof is in Appendix C.

4.3.6 Safe description of the program's calls

Theorem 4.3.4. *Let* $\mathcal{G}^0$ *and* $\mathcal{G}^+$ *be the following sets of size-change graphs:*

$$\mathcal{G}^0 = \{\ G^0_j \mid j > 0 \wedge \exists \mathtt{e}_i, G^0_i (0 \leq i \leq j) : \\ \mathtt{P} = \mathtt{e}_0 \wedge (\mathtt{e}_0 \to \mathtt{e}_1, G^0_1|\_\ ) \wedge \ldots \wedge (\mathtt{e}_{j-1} \to \mathtt{e}_j, G^0_j|\_\ )\ \}$$

$$\mathcal{G}^+ = \{\ G^+_j \mid j > 0 \wedge \exists \mathtt{e}_i, G^+_i (0 \leq i \leq j) : \\ \mathtt{P} = \mathtt{e}_0 \wedge (\mathtt{e}_0 \to \mathtt{e}_1, \_\ |G^+_1) \wedge \ldots \wedge (\mathtt{e}_{j-1} \to \mathtt{e}_j, \_\ |G^+_j)\ \}$$

Then:

(i) $\mathcal{G}^0$ *and* $\mathcal{G}^+$ *are computable,*

(ii) $\mathcal{G}^0$ *and* $\mathcal{G}^+$ *are safe for* $\mathtt{P}$

Proof (i) They are computable by exhaustive application of the approximate rules of tables 4.6 and 4.7 starting with expression P until no new graph or subexpression are found.

The computation terminates because there is a finite number of possible judgment forms and size-change graphs (since $\textit{subexp}(\mathtt{P}) = \mathcal{P}$ is a finite set).

(ii) Let c be an activable call. In the exact environment based semantics of P we have:

$$\mathtt{P} : [] = s_0 \to s_1 \to \ldots \to s_i \xrightarrow{c} s_{i+1}$$

where $s_k = \mathtt{e_k} : \rho_k$ for $k \leq i + 1$.

By the rules of the exact semantics of 4.4 and 4.5 we have:

$$s_i \xrightarrow{c} s_{i+1}, G^0_i|G^+_i$$

By theorem 4.3.2, G^0_i and G^+_i are safe (relatively to the safety definition of SCP^0 and SCP^+ respectively) for the pair (s_i, s_{i+1}).

For $0 \leq k \leq i$, we have $P : [] \rightarrow^* \mathtt{e_k} : \rho_k$ and $\mathtt{e_k} : \rho_k \rightarrow \mathtt{e_{k+1}} : \rho_{k+1}, G^0_k|G^+_k$. Thus by lemma 4.3.3:

$$e_k \rightarrow e_{k+1}, G^0_k|G^+_k \quad \text{for } 0 \leq k \leq i$$

relatively to the approximation semantics.

Hence by definition of the sets $\mathcal{G}^0$ and $\mathcal{G}^+$, $G^0_i \in \mathcal{G}^0$ and $G^+_i \in \mathcal{G}^+$.

■

After computing the two sets defined in theorem 4.3.4 by exhaustive application of the approximation rules, we can apply the second part of the algorithm of section 3.10 to decide size-change termination for SCP^0 and SCP^+.

Canonical forms:

$$\text{(ValueG)}\ \frac{}{\texttt{v} \Downarrow \texttt{v}, id^{=}_{\texttt{e}} | id^{=}_{\texttt{e}}}(\texttt{v} = \texttt{e} : \rho \text{ in canonical form})$$

Variables:

$$\text{(VarG)}\ \frac{}{\texttt{x} : \rho \Downarrow \rho(\texttt{x}), var^{0}_{x\mapsto} | var^{+}_{x\mapsto e'}}(\rho(x) = e' : \rho')$$

Integer equality:

$$\text{(EqTrueG)}\ \frac{\texttt{e}_1 : \rho \Downarrow \texttt{n} : [], \_\ |\_ \qquad \texttt{e}_2 : \rho \Downarrow \texttt{n} : [], \_\ |\_}{\texttt{e}_1 = \texttt{e}_2 : \rho \Downarrow \texttt{true} : [], \emptyset | \emptyset}$$

$$\text{(EqFalseG)}\ \frac{\texttt{e}_1 : \rho \Downarrow \texttt{n} : [], \_\ |\_ \qquad \texttt{e}_2 : \rho \Downarrow \texttt{m} : [], \_\ |\_ \qquad n \neq m}{\texttt{e}_1 = \texttt{e}_2 : \rho \Downarrow \texttt{false} : [], \emptyset\ | \emptyset}$$

Operator:

$$\text{(PredG)}\ \frac{\texttt{e} : \rho \Downarrow \texttt{n} : [], G|\_ \qquad n > 0}{\texttt{pred e} : \rho \Downarrow \texttt{n} - \texttt{1} : [], (\{\bullet \xrightarrow{=} \bullet\} \cup \{\texttt{x} \xrightarrow{\downarrow} \bullet \mid \texttt{x} \xrightarrow{r} \bullet \in G\}) | \emptyset}$$

$$\text{(SuccG)}\ \frac{\texttt{e} : \rho \Downarrow \texttt{n} : [], G|\_}{\texttt{succ e} : \rho \Downarrow \texttt{n} + \texttt{1} : [], (\{\bullet \xrightarrow{=} \bullet\} \cup \{\bullet \xrightarrow{\downarrow} \texttt{x} \mid \bullet \xrightarrow{r} \texttt{x} \in G\}) | \emptyset}$$

Function application:

$$\text{(ApplyG)}\ \frac{\texttt{e} : \rho \underset{c/if}{\longrightarrow} \texttt{e}' : \rho', G|G^{+} \qquad \texttt{e}' : \rho' \Downarrow \texttt{v}, G'|G'^{+}}{\texttt{e} : \rho \Downarrow \texttt{v}, (G; G')|(G^{+}; G'^{+})}$$

Local definition:

$$\text{(LocalG)}\ \frac{\texttt{e}_1 : \rho \Downarrow \texttt{v}_1, G_1|G_1^{+} \qquad \texttt{e}_2 : \rho[\texttt{x} \mapsto \texttt{v}_1] \Downarrow \texttt{v}_2, G_2|G_2^{+}}{\texttt{let x = e}_1 \texttt{ in e}_2 : \rho \Downarrow \texttt{v}_2, LocalGr^{0}(\texttt{x}, G_1, \texttt{e}_2, G_2) | LocalGr^{+}(\texttt{x}, G_1^{+}, \texttt{e}_2, G_2^{+})}$$

where

$$\begin{array}{rcl}
idv_{\texttt{e}} & \triangleq & \{\texttt{x} \xrightarrow{=} \texttt{x} \mid \texttt{x} \in fv(\texttt{e})\} \\
var^{0}_{x\mapsto} & \triangleq & \{x \xrightarrow{=} \bullet\} \cup \{\bullet \xrightarrow{=} \bullet\} \\
var^{+}_{x\mapsto e'} & \triangleq & \{x \xrightarrow{=} \bullet\} \cup \{\bullet \xrightarrow{\downarrow} \bullet\} \cup \{x \xrightarrow{\downarrow} y \mid y \in fv(e')\} \\
CallGr^{0}_{\texttt{x}}(G_1, G_2) & \triangleq & \{\bullet = \bullet\} \cup \{\ \texttt{y} \xrightarrow{r} \texttt{z} \mid \texttt{y} \xrightarrow{r} \texttt{z} \in G_1\} \cup \{\texttt{y} \xrightarrow{r} \texttt{x} \mid \texttt{y} \xrightarrow{r} \bullet \in G_2\} \\
CallGr^{+}_{\texttt{x}}(G_1, G_2) & \triangleq & G_1^{-\bullet} \cup G_2^{\bullet\mapsto\texttt{x}} \\
LocalGr^{0}(\texttt{x}, G_1, \texttt{e}, G_2) & \triangleq & CallGr^{0}_{\texttt{x}}(id^{=}_{\texttt{e}} \setminus \{\texttt{x} \xrightarrow{=} \texttt{x}\}, G_1)\ ; G_2 \\
LocalGr^{+}(\texttt{x}, G_1, \texttt{e}, G_2) & \triangleq & CallGr^{+}_{\texttt{x}}(id^{=}_{\texttt{e}} \setminus \{\texttt{x} \xrightarrow{=} \texttt{x}\}, G_1)\ ; G_2
\end{array}$$

Table 4.4: $\mathcal{L}_{ml}$ environment based evaluation semantics with graphs generation

Conditional:

$$(\text{IfCondCallG})\quad \frac{}{\texttt{if e then e}_1\texttt{ else e}_2 : \rho \rightarrow \texttt{e} : \rho, idv_{\texttt{e}} | id^{\downarrow}_{\texttt{e}}}$$

$$(\text{IfTrueCallG})\quad \frac{\texttt{e} : \rho \Downarrow \texttt{true} : \_ \,, \_ \,|\_}{\texttt{if e then e}_1\texttt{ else e}_2 : \rho \underset{if}{\rightarrow} \texttt{e}_1 : \rho, id^{=}_{\texttt{e}_1} | id^{\downarrow}_{\texttt{e}_1}}$$

$$(\text{IfFalseCallG})\quad \frac{\texttt{e} : \rho \Downarrow \texttt{false} : \_ \,, \_ \,|\_}{\texttt{if e then e}_1\texttt{ else e}_2 : \rho \underset{if}{\rightarrow} \texttt{e}_2 : \rho, id^{=}_{\texttt{e}_2} | id^{\downarrow}_{\texttt{e}_2}}$$

Integer equality:

$$(\text{EqCondTrueG})\quad \frac{}{\texttt{e}_1 = \texttt{e}_2 : \rho \rightarrow \texttt{e}_1 : \rho, idv_{\texttt{e}_1} | id^{\downarrow}_{\texttt{e}_1}}$$

$$(\text{EqCondFalseG})\quad \frac{}{\texttt{e}_1 = \texttt{e}_2 : \rho \rightarrow \texttt{e}_2 : \rho, idv_{\texttt{e}_2} | id^{\downarrow}_{\texttt{e}_2}}$$

Operator:

$$(\text{PredCallG})\quad \frac{}{\texttt{pred e} : \rho \rightarrow \texttt{e} : \rho, idv_{\texttt{e}} | id^{\downarrow}_{\texttt{e}}}$$

$$(\text{SuccCallG})\quad \frac{}{\texttt{succ e} : \rho \rightarrow \texttt{e} : \rho, id^{\downarrow}_{\texttt{e}} | id^{\downarrow}_{\texttt{e}}}$$

Local definition:

$$(\text{LocalDefCallG})\quad \frac{}{\texttt{let x = e}_1\texttt{ in e}_2 : \rho \rightarrow \texttt{e}_1 : \rho, idv_{\texttt{e}_1} | id^{\downarrow}_{e_1}}$$

$$(\text{LocalBodyCallG})\quad \frac{\texttt{e}_1 : \rho \Downarrow \texttt{v}_1, G_1 | G_1^{+}}{\texttt{let x = e}_1\texttt{ in e}_2 : \rho \rightarrow \texttt{e}_2 : \rho[\texttt{x} \mapsto \texttt{v}_1], (id^{=}_{\texttt{e}_2} \setminus \{x \overset{=}{\rightarrow} x\}) | id^{\downarrow}_{\texttt{e}_2}}$$

Function application:

$$(\text{OperatorG})\quad \frac{}{\texttt{e}_1\texttt{e}_2 : \rho \rightarrow \texttt{e}_1 : \rho, idv_{\texttt{e}_1} | id^{\downarrow}_{e_1}}$$

$$(\text{OperandG})\quad \frac{\texttt{e}_1 : \rho \Downarrow \texttt{v}_1, \_ \,|\_}{\texttt{e}_1\texttt{e}_2 : \rho \rightarrow \texttt{e}_2 : \rho, idv_{\texttt{e}_2} | id^{\downarrow}_{e_2}}$$

$$(\text{CallG})\quad \frac{\texttt{e}_1 : \rho \Downarrow \texttt{fun (x:ty)->e}_0 : \rho_0, G_1 | G_1^{+} \qquad \texttt{e}_2 : \rho \Downarrow \texttt{v}_2, G_2 | G_2^{+}}{\texttt{e}_1\texttt{e}_2 : \rho \underset{c}{\rightarrow} \texttt{e}_0 : \rho_0[\texttt{x} \mapsto \texttt{v}_2], CallGr^{0}_{x}(G_1, G_2) | CallGr^{+}_{x}(G_1^{+}, G_2^{+})}$$

$$(\text{CallRecG})\quad \frac{\texttt{e}_1 : \rho \Downarrow \overbrace{\texttt{fun f=(x:ty)->e}_0 : \rho_0}^{\texttt{v}}, G_1 | G_1^{+} \qquad \texttt{e}_2 : \rho \Downarrow \texttt{v}_2, G_2 | G_2^{+}}{\texttt{e}_1\texttt{e}_2 : \rho \underset{c}{\rightarrow} \texttt{e}_0 : \rho_0[\texttt{x} \mapsto \texttt{v}_2, \texttt{f} \mapsto \texttt{v}], CallGr^{0}_{x}(G_1, G_2) | CallGr^{+}_{x}(G_1^{+}, G_2^{+})}$$

Table 4.5: $\mathcal{L}_{ml}$ environment based call semantics with graphs generation

Canonical forms:

$$\text{(ValueAG)}\ \frac{}{\texttt{v} \Downarrow \texttt{v}, id^{=}_{\texttt{e}} | id^{=}_{\texttt{e}}}(\texttt{v}\text{ is a function})$$

$$\text{(ValueAG')}\ \frac{}{\texttt{n} \Downarrow ?^{\texttt{int}}, id^{=}_{\texttt{e}} | id^{=}_{\texttt{e}}}(\texttt{n} \in \mathbb{N} \cup \{?^{\texttt{int}}\})$$

$$\text{(ValueAG")}\ \frac{}{\texttt{b} \Downarrow ?^{\texttt{bool}}, id^{=}_{\texttt{e}} | id^{=}_{\texttt{e}}}(\texttt{b} \in \{true, false, ?^{\texttt{bool}}\})$$

Variables:

$$\text{(VarAG)}\ \frac{\texttt{e}_1\texttt{e}_2 \in \mathit{subexp}(\texttt{P}) \quad \texttt{e}_1 \Downarrow \left\{ \begin{array}{l} \texttt{fun (x : ty)->e}_0 \\ \text{or } \texttt{fun f = (x : ty)->e}_0 \end{array} \right. , \_ | \_ \quad \texttt{e}_2 \Downarrow \texttt{v}_2, \_ | \_}{\texttt{x} \Downarrow \texttt{v}_2, var^{0}_{x\mapsto} | var^{+}_{x\mapsto v_2}}$$

$$\text{(VarRecAG)}\ \frac{\overbrace{\texttt{fun f = (x : ty)->e}_0}^{\texttt{v}} \in \mathit{subexp}(\texttt{P})}{\texttt{f} \Downarrow \texttt{v}, var^{0}_{\texttt{f}\mapsto} | var^{+}_{\texttt{f}\mapsto v}}$$

$$\text{(VarLetAG)}\ \frac{\texttt{let x = e}_1 \texttt{ in e}_2 \in \mathit{subexp}(\texttt{P}) \quad \texttt{e}_1 \Downarrow \texttt{v}_1, \_ | \_}{\texttt{x} \Downarrow \texttt{v}_1, var^{0}_{x\mapsto} | var^{+}_{x\mapsto v_1}}$$

Integer equality:

$$\text{(EqAG)}\ \frac{}{\texttt{e}_1 = \texttt{e}_2 \Downarrow ?^{\texttt{bool}}, \emptyset | \emptyset}$$

Operator:

$$\text{(PredAG)}\ \frac{\texttt{e} : \rho \Downarrow ?^{\texttt{int}}, G | \_}{\texttt{pred e} : \rho \Downarrow ?^{\texttt{int}}, (\{\bullet \xrightarrow{=} \bullet\} \cup \{\texttt{x} \xrightarrow{\downarrow} \bullet \mid \texttt{x} \xrightarrow{r} \bullet \in G\}) | \emptyset}$$

$$\text{(SuccAG)}\ \frac{\texttt{e} : \rho \Downarrow ?^{\texttt{int}}, G | \_}{\texttt{succ e} \Downarrow ?^{\texttt{int}}, (\{\bullet \xrightarrow{=} \bullet\} \cup \{\bullet \xrightarrow{\downarrow} \texttt{x} \mid \bullet \xrightarrow{r} \texttt{x} \in G\}) | \emptyset}$$

Local definition:

$$\text{(LocalAG)}\ \frac{\texttt{e}_1 \Downarrow \texttt{v}_1, G_1 | G_1^{+} \qquad \texttt{e}_2 \Downarrow \texttt{v}_2, G_2 | G_2^{+}}{\texttt{let x = e}_1 \texttt{ in e}_2 \Downarrow \texttt{v}_2, \mathit{LocalGr}^{0}(\texttt{x}, G_1, \texttt{e}_2, G_2) | \mathit{LocalGr}^{+}(\texttt{x}, G_1^{+}, \texttt{e}_2, G_2^{+})}$$

Function application:

$$\text{(ApplyAG)}\ \frac{\texttt{e} \xrightarrow[c/if]{} \texttt{e}', G | G^{+} \qquad \texttt{e}' \Downarrow \texttt{v}, G' | G'^{+}}{\texttt{e} \Downarrow \texttt{v}, (G; G') | (G^{+}; G'^{+})}$$

Table 4.6: $\mathcal{L}_{ml}$ approximate evaluation semantics with graphs generation

Conditional:

$$(\text{IfCondCallAG})\ \frac{}{\texttt{if e then e}_1\ \texttt{else e}_2 \rightarrow \texttt{e}, idv_{\texttt{e}}|id^{\downarrow}_{\texttt{e}}}$$

$$(\text{IfTrueCallAG})\ \frac{}{\texttt{if e then e}_1\ \texttt{else e}_2 \underset{if}{\rightarrow} \texttt{e}_1, id^{=}_{\texttt{e}_1}|id^{\downarrow}_{\texttt{e}_1}}$$

$$(\text{IfFalseCallAG})\ \frac{}{\texttt{if e then e}_1\ \texttt{else e}_2 \underset{if}{\rightarrow} \texttt{e}_2, id^{=}_{\texttt{e}_2}|id^{\downarrow}_{\texttt{e}_2}}$$

Integer equality:

$$(\text{EqCondTrueAG})\ \frac{}{\texttt{e}_1 = \texttt{e}_2 \rightarrow \texttt{e}_1, idv_{\texttt{e}_1}|id^{\downarrow}_{\texttt{e}_1}} \qquad (\text{EqCondFalseAG})\ \frac{}{\texttt{e}_1 = \texttt{e}_2 \rightarrow \texttt{e}_2, idv_{\texttt{e}_2}|id^{\downarrow}_{\texttt{e}_2}}$$

Operator:

$$(\text{PredCallAG})\ \frac{}{\texttt{pred e} \rightarrow \texttt{e}, idv_{\texttt{e}}|id^{\downarrow}_{\texttt{e}}} \qquad (\text{SuccCallAG})\ \frac{}{\texttt{succ e} \rightarrow \texttt{e}, id^{\downarrow}_{\texttt{e}}|id^{\downarrow}_{\texttt{e}}}$$

Local definition:

$$(\text{LocalDefCallAG})\ \frac{}{\texttt{let x = e}_1\ \texttt{in e}_2 \rightarrow \texttt{e}_1, idv_{\texttt{e}_1}|id^{\downarrow}_{e_1}}$$

$$(\text{LocalBodyCallAG})\ \frac{}{\texttt{let x = e}_1\ \texttt{in e}_2 \rightarrow \texttt{e}_2, (idv_{\texttt{e}_2} \setminus \{x \overset{=}{\rightarrow} x\})|id^{\downarrow}_{\texttt{e}_2}}$$

Function application:

$$(\text{OperatorAG})\ \frac{}{\texttt{e}_1\texttt{e}_2 \rightarrow \texttt{e}_1, idv_{\texttt{e}_1}|id^{\downarrow}_{e_1}} \qquad (\text{OperandAG})\ \frac{}{\texttt{e}_1\texttt{e}_2 \rightarrow \texttt{e}_2, idv_{\texttt{e}_2}|id^{\downarrow}_{e_2}}$$

$$(\text{CallAG})\ \frac{\texttt{e}_1 \Downarrow \left\{\begin{array}{l}\texttt{fun (x:ty)->}e_0\\ \text{or }\texttt{fun f=(x:ty)->}e_0\end{array}\right., G_1|G_1^{+} \qquad \texttt{e}_2 \Downarrow \texttt{v}_2, G_2|G_2^{+}}{\texttt{e}_1\texttt{e}_2 \underset{c}{\rightarrow} \texttt{e}_0, CallGr^{0}_{x}(G_1, G_2)|CallGr^{+}_{x}(G_1^{+}, G_2^{+})}$$

Table 4.7: $\mathcal{L}_{ml}$ approximate call semantics with graphs generation

4.3.7 Improvements

The improvement described in section 3.11.2 has also been implemented for the core ML case: the set of program points becomes

$$\mathcal{P} = nodes(\texttt{P}) \cup \{?^{\texttt{int}}, ?^{\texttt{bool}}\} \qquad \text{instead of} \qquad subexp(\texttt{P}) \cup \{?^{\texttt{int}}, ?^{\texttt{bool}}\}$$

Then by giving small modifications to the rules (VarAG), (VarRecAG) and (VarLetAG) we can avoid the problem of variable renaming and at the same time reduce the number of approximation judgment forms generated.

For instance, the rule (VarRecAG) of table 4.6 has been implemented using the following definition:

$$(VarRecAG)\frac{i_{\texttt{funrec}} \in nodes(\texttt{P})}{i_{\texttt{f}} \Downarrow i_{\texttt{funrec}}, var^{0}_{\texttt{f}\mapsto}|var^{+}_{\texttt{f}\mapsto v}}$$

with the following side-conditions:

1. $node(i_{\texttt{funrec}}) = \langle \texttt{fun f} = (\texttt{x} : \texttt{ty})\texttt{->}, i_{\texttt{e}_0}\rangle$
2. `v` is the expression represented by the node $i_{\texttt{funrec}}$.
3. $node(i_{\texttt{f}}) = \langle \texttt{f} \rangle$
4. The node $i_{\texttt{f}}$ belongs to the subtree rooted at node $i_{\texttt{e}_0}$ and represents a free occurrence of variable `f`.

The effect of the last side-condition is to limit the scope of the variable `f` to the body $\texttt{e}_0$ of the recursively defined function `f`.

Similar definition are used for rules (VarAG) and (VarLetAG).

4.3.8 Example

In order to understand how the algorithm really works, we now apply it manually on an example. Consider the following recursively defined function which computes the minimum of two natural numbers:

```
min.cml
let rec min x y =
  if x = 0 then  0
  else
    if y = 0 then 0
    else
      succ (min (pred x) (pred y))
in
  min ? ?
;;
```

(The special character `?` is interpreted as $?^{\texttt{int}}$ by the parser). After removing the syntactic sugar and having numbered the program subexpressions we obtain the following code:

min.cml

```
1: let min =
  2: fun f=(x:int)->
    3: fun (y:int) ->
      4: if x = 0 then 0
      else
        9: if y = 0 then 0
        else
          14: (succ 15: (f (pred x) (pred y)))
in
  min ? ?;;
```

Let us apply the rules of tables 4.6 and 4.7 in order to approximate the evaluation and call semantics of this program. Our goal is to check whether the program is size-change terminating relatively to the first size-change principle (SCP^0). For this reason, we only compute the first graph component of the judgement forms:

$$\text{(ValueAG)} \qquad \frac{}{\boxed{2} \Downarrow \texttt{fun f=(x:int)->}\boxed{3}, \{\bullet \xrightarrow{=} \bullet\}} \tag{4.3.2a}$$

$$\text{(VarLetAG)} \qquad \frac{\texttt{let min=}\boxed{2}\ \texttt{in min}\ ?^{\texttt{int}}\ ?^{\texttt{int}} \in \mathit{subexp}(\texttt{P}) \qquad 4.3.2a}{\texttt{min} \Downarrow \boxed{2}, \{\bullet \xrightarrow{=} \bullet, \texttt{min} \xrightarrow{=} \bullet\}} \tag{4.3.2b}$$

$$\text{(ValueAG)} \qquad \frac{}{?^{\texttt{int}} \Downarrow ?^{\texttt{int}}, \{\bullet \xrightarrow{=} \bullet\}} \tag{4.3.2c}$$

$$\text{(CallRecAG)} \qquad \frac{4.3.2b \qquad 4.3.2c}{\texttt{min}\ ?^{\texttt{int}} \underset{c}{\rightarrow} \boxed{3}, \{\bullet \xrightarrow{=} \bullet\}} \tag{4.3.2d}$$

$$\text{(VarAG)} \qquad \frac{\texttt{min}\ ?^{\texttt{int}} \in \mathit{subexp}(\texttt{P}) \qquad 4.3.2b \qquad 4.3.2c}{\texttt{x} \Downarrow ?^{\texttt{int}}, \{\bullet \xrightarrow{=} \bullet, \texttt{x} \xrightarrow{=} \bullet\}} \tag{4.3.2e}$$

$$\text{(PredAG)} \qquad \frac{4.3.2e}{\texttt{pred x} \Downarrow ?^{\texttt{int}}, \{\bullet \xrightarrow{=} \bullet, \texttt{x} \xrightarrow{\downarrow} \bullet\}} \tag{4.3.2f}$$

$$\text{(VarRecAG)} \qquad \frac{\texttt{fun f=(x:int)->}\boxed{3} \in \mathit{subexp}(\texttt{P})}{\texttt{f} \Downarrow \boxed{2}, \{\bullet \xrightarrow{=} \bullet, \texttt{f} \xrightarrow{=} \bullet\}} \tag{4.3.2g}$$

$$\text{(CallRecAG)} \qquad \frac{4.3.2g \qquad 4.3.2f}{\texttt{f (pred x)} \underset{c}{\rightarrow} \boxed{3}, \{\bullet \xrightarrow{=} \bullet, \texttt{x} \xrightarrow{\downarrow} \texttt{x}\}} \tag{4.3.2h}$$

$$\text{(ValueAG)} \qquad \frac{}{\boxed{3} \Downarrow \boxed{3}, \{\bullet \xrightarrow{=} \bullet, \texttt{x} \xrightarrow{=} \texttt{x}\}} \tag{4.3.2i}$$

$$\text{(ApplyAG)}\qquad \dfrac{4.3.2h \qquad 4.3.2i}{\texttt{f (pred x)} \Downarrow \boxed{3}, \{\bullet \xrightarrow{=} \bullet, \texttt{x} \xrightarrow{\downarrow} \texttt{x}\}} \qquad (4.3.2j)$$

$$\text{(ApplyAG)}\qquad \dfrac{4.3.2d \qquad 4.3.2i}{\texttt{min } ?^{\texttt{int}} \Downarrow \boxed{3}, \{\bullet \xrightarrow{=} \bullet\}} \qquad (4.3.2k)$$

$$\text{(CallAG)}\qquad \dfrac{4.3.2k \qquad 4.3.2c}{\texttt{min } ?^{\texttt{int}}\ ?^{\texttt{int}} \underset{c}{\rightarrow} \boxed{3}, \{\bullet \xrightarrow{=} \bullet\}} \qquad (4.3.2l)$$

$$\text{(VarAG)}\qquad \dfrac{\texttt{min } ?^{\texttt{int}} \in \mathit{subexp}(\texttt{P}) \qquad 4.3.2k \qquad 4.3.2c}{\texttt{y} \Downarrow ?^{\texttt{int}}, \{\bullet \xrightarrow{=} \bullet, \texttt{y} \xrightarrow{=} \bullet\}} \qquad (4.3.2m)$$

$$\text{(PredAG)}\qquad \dfrac{4.3.2m}{\texttt{pred y} \Downarrow ?^{\texttt{int}}, \{\bullet \xrightarrow{=} \bullet, \texttt{y} \xrightarrow{\downarrow} \bullet\}} \qquad (4.3.2n)$$

$$\text{(CallAG)}\qquad \dfrac{4.3.2j \qquad 4.3.2n}{\texttt{f (pred x)(pred y)} \underset{c}{\rightarrow} \boxed{4}, \{\bullet \xrightarrow{=} \bullet, \texttt{x} \xrightarrow{\downarrow} \texttt{x}, \texttt{y} \xrightarrow{\downarrow} \texttt{y}\}} \qquad (4.3.2o)$$

The following calls also occur:

$$(\mathit{LocalBodyCallAG})\dfrac{}{\texttt{P} \rightarrow \texttt{min}?^{\texttt{int}}\ ?^{\texttt{int}}, \{\bullet \xrightarrow{=} \bullet\}}$$

$$(\mathit{IfFalseCallAG})\dfrac{}{\boxed{4} \underset{\mathit{if}}{\rightarrow} \boxed{9}, \{\bullet \xrightarrow{=} \bullet, x \xrightarrow{=} x, y \xrightarrow{=} y\}}$$

$$(\mathit{IfFalseCallAG})\dfrac{}{\boxed{9} \underset{\mathit{if}}{\rightarrow} \boxed{14}, \{\bullet \xrightarrow{=} \bullet, x \xrightarrow{=} x, y \xrightarrow{=} y\}}$$

$$(\mathit{SuccCallAG})\dfrac{}{\texttt{succ}\ \boxed{15} \rightarrow \boxed{15}, \{\bullet \xrightarrow{\downarrow} \bullet, x \xrightarrow{=} x, y \xrightarrow{=} y, \mathit{min} \xrightarrow{=} \mathit{min}\}}$$

We can now exhibit a loop activated at control point $\boxed{4}$:

$$\begin{array}{rcl}
\texttt{P} \rightarrow \texttt{min } ?^{\texttt{int}}\ ?^{\texttt{int}} \underset{c}{\rightarrow} \boxed{4} & \underset{\mathit{if}}{\rightarrow} & \boxed{9}, \{\bullet \xrightarrow{=} \bullet, x \xrightarrow{=} x, y \xrightarrow{=} y, \mathit{min} \xrightarrow{=} \mathit{min}\} \\
\boxed{9} & \underset{\mathit{if}}{\rightarrow} & \boxed{14}, \{\bullet \xrightarrow{=} \bullet, x \xrightarrow{=} x, y \xrightarrow{=} y, \mathit{min} \xrightarrow{=} \mathit{min}\} \\
\boxed{14} & \underset{\mathit{pred}}{\rightarrow} & \boxed{15}, \{\bullet \xrightarrow{\downarrow} \bullet, x \xrightarrow{=} x, y \xrightarrow{=} y, \mathit{min} \xrightarrow{=} \mathit{min}\} \\
\boxed{15} & \underset{c}{\rightarrow} & \boxed{4}, \{\bullet \xrightarrow{=} \bullet, x \xrightarrow{\downarrow} x, y \xrightarrow{\downarrow} y\}
\end{array}$$

By composing the call size-change graphs, we obtain the graph $\{\bullet \xrightarrow{\downarrow} \bullet, x \xrightarrow{\downarrow} x, y \xrightarrow{\downarrow} y\}$ which describes the loop $\boxed{4} \rightarrow^{*} \boxed{4}$.

Since the graph contains an arc of type $x \xrightarrow{\downarrow} x$, the loop is descending. By applying exhaustively all the rules, one can check that this is the only possible loop. Hence the program is size-change terminating. And since we use the special symbol $?^{\texttt{int}}$, we know that the function `min` terminates for any value of the parameters.

Here is the output of the analysis obtained when running my implementation of the algorithm on the `min.cml` example:

```
$ ./sct.opt.exe -ml min.cml
Exhaustive application of the judgment form rules...
Number of jf:54
Preparing for the closure computation...
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Computation of the closure by graph composition...
Loops extraction...
Loops analysis...
Program is size change terminating!
All the loops are descending:
4->*4,[*>*, x>x, y>y][9, 14, 15]
9->*9,[*>*, x>x, y>y][14, 15, 4]
14->*14,[*>*, x>x, y>y][15, 4, 9]
15->*15,[*>*, x>x, y>y][4, 9, 14]
Program is terminating on all input values!
Execution time: 0.02s
```

Note that the four detected loops ($\boxed{4} \to^* \boxed{4}$, $\boxed{9} \to^* \boxed{9}$, $\boxed{14} \to^* \boxed{14}$, $\boxed{15} \to^* \boxed{15}$) are in fact the same one.

We have verified with this example that the new algorithm works better than the first approach: it succeeds in detecting termination of the program `min` whereas the first approach failed.

4.3.9 Results

This section provides the results obtained on several interesting examples.

Infinite loop

The following program loops infinitely:

loop.cml

```
let loop =
  fun loop=(x:int)-> 3:(loop x)
in
  loop 0;;
```

As expected it is not size-change terminating:

```
$ ./sct.opt -ml loop.cml
Number of jf:14
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Program is not size change terminating! The critical loops are:
3->*3,[*=*, x=x][]
=== SIZE-CHANGE PRINCIPLE FOR HIGHER-ORDER EXPRESSION ===
Program is not size change terminating! The critical loops are:
3->*3,[x=x][]

Program is not size-change terminating.
```

Ackerman's function

The ackerman function can be defined in two ways:

- First using church numerals as we did in the λ-calculus case:

ackerman.chnum.cml

```
let b =
  fun g -> fun n -> 4:(n g 8:(g 1))
in
  let suc =
    fun k -> fun s -> fun z -> 15:(s 17:(k s z))
  in
    let two =
      fun s -> fun z -> (s (s z))
    in
      let three =
        fun s -> fun z -> 33:(s 35:(s 37:(s z)))
      in
        ((fun m -> ((m b) suc) two) three);;
```

In that case, the second size-change principle succeeds in detecting the SCT condition:

```
$ ./sct.opt -ml ackerman_chnum.cml
Number of jf:119
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Program is not size change terminating! The critical (ie. not
descending) loops are:
4->*4,[*=*][33]
4->*4,[][8]
8->*8,[][4]
15->*15,[s=s, z=z][17]
17->*17,[s=s, z=z][15]
33->*33,[][4, 8, 4]
33->*33,[*=*][4]
35->*35,[][4, 33]
37->*37,[][4, 33, 35]

=== SIZE-CHANGE PRINCIPLE FOR HIGHER-ORDER EXPRESSION ===
Program is size change terminating! All the loops are descending:
4->*4,[g>g][33]
8->*8,[g>g][4]
15->*15,[k>k, s=s, z=z][17]
17->*17,[k>k, s=s, z=z][15]
33->*33,[s>s][4]
35->*35,[s>s][4, 33]
37->*37,[s>s][4, 33, 35]

Program is terminating on all input values!
```

- The $\mathcal{L}_{ml}$ language allows us to define ackerman's function intuitively using the natural recursive definition:

ackerman.cml

```
let ackerman =
  fun ackerman=(m)-> fun n ->
  4:if m = 0 then
    succ n
  else
    10:if n = 0 then
      14:(ackerman (pred m) 1)
    else
      20:((ackerman (pred m)) 25:(ackerman m (pred n)))
in
  ackerman ? ?;;
```

In that case, the first size-change principle succeeds in detecting the SCT condition:

```
$ ./sct.opt -ml ackerman.cml
Number of jf:74
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Program is size change terminating! All the loops are descending:
4->*4,[m>m][10, 14, 4, 10, 20, 25]
4->*4,[n>n, m=m][10, 20, 25]
4->*4,[m>m, *=*][10, 14]
10->*10,[n>n, m=m][20, 25, 4]
10->*10,[m>m][14, 4, 10, 20, 25, 4]
10->*10,[m>m, *=*][14, 4]
14->*14,[m>m][4, 10, 20, 25, 4, 10]
14->*14,[m>m, *=*][4, 10]
20->*20,[n>n, m=m][25, 4, 10]
20->*20,[m>m][4, 10, 20, 25, 4, 10]
20->*20,[m>m, *=*][4, 10]
25->*25,[m>m][4, 10, 14, 4, 10, 20]
25->*25,[n>n, m=m][4, 10, 20]

=== SIZE-CHANGE PRINCIPLE FOR HIGHER-ORDER EXPRESSION ===
Program is not size change terminating! The critical loops are:
4->*4,[m=m][10, 20, 25]
4->*4,[][10, 20]
```

```
10->*10,[m=m][20, 25, 4]
10->*10,[][14, 4, 10, 14, 4]
14->*14,[][4, 10]
20->*20,[m=m][25, 4, 10]
20->*20,[][4, 10]
25->*25,[][4, 10, 14, 4, 10, 20]
25->*25,[m=m][4, 10, 20]

Program is terminating on all input values!
```

With this second definition of the function we obtain a more general result: the program terminates for any value of the input integers. This is due to the use of the special symbol $?^{\text{int}}$.

Counter-example

It is easy to build a counter-example. Consider the following program:

counter.cml

```
let counter =
  fun counter=(x:int)-> 3:
  if x = 0 then
    7:(counter (succ x))
  else
    1
in
  counter 0;;
```

This program is obviously terminating however it is not size-change terminating:

```
$ ./sct.opt -ml counter.cml
Exhaustive application of the judgment form rules...
Number of jf:29
Preparing for the closure computation...
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Computation of the closure by graph composition...
Loops extraction...
Loops analysis...
Program is not size change terminating!
The critical (ie. not descending) loops are:
3->*3,[*=*][7]
7->*7,[*=*][3]

=== SIZE-CHANGE PRINCIPLE FOR HIGHER-ORDER EXPRESSION ===
Computation of the closure by graph composition...
Loops extraction...
Loops analysis...
Program is not size change terminating!
The critical (ie. not descending) loops are:
3->*3,[][7]
```

```
7->*7,[][3]

Program is not size-change terminating.
```

Errors

Consider the following program:

exception.cml

```
let desc =
  fun desc=(y:int)-> 3:(desc (pred y))
in
  desc ?;;
```

It is size-change terminating:

```
$ ./sct.opt -ml error.cml
Number of jf:16
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Program is size change terminating! All the loops are descending:
3->*3,[*=*, y>y][]
```

It terminates because there is an infinite descent which eventually causes an error. Now, let us compose the program `loop`, which loops infinitely, with the program `desc`:

loopexception.cml

```
let loop =
  fun loop=(x:int)-> loop x
in
  let desc =
    fun desc=(y)-> 8:(desc (pred y))
  in
    (loop (desc ?));;
```

The program is still terminating:

```
$ ./sct.opt -ml looperror.cml
Number of jf:23
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Program is size change terminating! All the loops are descending:
8->*8,[*=*, y>y][]
Program is terminating on all input values!
Execution time: 0.s
```

The reason is that with the call-by-value evaluation, the error occurs before the evaluation of the `loop` function.

Lucas sequences

Lucas sequences are generalization of Fibonacci numbers:

lucas.cml

```
1:let add =
  fun add=(x)-> fun y -> 4:
  if x = 0 then y
  else
    9:(add (pred x) (succ y))
in
  let sub =
    fun sub=(x)-> fun y -> 19:
    if y = 0 then x
    else
      24:(sub (pred x) (pred y))
  in
    let times =
      fun times=(x)-> fun y -> 34:
      if y = 0 then 0
      else
        39:((add x) 43:((times x) (pred y)))
    in
      let lucas =
        fun lucas=(p)-> fun q -> fun n -> 53:
        if n = 0 then 0
        else
          58:
          if n = 1 then 1
          else
            63:(64:(sub 66:((times p) 70:((lucas p q) (pred n))))
78:((times q) 82:((lucas p q) (pred (pred n)))))
      in
        lucas ? ? ?;;
```

```
$ ./sct.opt -ml lucas.cml
Number of jf:208
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Program is size change terminating! All the loops are descending:
4->*4,[x>x, *=*][9]
9->*9,[x>x, *=*][4]
19->*19,[x>x, y>y, *=*][24]
24->*24,[x>x, y>y, *=*][19]
34->*34,[y>y, x=x][39, 43]
39->*39,[y>y, x=x][43, 34]
43->*43,[y>y, x=x][34, 39]
53->*53,[n>n, p=p, q=q][58, 63, 64, 66, 70]
58->*58,[n>n, p=p, q=q][63, 64, 66, 70, 53]
63->*63,[n>n, p=p, q=q][64, 66, 70, 53, 58]
64->*64,[n>n, p=p, q=q][66, 70, 53, 58, 63]
66->*66,[n>n, p=p, q=q][70, 53, 58, 63, 64]
```

```
70->*70,[n>n, p=p, q=q][53, 58, 63, 64, 66]
78->*78,[n>n, p=p, q=q][82, 53, 58, 63]
82->*82,[n>n, p=p, q=q][53, 58, 63, 78]
Program is terminating on all input values!
```

Y combinator

Instead of using the `let rec` feature of the $\mathcal{L}_{ml}$ language, one can use the Y-combinator. However this makes the analysis run slower since the Y combinator is expanded and produces more judgement forms.

Moreover each recursively defined function must have its own copy of the Y combinator.

For instance using the Y combinator, the `lucas` example has to be rewritten like this:

lucas.ycomb.cml

```
let add =
let y = fun p -> (fun q -> p (fun s -> q q s)) (fun t -> p (fun u -> t t
u))
in
  y (fun f x y ->
    if x = 0 then y
    else
      f (pred x) (succ y)
  )
in

(* precondition: x>=y *)
let sub =
let y = fun p -> (fun q -> p (fun s -> q q s)) (fun t -> p (fun u -> t t
u))
in
  y (fun f x y ->
       if y = 0 then x
       else
         f (pred x) (pred y)
     )
in

let times =
let y = fun p -> (fun q -> p (fun s -> q q s)) (fun t -> p (fun u -> t t
u))
in
  y (fun f x y ->
      if y = 0 then 0
      else
        add x (f x (pred y)))
in

let lucas =
let y = fun p -> (fun q -> p (fun s -> q q s)) (fun t -> p (fun u -> t t
u))
in
```

```
  y
  (fun f p q n ->
     if n = 0 then 0
     else
       if n = 1 then 1
       else
         sub
          (times p (f p q (pred n)))
          (times q (f p q (pred (pred n))))
  )
in
lucas ? ? ?
;;
```

And the size-change termination is still detected:

```
$ ./sct.opt -ml lucas.ycomb.cml
Number of jf:418
=== SIZE-CHANGE PRINCIPLE FOR INTEGERS ===
Program is size change terminating! All the loops are descending:
28->*28,[x>x, *=*][33]
33->*33,[x>x, *=*][28]
67->*67,[x>x, y>y, *=*][72]
72->*72,[x>x, y>y, *=*][67]
106->*106,[y>y, x=x][111, 115]
111->*111,[y>y, x=x][115, 106]
115->*115,[y>y, x=x][106, 111]
149->*149,[n>n, p=p, q=q][154, 159, 160, 162, 166]
154->*154,[n>n, p=p, q=q][159, 160, 162, 166, 149]
159->*159,[n>n, p=p, q=q][160, 162, 166, 149, 154]
160->*160,[n>n, p=p, q=q][162, 166, 149, 154, 159]
162->*162,[n>n, p=p, q=q][166, 149, 154, 159, 160]
166->*166,[n>n, p=p, q=q][149, 154, 159, 160, 162]
174->*174,[n>n, p=p, q=q][178, 149, 154, 159]
178->*178,[n>n, p=p, q=q][149, 154, 159, 174]
```

However the performance are poor: the analysis runs in 4.196 seconds instead of 0.65second with the first version.

4.3.10 Implementation details

The code consists in 1183 lines (36 kilobytes) of commented Objective Caml code. It reuses the common tools developed for the λ-calculus case. There is a separate module `Sct_coreml` for the $\mathcal{L}_{ml}$ part of the implementation which contains a class deriving from the main class defined in the module `Sct`.

This class contains the methods which constructs the judgement forms for all the rules of the approximation semantics.

A parser and a lexer for $\mathcal{L}_{ml}$ are also implemented.

Data structure

The following data structure is used to store nodes of the abstract tree during the parsing of the expression:

```
type ml_expr =
    MlVar of ident
  | Fun of ident * ml_expr
  | MlAppl of ml_expr * ml_expr
  | Let of (ident * (ident list) * ml_expr) list  * ml_expr
  | Letrec of (ident * (ident list) * ml_expr) list  * ml_expr
  | If of ml_expr * ml_expr * ml_expr
  | MlInt of int
  | AnyInt
  | MlBool of bool
  | EqTest of ml_expr * ml_expr
  | Pred
  | Succ
;;
```

The abstract tree is then converted into an array of subexpressions nodes. The type of the nodes is:

```
type ml_node =
    VarN of int
  | FunN of int * sub_expr
  | FunrecN of int * int * sub_expr
  | ApplN of sub_expr * sub_expr
  | LetN of int * sub_expr * sub_expr
  | IfN of sub_expr * sub_expr * sub_expr
  | MlIntN of int
  | AnyIntN
  | MlBoolN of bool
  | EqTestN of sub_expr * sub_expr
  | PredN of sub_expr
  | SuccN of sub_expr
```

The data structure for size-change graph is the same as the one defined in section 3.12.1.

The following types give the different flavors of judgement forms:

```
type ml_calltype = Operator | Operand | FuncApp | IfThenElse |
 IfCond | EqCond | PredSucc | LocalDef | FuncAppStar

type ml_evaltype = Normal

type ml_jftype = Call of ml_calltype | Evaluation of
ml_evaltype;;
```

The generated component in judgment forms is now a pair of two sets of size-change graph arcs, one for each of the two different instances of the size-change principle:

```
type ml_jfgen = scg_arc list * scg_arc list
```

The judgement form type is:

```
type ml_jf = ml_jftype * ml_jfgen
```

Parser

The parser developed with `ocamlyacc` and `ocamllex` recognizes an extended version of the $\mathcal{L}_{ml}$ syntax defined in 4.1.1. It accepts `let rec` structures and the special symbol $?^{\texttt{int}}$.

The following code is an example of a $\mathcal{L}_{ml}$ program which can be parsed:

```
let rec f x =
  if x = 0 then  0
  else
    succ (f (pred x))
in
  f ?
;;
```

Performance

Table 4.9 gives the times it takes to run the analysis on the different $\mathcal{L}_{ml}$ examples using the natively compiled version of the Objective Caml program (these figures have been measured on a laptop computer equipped with a P3 2.4Ghz processor, 512Mb of RAM and running Windows XP).

Examples given in the report are typeset in bold:

$\mathcal{L}_{ml}$ expression	Time
min.cml	0.02s
min.ycomb.cml	0.08s
min.chnum.cml	3.745s
counter.cml	0.00s
fibo.cml	0.06s
lucas.cml	0.65s
lucas.ycomb.cml	4.196s
ackerman.cml	0.05s
ackerman.chnum.cml	0.10s
loop.cml	0.00s
error.cml	0.00s
looperror.cml	0.00s

Table 4.9: Performance of “native” $\mathcal{L}_{ml}$ analysis

Chapter 5

Conclusion and further directions

5.1 Brief summary

The techniques proposed in [5] have been implemented to obtain a termination analyzer for higher-order λ-calculus expressions.

Then, since λ-calculus is more a theoretical tool than a programming language (it can be a hard task to program mathematical functions in λ-calculus) we concentrated on the adaptation of the principle to a basic higher-order functional language.

The right approach has consisted in following the steps of [5] to redefine the size-change principle to the particular case of our mini-language. This final algorithm detects size-change termination by doing simultaneous analysis of higher-order values and ground type values like integers.

This project shows that the size-change principle introduced in [6] is a powerful tool for termination analysis.

5.2 Personal enrichment

Throughout this project, I learnt how to do research in theoretical computer science, especially while I was working on the extension of the principle to a small functional language. The research I carried out required me to state and prove lemmas and theorems justifying the correctness of the algorithm.

5.3 Possible extension

The $\mathcal{L}_{ml}$ language has been highly restricted to facilitate the adaptation of the size-change principle.

A possible continuation of this project can consist in extending the algorithm to handle a more complex language. It is possible to use the language of [7] which features sequential composition, storage locations and references (as it is implemented in Objective Caml [4]).

One way to deal with locations could consist in extending environments to make them include locations in addition to free variables. The graph basis of size-change graphs would then contain free variables and storage locations.

Support for other language features could be also added: tuples with operators `fst` and `snd`, list and user defined data structures.

It could be interesting to add `while` and `for` loop structures. But then, the call semantics has to be defined very carefully in order to preserve the property that non-termination is characterized by the presence of infinite call sequences.

Appendix A

Proof of Lemma 4.3.1

The proof of this lemma follows the same steps as the proof of lemma 4 in [5]. However there are more difficulties to handle here. Indeed, in $\mathcal{L}_{ml}$ errors can occur during evaluation. Errors are therefore new causes of program termination. Consequently, we need to consider new cases in this proof and to use the rules of the error semantics (table 4.2).

$\boxed{\Leftarrow}$ We show that $\mathtt{e} : \rho \Downarrow$ or $\mathtt{e} \oslash$ implies that any call chain starting from $\mathtt{e}$ is finite.

We proceed by induction on the proof showing that $\mathtt{e} : \rho \Downarrow$ or $\mathtt{e} \oslash$: we assume that the property we want to show is true for any evaluation or error occurring before the rule concluding $\mathtt{e} : \rho \Downarrow$ or $\mathtt{e} \oslash$.

Consider the different cases:

- (ValueG) Then $\mathtt{e}$ is a canonical form. Hence there is no call chain starting from $\mathtt{e}$.
- (VarG) $\mathtt{e}$ is a variable, there is no call chain starting from it.
- (EqTrueG) $\mathtt{e} \equiv \mathtt{e}_1 = \mathtt{e}_2$. The only calls occurring at state $\mathtt{e} : \rho$ are $\mathtt{e} : \rho \rightarrow \mathtt{e}_1 : \rho$ from rule (EqCondTrueG) and $\mathtt{e} : \rho \rightarrow \mathtt{e}_2 : \rho$ from rule (EqCondFalseG). By using the induction hypothesis on the premise of rule (EqTrueG), we conclude that there is no infinite call sequence starting from $\mathtt{e}$.
- (EqFalseG) see (EqTrueG)
- (PredG), (SuccG), (LocalG), (ErrOp1) , (ErrOp2), (ErrOp3), (ErrEq1), (ErrEq2)

  Proof similar to (EqTrueG) : the only possible calls from state $\mathtt{e} : \rho$ are call of type $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho$ such that there is an evaluation $\mathtt{e}' : \rho \Downarrow v$ for some $\mathtt{v}$ in the premise of the rule.
- (ApplyG)

  There are two cases for $\mathtt{e}$:

  1. $\mathtt{e} \equiv \mathtt{if}\ \mathtt{e_c}\ \mathtt{then}\ \mathtt{e_1}\ \mathtt{else}\ \mathtt{e_2}$

     Consider the two possibilities for the first premise of (ApplyG):

  - Suppose that the first premise is $\mathtt{e} : \rho \underset{if}{\rightarrow} \mathtt{e_1} : \rho$, concluded by rule (IfTrueCallG).

    There are two possible calls from $\mathtt{e} : \rho$:

    * $\mathtt{e} : \rho \underset{if}{\rightarrow} \mathtt{e_c} : \rho$ In that case by induction on the first premise of rule (IfTrueCallG), we know that there is not infinite call from $\mathtt{e_c}$.
    * $\mathtt{e} : \rho \underset{if}{\rightarrow} \mathtt{e_1} : \rho$ In that case by induction on the second premise of rule (ApplyG) we know that there is not infinite call from $\mathtt{e_1}$.

  - Suppose that the first premise is $\mathtt{e} : \rho \underset{if}{\rightarrow} \mathtt{e_2} : \rho$, concluded by rule (IfFalseCallG). The proof is the same as the previous case.

  Any call form $\mathtt{e}$ leads to an expression from which there is no infinite call chain. Hence there is no infinite call chains from $\mathtt{e}$.

2. $\mathtt{e} \equiv \mathtt{e_1 e_2}$

   Suppose that the first premise of (ApplyG) is $\mathtt{e} : \rho \underset{c}{\rightarrow} \mathtt{e_0} : \rho_0[\mathtt{x} \mapsto \mathtt{v_2}]$ concluded using rule (CallG).

   From state $\mathtt{e} : \rho$, the only possible states which can be called are $e_1 : \rho$, $e_2 : \rho$, and $e_0 : \rho_0[x \mapsto v_2]$. But all these states are evaluated in the premise of the rule (CallG) or (ApplyG). Hence by induction, there is no infinite call chain starting from $\mathtt{e} : \rho$.

   The same reasoning is done when the rule (CallRecG) is used instead of (CallG).

- (ErrIf1) $\mathtt{e} \equiv \mathtt{if\ e_c\ then\ e_1\ else\ e_2}$

  Since evaluating $\mathtt{e}$ causes an error, there is no evaluation of $\mathtt{e}$ (see lemma 4.1.1). Therefore the rules (IfTrueCallG) and (IfFalseCallG) cannot be used.

  Hence the only possible call from $\mathtt{e}$ is $\mathtt{e} \rightarrow \mathtt{e_c}$ concluded by the rule (IfCondCallG).

  Applying the induction on the premise $\mathtt{e_c} \oslash$ of rule (ErrIf1) we obtain the desired result.

- (ErrIf2) $\mathtt{e} \equiv \mathtt{if\ e_c\ then\ e_1\ else\ e_2}$

  The only possible call from $\mathtt{e}$ are $\mathtt{e} \rightarrow \mathtt{e_c}$ concluded by the rule (IfCondCallG) and $\mathtt{e} \rightarrow \mathtt{e_1}$ concluded by the rule (IfTrueCallG).

  In both case, by applying the induction on one of the premise of (ErrIf2) we obtain the desired result.

- (ErrIf3) Same as (ErrIf2).

- (ErrLocalDef1), (ErrLocalDef2), (ErrApp1), (ErrApp2), (ErrApp3)

  Similar to cases (ErrIf1) and (ErrIf2).

$\boxed{\Rightarrow}$ Suppose that $\mathtt{P} : [] \rightarrow^* \mathtt{e} : \rho$ and all call chains from $\mathtt{e} : \rho$ are finite.

We prove that $\mathtt{e} : \rho \Downarrow$ by induction on the length n of the longest call chain from $\mathtt{e} : \rho$.

If $n = 0$ then there is no possible call from $\mathtt{e} : \rho$. Therefore $\mathtt{e}$ must be in canonical form. Hence by rule (ValueG) : $\mathtt{e} : \rho \Downarrow \mathtt{e} : \rho$.

If $n > 0$ then the only possible cases are:

- $\mathtt{e} \equiv \mathtt{e}_1\mathtt{e}_2$

  Since the program is well-typed the expression $\mathtt{e}_1$ in the application $\mathtt{e}_1\mathtt{e}_2$ must be a function. We assume that $\mathtt{e}_1 \equiv \mathtt{fun\ (x{:}ty){-}{>}e}_0 : \rho_0$ (the proof is similar if $\mathtt{e}_1 \equiv \mathtt{fun\ f{=}(x{:}ty){-}{>}e}_0 : \rho_0$).

  By rule (OperatorG) there is a call $\mathtt{e}_1\mathtt{e}_2 \rightarrow \mathtt{e}_1$. The longest call chain from $\mathtt{e}_1$ is therefore shorter than the longest call chain from $\mathtt{e}$. Consequently, by the induction hypothesis:

  - either $\mathtt{e}_1 \oslash$ and in that case, by the rule (ErrApp1) of table 4.2 $\mathtt{e} \oslash$
  - either there exist $\mathtt{v}_1$ such that $\mathtt{e}_1 : \rho \Downarrow \mathtt{v}_1$. By rule (OperandG) we then conclude $\mathtt{e}_1\mathtt{e}_2 : \rho \rightarrow \mathtt{e}_2 : \rho$. Again by induction:
    - either $\mathtt{e}_2 \oslash$ and in that case by the rule (ErrApp2) of table 4.2 we conclude $\mathtt{e} \oslash$.
    - either there exist $\mathtt{v}_2$ such that $\mathtt{e}_2 : \rho \Downarrow \mathtt{v}_2$ and then we can apply the rule (CallG) to conclude that $\mathtt{e}_1\mathtt{e}_2 : \rho \rightarrow \mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto \mathtt{v}_2]$.

      We now use a third time the induction hypothesis:
      - either $\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto \mathtt{v}_2] \oslash$ and in that case, by the rule (ErrApp3) we conclude $\mathtt{e} \oslash$.
      - either $\mathtt{e}_0 : \rho_0[\mathtt{x} \mapsto \mathtt{v}_2] \Downarrow \mathtt{v}$ for some $\mathtt{v}$. This gives use all the premises for the rule (ApplyG), thus $\mathtt{e} \equiv \mathtt{e}_1\mathtt{e}_2 : \rho \Downarrow \mathtt{v}$.

- $\mathtt{e} \equiv \mathtt{if\ e}_c\ \mathtt{then\ e}_1\ \mathtt{else\ e}_2$

  The proof follow exactly the same outline as for the previous case. The only differences are the rules used: (IfCondCallG), (IfTrueCallG), (IfFalseCallG), (ApplyG) and the error rules (ErrIf1), (ErrIf2) and (ErrIf3).

- $\mathtt{e} \equiv \mathtt{e}_1 = \mathtt{e}_2$

  Similar proof using rules (EqCondTrueG), (EqCondFalseG), (EqTrueG), (EqFalseG) and the error rules (ErrEq1) and (ErrEq2).

- $\mathtt{e} \equiv \mathtt{pred\ e}'$

  Similar proof using rules (PredCallG) and (PredG) and the error rules (ErrOp1) and (ErrOp2).

- $\mathtt{e} \equiv \mathtt{succ\ e}'$

  Similar proof using rules (SuccCallG) and (SuccG) and the error rules (ErrOp3).

- $\mathtt{e} \equiv \mathtt{let\ x\ =\ e_1\ in\ e_2}$

  Similar proof using rules (LocalDefCallG), (LocalBodyCallG), (LocalG) and the error rules (ErrLocalDef1), (ErrLocalDef2).

■

Appendix B

Proof of Theorem 4.3.2

We give a separate proof for the two principles:

1. Higher-order graphs (SCP^+):

   These are basically the same graphs as in the λ-calculus case (see theorem 2 of [5] for a complete proof). The only differences come from the new structures introduced in the syntax of $\mathcal{L}_{ml}$ (`let, let rec, if`) and the integers and boolean operators (`succ, pred`). The new rules are (EqTrueG), (EqFalseG), (PredG), (SuccG), (LocalG), (IfCondCallG), (IfTrueCallG), (IfFalseCallG), (PredCallG), (SuccCallG),(EqCondTrueG), (EqCondFalseG), (CallRecG),(LocalDefCallG), (LocalBodyCallG).

   - The proof for the rule (LocalG) has been given in remark 4.3.3.
   - For (EqTrueG), (EqFalseG), (PredG), (SuccG), the generated graphs are the empty set $\emptyset$ which is always safe.
   - For (IfCondCallG), (IfTrueCallG), (IfFalseCallG), (PredCallG), (SuccCallG), (EqCondTrueG), (EqCondFalseG), (CallRecG), (LocalDefCallG), (LocalBodyCallG)

     the jugdement forms are all of type $\mathtt{e} \rightarrow \mathtt{e_{sub}}, \_ \; | id^{\downarrow}_{\mathtt{e_{sub}}}$ where $\mathtt{e_{sub}}$ is a subexpression of $\mathtt{e}$. Therefore these graphs are safe.

2. Ground-type graphs (SCP^0):

   We show the safety property by induction on the proof of $s \Downarrow s', G|\_$ or $s \rightarrow s', G|\_$.

   We just give a proof for the rule (PredG):

   - (EqTrueG), (EqFalseG)

     In these two rules, the generated graph has no arcs therefore it is safe.
   - (ValueG) $id^{=}_{\mathtt{e}}$ is safe for $(\mathtt{v}, \mathtt{v})$. This is immediate by definition of arc safety.
   - (VarG) Arc $\mathtt{x} \overset{=}{\rightarrow} \bullet$ is safe because $\overline{x : \rho}(x) = \rho(x) = \overline{\rho(x)}(\bullet)$.

     Consider the arc $\bullet \overset{=}{\rightarrow} \bullet$.

    - Suppose the type of $\mathtt{x}$ is a ground type, *int* for instance. The only possible evaluation of $\mathtt{x} : \rho$ is $\mathtt{x} : \rho \Downarrow \mathtt{n} : []$ where $\mathtt{n} \in \mathbb{N}$.
      But $\Downarrow$ is deterministic therefore $\rho(\mathtt{x}) = \mathtt{n} : []$.
      Using the rule (ValueG) we have $\mathtt{n} : [] \Downarrow \mathtt{n} : []$. Consequently $\mathtt{x} : \rho$ and $\mathtt{n} : []$ evaluate to the same number therefore $\mathtt{x} : \rho =_0 \mathtt{n} : []$. Hence we have:
      $\overline{x : \rho}(\bullet) = x : \rho \quad =_0 \quad \mathtt{n} : [] = \rho(\mathtt{x}) = \overline{\rho(\mathtt{x})}(\bullet)$
    - if $\mathtt{x}$ is a higher-order value then the type of $\rho(\mathtt{x})$ is also a higher-order function, therefore they are considered to be equal (relatively to definition 4.3.1).

  The arc $\bullet \overset{=}{\rightarrow} \bullet$ is therefore safe.

- (OperatorG), (OperandG), (LocalDefCallG), (PredCallG), (EqCondTrueG), (EqCondFalseG) and (IfCondCallG)

  All these rules are axioms. The conclusion is a judgment form of type $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho$ with the generated graph $idv_{\mathtt{e}}$. Consider an arc $\mathtt{x} \overset{=}{\rightarrow} \mathtt{x} \in \mathtt{idv}_{\mathtt{e}}$ where $\mathtt{x} \in fv(\mathtt{e})$ then

  $$\overline{\mathtt{e} : \rho}(x) = \rho(x) = \overline{\mathtt{e}' : \rho}(x)$$

- (IfTrueCallG)

  The conclusion judgement form is $\underbrace{\texttt{if e then e}_1\texttt{ else e}_1}_{\mathtt{e}_{if}} : \rho \underset{if}{\rightarrow} \mathtt{e}_1 : \rho$ The generated graph is $id^{=}_{\mathtt{e}_1} = idv_{\mathtt{e}_1} \cup \{\bullet \overset{=}{\rightarrow} \bullet\}$. The arcs of $idv_{\mathtt{e}_1}$ are safe for the same reason as in the previous case.

  Consider the arc $\bullet \overset{=}{\rightarrow} \bullet$. There are two possibilities:

    - $\mathtt{e}_{if}$ is a higher-order value: $\Gamma \vdash \mathtt{e}_{if} : ty_1 \rightarrow ty_2$ then $\Gamma \vdash \mathtt{e}_1 : ty_1 \rightarrow ty_2$ and

      $$\overline{\mathtt{e}_{if} : \rho}(\bullet) = \mathtt{e}_{if} : \rho \quad \succeq_0 \quad \mathtt{e}_1 : \rho = \overline{\mathtt{e}_1 : \rho}(\bullet)$$

    - $\mathtt{e}_{if}$ is a ground type value. For instance $\Gamma \vdash \mathtt{e}_{\mathtt{if}} : int$ then $\Gamma \vdash \mathtt{e}_1 : int$.
      Suppose that $\mathtt{e}_1 \Downarrow \mathtt{v}$ then, using the conclusion of the rule (IfTrueCallG) we can apply the rule (ApplyG):

      $$(ApplyG)\frac{\mathtt{e}_{if} : \rho \underset{if}{\rightarrow} \mathtt{e}_1 : \rho', id^{=}_{\mathtt{e}_1}|\_ \qquad \mathtt{e}_1 : \rho' \Downarrow \mathtt{v} : \_ \, , \_ \; |\_}{\mathtt{e}_{if} : \rho \Downarrow \mathtt{v}, \_ \; |\_}$$

      Since $\mathtt{e}_{if}$ and $\mathtt{e}_1$ both evaluate to $\mathtt{v}$ we have $\mathtt{e}_{if} : \rho =_0 \mathtt{e}_1 : \rho$. Consequently:

      $$\overline{\mathtt{e}_{if} : \rho}(\bullet) = \mathtt{e}_{if} : \rho \quad =_0 \quad \mathtt{e}_1 : \rho = \overline{\mathtt{e}_1 : \rho}(\bullet)$$

      Since $\Downarrow$ is deterministic, $\mathtt{v}$ is the only possible evaluation of $\mathtt{e}_{if}$. Therefore the arc $\bullet \overset{=}{\rightarrow} \bullet$ is safe for the call $\mathtt{e}_{if} : \rho \ \rightarrow \ \mathtt{e}_1 : \rho$.

- (IfFalseCallG) same as (IfTrueCallG)

- (SuccCallG)

  The graph generated is $id^{\downarrow}_{\mathtt{e}_2} = idv_{\mathtt{e}_2} \cup \{\bullet \overset{\downarrow}{\rightarrow} \bullet\}$. The arcs of $idv_{\mathtt{e}_2}$ are safe (same explanation as for rule (OperatorG)).

  The expression `succ e` and `e` are of type `int`.

  Suppose `e` evaluates to `n` then by using the rule (SuccG) we have: $\mathtt{succ\ e} \Downarrow \mathtt{n} + \mathtt{1}$. These are the only possible evaluation since $\Downarrow$ is deterministic.

  Since $n + 1 > n$ we have:

  $$\overline{\mathtt{succ\ e} : \rho}(\bullet) = \mathtt{succ\ e} : \rho \qquad \succ_0 \qquad \mathtt{e} : \rho = \overline{\mathtt{e} : \rho}(\bullet)$$

  The arc $\{\bullet \overset{\downarrow}{\rightarrow} \bullet\}$ is therefore safe.

- (LocalBodyCallG)

  The conclusion judgement form is $\underbrace{\mathtt{let\ x\ =\ e_1\ in\ e_2}}_{\mathtt{e}_{let}} : \rho \rightarrow \mathtt{e}_2 : \rho[\mathtt{x} \mapsto \mathtt{v}_1]$

  The arcs $\mathtt{y} \overset{=}{\rightarrow} \mathtt{y}$ with $\mathtt{x} \neq \mathtt{y}$ are safe. The proof is the same as in rule (OperatorG). Note that the arc $\mathtt{x} \overset{=}{\rightarrow} \mathtt{x}$ has not been included in the graph. The reason is that the possibly free occurrences of `x` in $\mathtt{e}_{let}$ and the free occurrences of `x` in $\mathtt{e}_2$ refer to two different variables (bound in two different places in the program).

  Consider the arc $\bullet \overset{=}{\rightarrow} \bullet$. There are two possibilities:

  - $\mathtt{e}_{let}$ is a higher-order value: $\Gamma \vdash \mathtt{e}_{let} : ty_1 \rightarrow ty_2$ then $\Gamma \vdash \mathtt{e}_2 : ty_1 \rightarrow ty_2$ and

    $$\overline{\mathtt{e}_{let} : \rho}(\bullet) = \mathtt{e}_{let} : \rho \quad \succeq_0 \quad \mathtt{e}_2 : \rho = \overline{\mathtt{e}_2 : \rho}(\bullet)$$

  - $\mathtt{e}_{let}$ is a ground type value. For instance $\Gamma \vdash \mathtt{e}_{\mathtt{let}} : int$ then $\Gamma \vdash \mathtt{e}_2 : int$. Suppose that $\mathtt{e}_2 : \rho[\mathtt{x} \mapsto \mathtt{v}_1] \Downarrow \mathtt{v}$. Using this evaluation in conjunctions with the premise of rule (LocalBodyCallG) we can apply the rule (LocalG):

    $$(LocalG)\frac{\mathtt{e}_1 : \rho \Downarrow \mathtt{v}_1, \_ \ |\_ \qquad \mathtt{e}_2 : \rho[\mathtt{x} \mapsto \mathtt{v}_1] \Downarrow \mathtt{v}_2, \_ \ |\_}{\mathtt{e}_{let} : \rho \Downarrow \mathtt{v}_2, \_ \ |\_}$$

    Since $\mathtt{e}_{let}$ and $\mathtt{e}_2$ both evaluate to $\mathtt{v}_2$ we have:

    $$\overline{\mathtt{e}_{let} : \rho}(\bullet) = \mathtt{e}_{let} : \rho \quad =_0 \quad \mathtt{e}_2 : \rho = \overline{\mathtt{e}_2 : \rho}(\bullet)$$

    Since $\Downarrow$ is deterministic, $\mathtt{v}_2$ is the only possible evaluation of $\mathtt{e}_{let}$. Therefore the arc $\bullet \overset{=}{\rightarrow} \bullet$ is safe for the call $\mathtt{e}_{let} : \rho \ \rightarrow \ \mathtt{e}_2 : \rho[x \mapsto v_1]$.

- (PredG)

  The conclusion of the rule (PredG) tells us that $\mathtt{pred\ e} \Downarrow \mathtt{n} - \mathtt{1}$. and by the rule (ValueG), `n-1` $\Downarrow$ `n-1`.

  By the determinism of the relation $\Downarrow$, these are the only possible evaluations for `pred e` and `n`. Hence:

  $$\overline{s}(\bullet) = \mathtt{pred\ e} \succeq_0 \mathtt{n} - \mathtt{1} = \overline{s'}(\bullet)$$

This justifies the safety of the arc $\bullet \xrightarrow{=} \bullet$.

Now suppose that $\mathtt{x} \rightarrow \bullet \in G$ then

$$\overline{\mathtt{e} : \rho}(\mathtt{x}) = \rho(\mathtt{x}) \qquad \text{and} \qquad \overline{\mathtt{n} : []}(\bullet) = \mathtt{n} : []$$

Suppose that $\rho(\mathtt{x}) \Downarrow q$. By rule (ValueG) we have $n : [] \Downarrow n : []$ therefore by induction $q \geq n$.

Since $\overline{\mathtt{pred\ e} : \rho}(\mathtt{x}) = \rho(\mathtt{x})$, we have $\overline{\mathtt{pred\ e} : \rho}(\mathtt{x}) \Downarrow q$. Moreover $n - 1 : [] \Downarrow n - 1 : []$.

By determinism, these are the only possible evaluations. Since $q \geq n > n - 1$, the arc $\mathtt{x} \xrightarrow{\downarrow} \bullet$ is safe.

- (SuccG) the proof is symmetrical to the proof of (PredG).
- (LocalG) see remark 4.3.3.
- (ApplyG) This is due to the fact that the safety property is preserved by graph composition. This is shown in proof of theorem 2 in the Appendix of [5].
- (CallG) See proof for rule (CallRecG) below.
- (CallRecG)
  - Arc $\bullet \xrightarrow{=} \bullet$.

    The proof is similar to the case of rule (IfFalseCallG), there are two cases: $\mathtt{e_1 e_2}$ is a higher-order value or a ground type value. Expression $\mathtt{e_0}$ and $\mathtt{e_1 e_2}$ are of the same type. Suppose that it is a higher-order value then the by definition 4.3.1 they are equal. Now suppose it is a ground type value and assume that $\mathtt{e_0} : \rho_0[\mathtt{x} \mapsto \mathtt{v_2}, \mathtt{f} \mapsto \mathtt{v}] \Downarrow \mathtt{v}'$ then by applying the rule (ApplyG) we have $\mathtt{e_1 e_2} \Downarrow \mathtt{v}'$. Hence the states $\mathtt{e_1 e_2} : \rho$ and $\mathtt{e_0} : \rho_0[\mathtt{x} \mapsto \mathtt{v_2}, \mathtt{f} \mapsto \mathtt{v}]$ evaluate to the same ground type value. This justifies the safety of arc $\bullet \xrightarrow{=} \bullet$.
  - Arc $y \xrightarrow{r} z$ where $y \xrightarrow{r} z \in G_1$.

    $\mathtt{x}$ and $\mathtt{f}$ are not free in $\mathtt{v}$ therefore $z \notin \{\mathtt{x}, \mathtt{f}\}$

    By induction G_1 is safe for $(\mathtt{e_1} : \rho, \mathtt{fun\ f=(x:ty)\text{->}e_0} : \rho_0)$, therefore $\rho(y) \succeq \rho_0(z)$ Thus:

    $$\overline{\mathtt{e_1 e_2} : \rho}(y) = \rho(y) \succeq \rho_0(z) = \overline{\mathtt{e_0} : \rho_0[\mathtt{x} \mapsto \mathtt{v_2}, \mathtt{f} \mapsto \mathtt{v}]}(z)$$

    The last equality is justified by the fact that $z \notin \{\mathtt{x}, \mathtt{f}\}$.

    Hence the arc $y \xrightarrow{r} z$ is safe.
  - Arc $y \xrightarrow{r} x$ where $y \xrightarrow{r} \bullet \in G_2$.

    By safety of G_2, we have $\rho(y) \succeq \mathtt{v_2}$. Thus:

    $$\overline{\mathtt{e_1 e_2} : \rho}(y) = \rho(y) \succeq \mathtt{v_2} = \overline{\mathtt{e_0} : \rho_0[\mathtt{x} \mapsto \mathtt{v_2}, \mathtt{f} \mapsto \mathtt{v}]}(x)$$

    Hence the arc $y \xrightarrow{r} x$ is safe.

■

Appendix C

Proof of Lemma 4.3.3

This proof follows the same directions as its counterpart in the λ-calculus case (see proof of lemma 10 and lemma 11 of [5]).

We proceed by induction on the size of the proof concluding $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho'$, $\mathtt{e} : \rho \Downarrow \mathtt{v} : \rho'$, $\mathtt{e} : \rho \Downarrow \mathtt{n} : \rho'$ or $\mathtt{e} : \rho \Downarrow \mathtt{b} : \rho'$: let us assume that the lemma holds for all calls and evaluations performed in the computation before the concluding rule is applied.

We now proceed by cases analysis on the rule applied to conclude $\mathtt{e} : \rho \rightarrow \mathtt{e}' : \rho'$, $\mathtt{e} : \rho \Downarrow \mathtt{v} : \rho'$, $\mathtt{e} : \rho \Downarrow \mathtt{n} : \rho'$ or $\mathtt{e} : \rho \Downarrow \mathtt{b} : \rho'$.

For each case, we show that some rule in the approximation semantics can be applied to give the corresponding conclusion:

- Base cases: Rule (ValueG) is modeled by rules (ValueAG), (ValueAG') and (ValueAG") in the approximation semantics. Rules (EqTrueG) and (EqFalseG) are modeled by the rule (EqAG). Rules (IfCondCallG), (IfTrueCallG), (IfFalseCallG), (EqCondTrueG), (EqCondFalseG), (PredCallG), (SuccCallG), (LocalDefCallG), (LocalBodyCallG), (OperatorG), (OperandG) are modeled by the corresponding rules with the suffix "AG" instead of "G" in the approximation semantics.

  For all these cases, the approximation rules are the same as their exact semantics counterparts after removal of the environment component and removal of a premise for (OperandA), (LocalBodyCallA), (IfTrueCallA), (IfFalseCallG), (EqTrueG) and (EqFalseG).

  The graph generated are the same in the approximation rules (graphs generation is not influenced by the environment).

- (VarG): Suppose the variable is $\mathtt{z}$. We have $\mathtt{P} : [] \rightarrow^* \mathtt{z} : \rho$ and $\mathtt{z} : \rho \Downarrow \rho(\mathtt{z})$ where $\rho(z) = e' : \rho'$.

  The call sequence $\mathtt{P} : [] \rightarrow^* \mathtt{z} : \rho$ starts with an empty environment and finishes with the environment ρ which is not empty (since $\mathtt{z} \in \mathtt{dom}(\rho)$).

  The only way $\mathtt{z}$ can be bound is by the use of rules (CallG), (CallRecG) or (LocalBodyCallG), the only rules which extend the environment.

  - Suppose the variable $\texttt{z}$ corresponds to the variable $\texttt{f}$ bound in the rule (CallRecG). Since the first premise of (CallRecG) is $\texttt{e}_1 : \rho \Downarrow \overbrace{\texttt{fun f} = (\texttt{x} : \texttt{ty})\texttt{->}\texttt{e}_0 : \rho_0}^{\texttt{v}}$ then $\texttt{fun f} = (\texttt{x} : \texttt{ty})\texttt{->}\texttt{e}_0$ must be a subexpression of $\texttt{P}$.

    Moreover, in the conclusion of the rule, the variable $\texttt{z} = \texttt{f}$ is bound to the value $\texttt{v}$. Therefore $\rho(\texttt{z}) = \texttt{v} \equiv \texttt{fun f} = (\texttt{x} : \texttt{ty})\texttt{->}\texttt{e}_0 : \rho_0$.

    Hence we can apply the rule (VarRecAG) to obtain the required judgment form.

  - Suppose the variable $\texttt{z}$ correspond to the variable $\texttt{x}$ bound in the rule (CallG) or (CallRecG).

    These rules require that $\texttt{e}_1\texttt{e}_2 \in \textit{subexp}(\texttt{P})$. By applying the induction hypothesis to the premises of rule (CallG) or (CallRecG) we obtain the premises of rule (VarAG). Thus we can conclude the required judgement form.

  - Suppose the variable $\texttt{z}$ correspond to the variable $\texttt{x}$ bound in the rule (LocalG).

    This rule requires that $\texttt{let x} = \texttt{e}_1 \texttt{ in } \texttt{e}_2 \in \textit{subexp}(\texttt{P})$. By applying the induction hypothesis to the first premises of rule (LocalG) we obtain the second premise of rule (VarLetAG). Thus we can conclude the required judgement form.

- (PredG), (SuccG): By applying the induction hypothesis on the first premise of (PredG) and (SuccG) we have:

$$\texttt{e} : \rho \Downarrow ?^{\texttt{int}}, G^0 | G^+$$

  By applying the rule (PredAG) and (SuccAG) we obtain the required conclusion with the same generated graph as in (PredG) and (SuccG).

- (LocalG), (ApplyG), (CallG) and (CallRecG): By applying the induction hypothesis on the premises of these rules we obtain the premises for the corresponding rules (LocalAG), (ApplyAG), (CallAG) and (CallRecAG). Thus we can conclude the required result.

■

Bibliography

[1] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[2] Hendrik Pieter Barendregt. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[3] William Blum. Implementation sources and report of this MSc thesis, 2004. `http://www.famille-blum.org/~william/mscthesis/`.

[4] INRIA. Objective Caml Programming language, 2003. `http://caml.inria.fr/`.

[5] N.D Jones and Nina Bohr. Termination analysis of the untyped λ-calculus. In *Rewriting Techniques and Applications. Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 2004.

[6] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, january 2001.

[7] Andrew M. Pitts. Operational semantics and program equivalence. volume 2395:378–412, 2002.

[8] G.D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1, 1975.

[9] Wikipedia. Wikipedia, the free encyclopedia., 2004. `http://en.wikipedia.org/`.