Technical Report DIKU-TR-99/6
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø
DENMARK

Termination Analysis for Offline Partial Evaluation of a Higher Order Functional Language

Peter Holst Andersen
Department of Computer Science,
University of Copenhagen,
txix@diku.dk

August 29, 1996

Abstract

One of the remaining problems on the path towards fully automatic partial evaluation is ensuring termination of the specialization phase. In [Holst, 1991] we gave a termination analysis which could be applied to partial evaluation of first-order strict languages, using a new result about inductive arguments (loosely: if whenever something grows, something gets smaller then the program will only enter finitely many different states). In this thesis we extend this work to cover higher-order functional languages. We take an operational approach to the problem and consider the closure representation of higher-order functions to perform a combined data- and control-dependency analysis. The result of this analysis is then used, as in the first-order case, to decide which arguments need to be dynamic to guarantee termination of partial evaluation of the analysed program. The new methods have been tested on a variety of programs, and will be incorporated in a future release of the Similix partial evaluator for Scheme.

Preface

This report is my master thesis ("speciale") and accounts for 10 written credits ("skriftige punkter") in the partial fulfillment of the requirements for a Danish cand. scient. degree ("kandidatgrad").

The work presented here is joint with Carsten Kehler Holst and has partly been published in [Andersen and Holst, 1996]. Neil D. Jones has been the supervisor of the project.

Prerequisites

This thesis requires knowledge of a functional language, closures as an implementation technique, of abstract interpretation corresponding to, for example, [Jones and Nielson, 1994], and of partial evaluation corresponding to, for example, [Jones et al., 1993, Part II],

Acknowledgments

Thanks to Neil D. Jones and Arne Glenstrup for much needed sparring and helpful comments.

Thanks to Mark P. Jones for providing Gofer, without which it would have been impossible to conduct quite as many experiments as we did.

Contents

Chapter 1

Introduction

Over the last thirty years software systems have become increasing bigger and more complex. However, the tools and techniques for software development have not been able to keep up with the demand, thus resulting in the so-called *software crisis*. Fundamentally, software engineering is still a handcraft, where almost every line of code is written by hand, and where tremendous manpower is spent debugging and maintaining code.

Partial evaluation or *automatic program specialization* is a tool that can help to *automate* part of the software engineering process, thereby reducing the time-to-market and the overall cost.

For partial evaluation to be successful as an automatic tool for non-specialist users, the user must be able to use it as a "black box" similar to the way optimizing compilers are used today. However, this is complicated by the fact that almost all of today's partial evaluators have unsafe termination properties (i.e., they are not guaranteed to terminate).

In this thesis we show how termination can be achieved even for partial evaluation of higher-order languages. First, we give a brief introduction to partial evaluation.

1.1 Partial Evaluation

Efficiency Versus Generality. In software development it is often the case that one has to solve several similar problems. One way to do it is to implement one *specialized* program for each problem, but this has the disadvantage of developing, debugging, and maintaining several programs.

Alternatively, one can write one *general* highly parameterized program, that solves all the problems. This saves a lot of time, since there is only one program to develop, debug and maintain. However, a general program is rarely as *efficient* as a program written only to solve one task.

This is where partial evaluation comes in. Partial evaluation provides the best of both worlds: it allows the programmer to work at a higher level of abstraction without having to worry about efficiency. The programmer can develop a general program and use the partial evaluator to *automatically* generate efficient versions of it, each specialized to a given subproblem.

Modularity. When developing software systems it is convenient and sometimes necessary to divide it into separate modules. This make maintenance easier, enables reuse of the code, and allows several people to work on the same system. However, loss of efficiency cannot be entirely avoided. Some modules are perhaps overly general for a specific application, or they may contain interface code checking for various conditions. Partial evaluation can help by removing superfluous interface code, and by specializing overly general modules.

How Is Partial Evaluation Done? A partial evaluator is a program, that given another program `p` and some of its input `s` (the *static* input), produces a *residual* program $\mathtt{p_s}$, which is a version of `p` specialized with respect to the input `s`. When the residual program is executed on the rest of the input `d` (the *dynamic* input) it will produce the same result as the original program would if executed on the entire input `(s,d)`.

The time when the residual program is generated is called *specialization time* or *mix time*, and the time when the program is executed is called *run time* as usual.

Example 1.1 Consider the following function for appending two lists:

$$append(xs, ys) = if\ \ null?\ xs\ then\ ys\ else\ cons(hd\ xs, append(tl\ xs, ys))$$

Suppose we know the value of *xs* to be the list (1 2 3), then we can generate a specialized version of *append*, which is much faster than the original:

$$append_{123}(ys) = cons(1, cons(2, cons(3, ys)))$$

□

A partial evaluator will execute those parts of the program that solely depends on the static input and generate code for the rest. Since it does a *mixture* of interpretation and compilation it is often called *mix*.

A partial evaluator uses standard optimization techniques: it pre-computes expressions, that solely depend on the static input (*constant folding*), it reduces expressions that depend on the dynamic input, unfolds function calls, and generates specialized versions of functions (*procedure cloning*). It also does *constant propagation* to make the known values available throughout the program.

In the example above the test on *xs* in the conditional was evaluated to decide which branch to choose, and the recursive calls were unfolded.

Online and Offline. Basically, there are two flavours of partial evaluation, *online* and *offline*. The differences between the two lies in *when* the decision to evaluate or to generate code for an expression is taken. In online partial evaluation the decision is taken *during* specialization, whereas in offline partial evaluation it is taken in a prephase (the *binding-time analysis* or BTA for short). We shall briefly describe how offline partial evaluation works.

The traditional pipeline for an offline partial evaluator is shown in Figure 1. The binding-time analysis takes the subject program, the name of the function to specialize (the *goal* function)[1], and the initial *binding-time pattern* as input. The binding-time pattern specifies which of the goal function's parameters are static and which are dynamic, but the actual values of the static data are not given at this point.

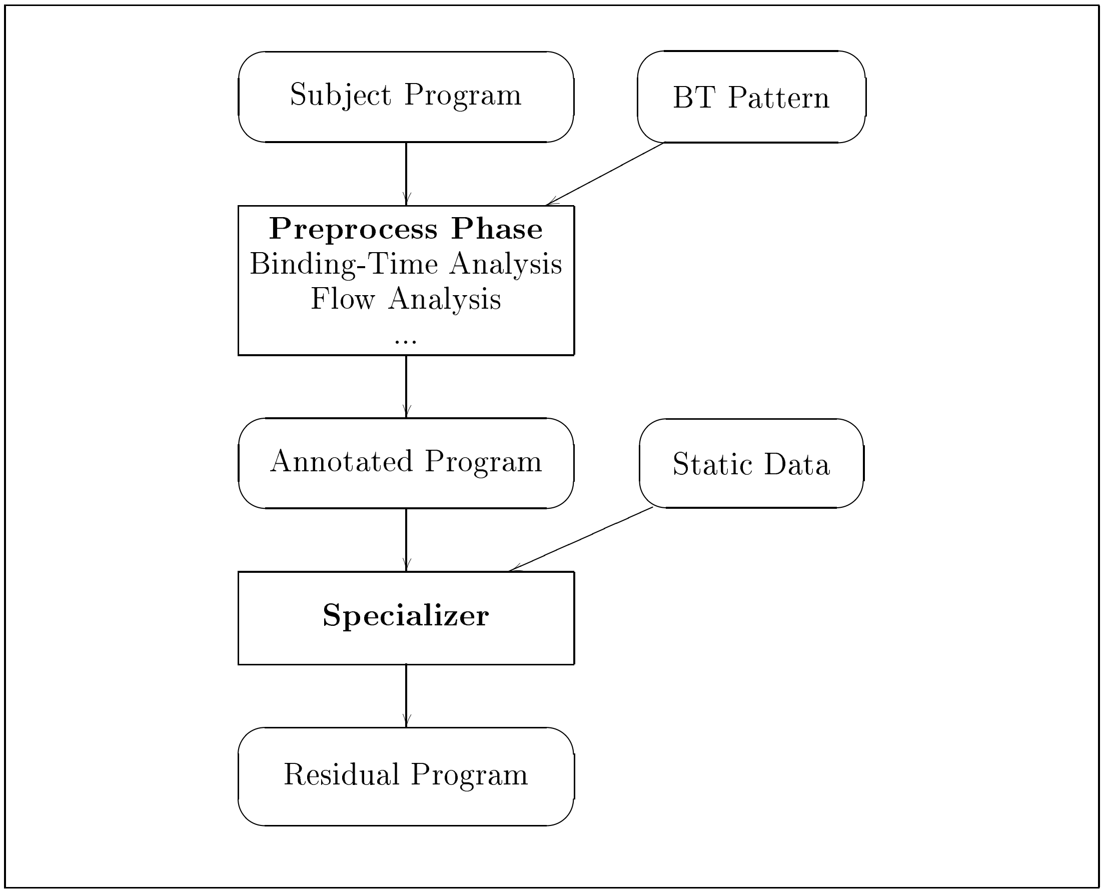

Figure 1: Offline Partial Evaluation

The job of the binding-time analysis is figure out which parts of the program the specializer shall evaluate and which it shall generate code for. Basically, the binding-time analysis is a dependency analysis that detects which parts of the

[1]The goal function is not shown in Figure 1.

program depend on the dynamic input and marks these as *residual* (= parts to generate code for). Usually it is necessary to perform other analyses to aid the binding-time analysis; for example, a *flow analysis* to trace the flow of data structures or higher-order values, or an *evaluation-order analysis* to ensure that side effects are executed in the right order.

The output of the binding-time analysis is an *annotated* program. The annotations indicate which parts shall be evaluated and which shall be residualized. Given the values of the static input, the specializer follows the annotations and produces a residual program, which can be run on the dynamic part of the input.

Example 1.2 Below is an annotated version of the append function where *xs* is static and *ys* is dynamic:

$$append(xs, \underline{ys}) = if\ \ null?\ xs\ then\ \underline{ys}\ else\ \underline{cons}(lift(hd\ xs), append(tl\ xs, \underline{ys}))$$

The dynamic parts are marked by underlining. The *cons* operation is dynamic since it depends on *ys*. □

1.2 This Thesis

The reason that all of today's offline partial evaluators have unsafe termination properties, is an apparently unavoidable conflict between two desirable properties: A partial evaluator should *terminate* on every program `p` and static data `s`, and it should be *computationally complete*, meaning that it should compute *all* of `p`'s actions that depend only on `s`.

Many successful partial evaluators have prioritized computational completeness over termination (e.g., Similix [Sim, 1995, Bondorf and Jørgensen, 1993], Schism [Consel, 1993], and C-Mix [Andersen, 1993, Andersen, 1994]). In this thesis we show that termination can be achieved even for higher-order languages with an acceptable loss of computational completeness.

The sources of nontermination are *infinite specialization* (an attempt to create an infinitely large specialized program) and *completely static loops* (loops in `p` that depend only on the static input `s`). We develop an analysis[2], which changes some of the binding times in the program from static to dynamic, in such a way that partial evaluation of the program only enters finitely many different configurations. This, together with memoization, guarantees termination of partial evaluation.

The analysis is based on the same foundation as Holst's analysis for a first-order language [Holst, 1991], which works as follows: First an approximation of the program's control- and data-flow during partial evaluation is computed. The approximation gives information about which variables depend on which,

[2]for a higher-order untyped strict functional language (e.g., Scheme)

and whether they grow or get smaller along the possible evaluation paths. The gathered information is then used to generalize[3] variables for which upper bounds cannot be guaranteed. To adopt the approach from [Holst, 1991] to a higher-order language (or any other language for that matter), "all" we have to do is to devise an analysis that collects an approximation of the program's control- and data-flow, and then apply the main result from [Holst, 1991]. The problem is to collect an interesting approximation.

Termination of computationally complete partial evaluation in general is undecidable (an easy consequence of Rice's Theorem [Rice, 1953]). Therefore the analysis will make a *safe approximation*, that is guaranteed to detect all infinite loops, but may classify some loops as infinite even though they will always terminate. This corresponds to the safety condition found in other abstract interpretations, e.g., strictness analysis. However, in strictness analysis even a little information can be useful, whereas our analysis is uninteresting unless it solves the problem for a large class of programs. Therefore, the development of the analysis has mainly been driven by experimentation with small programs containing non-trivial recursion and usages of higher-order functions. This approach was motivated by the belief that, if the analysis can handle these small, but complex programs satisfactory, then it can handle real programs as well. Experiments show that the analysis is strong enough to detect that partial evaluation of non-trivial interpreters using higher-order features will terminate, and at the same time all interpretive overhead will be specialized away.

A related problem, which we do not address in this thesis, is *abnormal termination* of partial evaluation (errors occurring while executing static code). The problem has been fixed in Similix [Sim, 1995], which generates code to produce the error at run time when encountering an erroneous static expression.

Outline of this Thesis. Chapter 2 presents the subject language and defines what it means for a program to be well-annotated. Chapter 3 defines quasi-termination and gives some intuition behind the principles on which the termination analysis is based. Chapter 4 gives a brief overview of the components which the analysis consists of. Chapter 5 describes how the analysis has been developed step by step. Chapter 6 reports the results of practical experiments with the analysis. Chapter 7 presents the technical details of two versions of the analysis — a simple one, which was our first attempt, and a more complex one, which is the final version. Chapter 8 concludes and describes related and future work.

The thesis has two appendices: Appendix A, which is mainly included for the author's own reference, describes (some of) the notational conventions used in this thesis. Appendix B lists the subject programs used in the experiments in Chapter 6.

[3]To generalize a variable means "to change its binding time from static to dynamic."

Contributions of this Work. The work presented here is an extension of [Holst, 1991], its main contributions are handling of the higher order case, and we hope, a more intuitive presentation of the ideas. The work also includes an evaluation of the analysis based on empirical results. Finally, we expect that the analysis can serve as a template for other similar analyses.

Chapter 2

Preliminary Definitions

In this chapter we define the subject language and define what it means for a program to be *well-annotated* [Jones et al., 1993].

2.1 Language

We are essentially dealing with the untyped lambda-calculus augmented with named functions. To get an explicit handle on all lambda expressions and their free variables prior to the analysis, we lambda lift the program and tuple the arguments, such that all functions have two argument tuples; one with the free variables and one with the lambda bound.

Following common practice in offline partial evaluation we use underlining to annotate an expression as dynamic. The language also contains a *lift* expression, which is used for static expressions appearing in dynamic contexts. The syntax of the language after lambda-lifting and annotation is given in Figure 2.

$$x \in \text{Var}, \quad e \in \text{Exp}, \quad f \in \text{Fname}, \quad o \in \text{Operator}, \quad c \in \text{Constant}$$

$$\text{Prog} \;::=\; f_1(x_{1,1}, \ldots, x_{1,m})(x_{1,m+1}, \ldots, x_{1,n_1}) = e_1, \;\ldots$$

$$\begin{array}{lllllll}
\text{Exp} & ::= & c & \mid & x_{i,j} & \mid & o(e_1, \ldots, e_n) \\
 & \mid & \mathit{Clo}(f, \langle e_1, \ldots, e_n \rangle) & \mid & \mathit{if}\ e_1\ e_2\ e_3 & \mid & e\ (e_1, \ldots, e_n) \\
 & \mid & \mathit{lift}\ \text{e} & \mid & \underline{x_{i,j}} & \mid & \underline{o}(e_1, \ldots, e_n) \\
 & \mid & \underline{\mathit{Clo}}(f, \langle e_1, \ldots, e_n \rangle) & \mid & \underline{\mathit{if}}\ e_1\ e_2\ e_3 & \mid & e\ \underline{@}\ (e_1, \ldots, e_n)
\end{array}$$

Figure 2: Syntax

Evaluation of an expression of the form $\mathit{Clo}(f, \langle e_1, \ldots, e_n \rangle)$ causes the creation of an f-closure where the free variables are bound to the values of $e_1, \ldots, e_n$.

Example 2.1 This example illustrates the lambda-lifting. Suppose we have the following program:

$$\begin{array}{lcl} f(x) & = & g(x,x) \\ g(x,y) & = & (\lambda z.+\ x\ y\ z)\ (42) \end{array}$$

The abstraction is lifted out and replaced by the function h.

$$\begin{array}{lcl} f()(x) & = & \mathit{Clo}(g,\langle\rangle)\ (x,x) \\ g()(x,y) & = & \mathit{Clo}(h,\langle x,y\rangle)\ (42) \\ h(x,y)(z) & = & +\ x\ y\ z \end{array}$$

□

After the lambda lifting all calls in the program are of the form $e\ (e_1,\ldots,e_n)$. Since the named functions in the original program do not have any free variables, calls to them now have the form $\mathit{Clo}(f,\langle\rangle)\ (e_1,\ldots,e_n)$.

For simplicity, we assume that the functions are named f_1 to f_n and that the variables in function f_i are named $x_{i,1}$ to $x_{i,m}$. For readability, we use ordinary names in examples when the subscripts are not needed. Furthermore, we assume that f_1 is the goal function, that the initial arguments to f_1 are first order, and that there are no calls to f_1 in the program.

2.2 Well-annotatedness

Annotated programs. The job of the binding-time analysis is to annotate the source program in such a way, that the subsequent specialization cannot commit a binding-time error (i.e., attempt to generate code for something static, or attempt to evaluate something dynamic).

Binding-Time Checking Rules. A binding-time type $t \in$ BindingTime is either first-order static (S), dynamic (D), or a function: $t ::= S \mid D \mid (t_1,\ldots,t_n) \to t$. We shall use the term *static* to cover both first-order and function binding times.

A binding-time environment τ maps variables to their binding times and function names to the binding time of the closure they return (i.e., either dynamic or a function): $\tau : (\mathrm{Var} + \mathrm{Fname}) \to \mathrm{BindingTime}$.

Rules for checking the annotations of expressions are given in Figure 3. The rules are the usual ones for a monovariant binding-time analysis, although Clo and DynClo may look unfamiliar. If the closure is to be created at specialization time, then the Clo rule is used. If it is to be created at runtime, then the DynClo rules is used, and the expression is marked as residual. Note that the free variables in a residual closure need not be dynamic. This allows the specializer to evaluate under dynamic lambdas thereby specializing the abstraction.

In a judgement $\tau \vdash e : t$, τ is a binding-time environment, e is an annotated expression, and t is the binding time of e in the environment τ.

(Const) $$\tau \vdash c : S$$

(Lift) $$\frac{\tau \vdash e : S}{\tau \vdash \mathit{lift}\ e : D}$$

(Var) $$\tau[x_{i,j} \mapsto t] \vdash x_{i,j} : t, \qquad t \neq D$$

(DynVar) $$\tau[x_{i,j} \mapsto D] \vdash \underline{x}_{i,j} : D$$

(Op) $$\frac{\tau \vdash e_1 : S \quad \ldots \quad \tau \vdash e_n : S}{\tau \vdash o(e_1, \ldots, e_n) : S}$$

(DynOp) $$\frac{\tau \vdash e_1 : D \quad \ldots \quad \tau \vdash e_n : D}{\tau \vdash \underline{o}(e_1, \ldots, e_n) : D}$$

(Clo) $$\frac{\tau \begin{bmatrix} x_{i,1} \mapsto t_1,\ \ldots,\ x_{i,m} \mapsto t_m, \\ x_{i,m+1} \mapsto t_{m+1},\ \ldots,\ x_{i,k} \mapsto t_k, \\ f_i \mapsto (t_{m+1},\ \ldots,\ t_k) \to t \end{bmatrix} \vdash e_1 : t_1 \ \ldots \ e_m : t_m}{\mathit{Clo}(f_i, \langle e_1, \ldots, e_m \rangle) : (t_{m+1},\ \ldots,\ t_k) \to t}$$

(DynClo) $$\frac{\tau \begin{bmatrix} x_{i,1} \mapsto t_1,\ \ldots,\ x_{i,m} \mapsto t_m, \\ x_{i,m+1} \mapsto D,\ \ldots,\ x_{i,k} \mapsto D, \\ f_i \mapsto D \end{bmatrix} \vdash e_1 : t_1 \ \ldots \ e_m : t_m}{\underline{\mathit{Clo}}(f_i, \langle e_1, \ldots, e_m \rangle) : D}$$

(If) $$\frac{\tau \vdash e_1 : S \quad \tau \vdash e_2 : t \quad \tau \vdash e_3 : t}{\tau \vdash \mathit{if}\ e_1\ e_2\ e_3 : t}$$

(DynIf) $$\frac{\tau \vdash e_1 : D \quad \tau \vdash e_2 : D \quad \tau \vdash e_3 : D}{\tau \vdash \underline{\mathit{if}}\ e_1\ e_2\ e_3 : D}$$

(Apply) $$\frac{\tau \vdash e : (t_1, \ldots, t_n) \to t \quad \tau \vdash e_1 : t_1 \quad \ldots \quad \tau \vdash e_n : t_n}{\tau \vdash e\ (e_1, \ldots, e_n) : t}$$

(DynApply) $$\frac{\tau \vdash e : D \quad \tau \vdash e_1 : D \quad \ldots \quad \tau \vdash e_n : D}{\tau \vdash e\ \underline{@}\ (e_1, \ldots, e_n) : D}$$

Figure 3: Binding-Time Checking Rules for Annotated Expressions

We shall briefly explain the rest of the rules. A constant is always first-order static. The lift rules allow us to lift first-order expressions; but not expressions of function type. The Op and DynOp rules say that arguments to built-in operators must either be first-order static or dynamic, and if one of them is dynamic, they must all be dynamic. The binding time of the condition in an if expression determines whether or not the expression is residual. Similarly, the lefthand expression in an application determines whether or not the application is residual.

Definition 2.1 *Given a binding-time environment τ and an annotated program. The program is said to be* well-annotated, *if for every function $f_i(\ldots)(\ldots) = e_i$ we have $\tau[f_i \mapsto t_i] \vdash e_i : t_i$ according to the rules in Figure 3. [Jones et al., 1993].*

Similar to showing that evaluation of a well-typed program cannot lead to a type error [Milner, 1978], it is possible to show that partial evaluation of a well-annotated program cannot lead to a binding-time error.

Chapter 3

Quasi-termination

In this chapter we define quasi-termination, state a central theorem giving a sufficient condition for a program to be quasi-terminating, give some intuition behind the principles on which the termination analysis is based, and introduce key terminology used in this thesis. The material presented in this chapter can also be found in [Holst, 1991], which contains a proof of Theorem 3.1.

Configurations. A *configuration* is composed of a *program point* identifying a position in a program, and the values of the variables at that point.

We can think of evaluation of a flowchart program as going through a sequence of configurations, where each configuration C_i is composed by a label and a mapping of variables to values.

$$C_1 \to C_2 \to \cdots \to C_n$$

It should be clear that each configuration uniquely determines the rest of the sequence. A program is terminating if the sequence of configurations is finite for all inputs. A program is *quasi-terminating* if it for any input only enters finitely many different configurations.

In pure functional programs a program point could be identified by an expression in the program, and a configuration would then be an (expression, environment) pair. Instead of considering a sequence of configurations it would be more natural to consider evaluation trees (finite or infinite) with (expression, environment) pairs as nodes, and with a subtree for each subevaluation. Each node uniquely determines the subtree of which it is the root. If the tree is finite for all input, then the program is terminating. This does not necessarily mean that the result is well defined, e.g., it might terminate with an error (typically something along the lines of "`can't take hd of nil`"). If the tree only contains finitely many different nodes for any input, then the program is quasi-terminating. Clearly this does not imply that the tree is finite.

Applying König's lemma "In a finitely branching infinite tree some paths will be infinite" makes it possible to consider paths in a tree instead of the whole tree.

If all paths are finite the tree will be finite, and if all paths contain only finitely many different nodes, the tree as a whole will contain only finitely many different nodes.

In the following example f is an obviously non-terminating, but quasi-terminating program, whereas g is neither a terminating nor a quasi-terminating program.

$$\begin{aligned} f(x) &= f(x) \\ g(x) &= g(x+1) \end{aligned}$$

Transitions. If for some input a program goes through configurations C_1, C_2, and C_3 in that order, we say that there is a *transition* from configuration C_1 to C_2, one from C_2 to C_3, and one from C_1 to C_3. We call the "smallest" transitions *1-step* or *simple* transitions and the others *composite* transitions. Since the proof for Theorem 3.1 argues that if the program goes through only finitely many 1-step transitions then it is terminating, it is important that the 1-step transitions are primitive, meaning they only take finite time.

In our lambda-lifted language we shall use function calls as 1-step transitions. They are primitive, since a function cannot loop without calling itself. In our language a transition is a mapping between environments, or argument tuples. The set of all 1-step transitions defined by a program is the collection of all the function calls that can occur during any run. It is important to notice that the set of all transitions in a program is not the set of 1-step transitions but the transitive closure of these.

An *endotransition* is a transition from a program point back to itself. This need not be a 1-step transition.

Example 3.1 Consider the following programs operating on natural numbers:

$$\begin{aligned} f(x,y) &= \textit{if } y=1 \textit{ then } y \textit{ else } f(x+1,y-1) \\ g(x,y) &= \textit{if } y=1 \textit{ then } g(2,x) \textit{ else } g(x+1,y-1) \end{aligned}$$

When f is called with $(2,5)$ the evaluation goes through the following sequence of configurations, where each arrow denotes a 1-step transition equivalent to a call in the program:

$$f(2,5) \rightarrow f(3,4) \rightarrow f(4,3) \rightarrow f(5,2) \rightarrow f(6,1)$$

If we, for example, compose the first two 1-step transitions we get a composite transition from $f(2,5)$ to $f(4,3)$. □

Inductive Arguments. We focus our interest on inductive arguments (i.e., arguments or argument positions, that depend on themselves). Consider the endotransition, which comes from a direct call from h to itself somewhere in its body:

$$h(a,b) = \ldots\ h(A,B)\ \ldots$$

If $A < a$ we say that a (in the sense "the first argument position") is *in situ decreasing*. $A < a$ should be read as "for all possible values of a this transition gives rise to a value of A that is strictly less than that of a." If the same holds for B (i.e., if $B < a$) then b is said to be *decreasing*. If we only can guarantee $A \leq a$ we say the argument is *in situ weakly decreasing*, or *in situ equal*. Similarly, if $B \leq a$ then b is said to be *weakly decreasing* or *equal*. If we cannot guarantee that the argument is at most equal we consider it an *increasing* argument (this is safe if imprecise). If an increasing argument depends on itself it is *in situ increasing*.

In the example above the call $f(x+1, y-1)$ has an in situ increasing first argument and an in situ decreasing second argument. Since all the transitions in f has this form Theorem 3.1 tells us that f is quasi-terminating.

Theorem 3.1 requires the in situ decreasing parameters controlling the loops to be *bounded*[1] for the program to be quasi-terminating. The reason behind this requirement is illustrated by the function g in the example above. We have an endotransition from g to itself, where y is in situ decreasing and x is in situ increasing, but the program is not quasi-terminating. Consider for example the following infinite evaluation path:

$$\underline{g(2,2)} \rightarrow g(3,1) \rightarrow \underline{g(2,3)} \rightarrow g(3,2) \rightarrow g(4,1) \rightarrow \underline{g(2,4)} \rightarrow \ldots$$

The problem is that y is reset every once in a while (at the underlined configurations) to a value on which there is no bound.

The rest of this thesis attempts to bring us in position to use the main result of [Holst, 1991] stated below. The problem is to collect an interesting approximation of the set of transitions.

Theorem 3.1 *Consider all transitions defined by a given program (composite as well as simple). Assume the domains are finitely downwards closed*[2] *with respect to some size ordering. Then the program is quasi-terminating if every endotransition with an in situ increasing argument, also has a bounded in situ decreasing argument.*

The Connection to Partial Evaluation. Let $trans_p$ be the set of all transitions during partial evaluation of a well-annotated program p. The dynamic variables do not occur in $trans_p$, as they do not take on any values during partial evaluation. If every endotransition in $trans_p$ with an in situ increasing argument, also has a bounded in situ decreasing argument, then the program will only enter finitely many different configurations during partial evaluation.

Termination of partial evaluation of the program can now be ensured by memoizing the configurations. Note that completely static configurations must

[1]An argument is said to be *bounded*, if it has an upper bound for every run.

[2]A domain is finitely downwards closed if for any value the set of values smaller than it is finite. This is strictly stronger than the descending chain condition [Davey and Priestley, 1990].

be memoized as well, otherwise a static loop could cause non-termination. This differs from what is normally done in partial evaluation, where only configurations that lead to specialized program points are memoized.

The objective of the termination analysis is to change some of the binding times to dynamic, such that the “offending” or “dangerous” arguments do not appear in $trans_p$.

Chapter 4

Overview of the Analysis

In this section we give an overview of the termination analysis including brief descriptions of the analyses on which it is based. Figure 4 illustrates the dependencies between the various analyses.

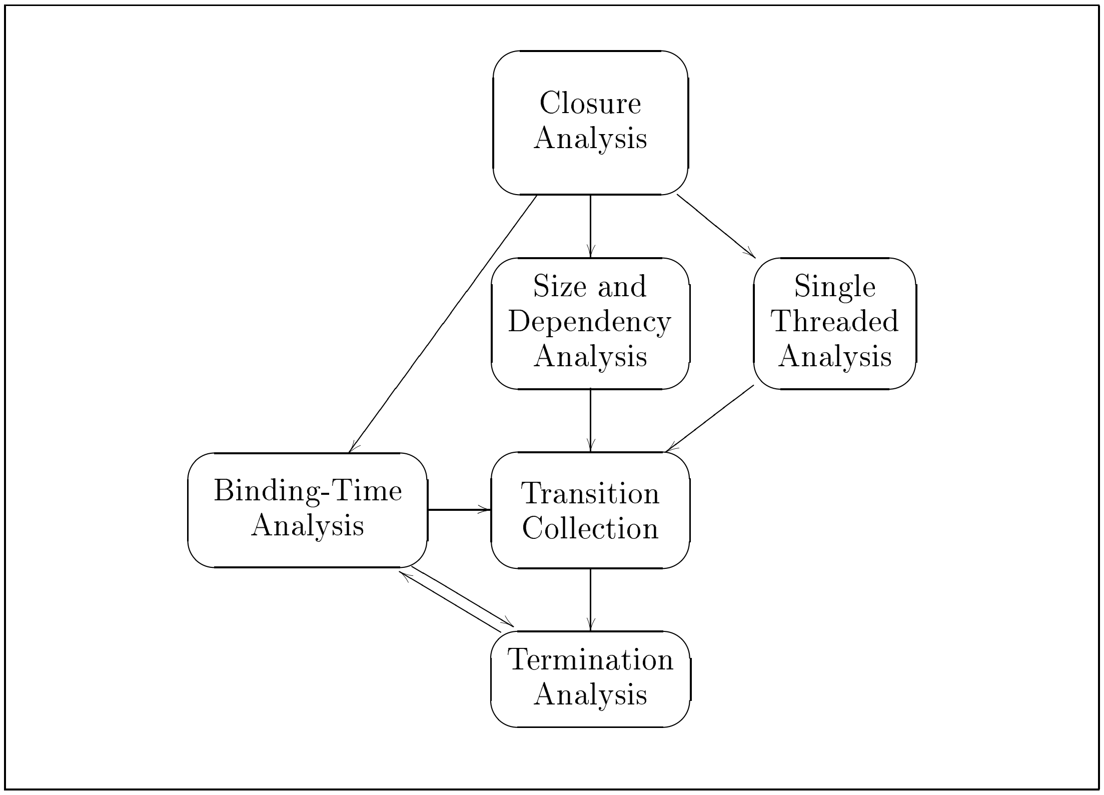

Figure 4: Overview of the Analysis

Closure Analysis. The net result of the closure analysis [Sestoft, 1988] is a safe approximation of which closures a given expression can evaluate to. In addition to tracing the flow of closures, we also trace the flow of first-order values (i.e., the analysis detects if a given expression can evaluate to a first-order value).

Binding-Time Analysis. The binding-time analysis produces a well-annotated version of the program. See Chapter 2.

Single-Threaded Analysis. This analysis identifies *single-threaded* closures (i.e., closures that are applied at most once). The information is be used to get higher precision when tracing flow in the program.

The analysis of [Turner et al., 1995] can be used to detect single-threaded closures.

Size and Dependency Analysis. The size analysis is a data-dependency analysis; which part of the arguments is the value build from, and how. The dependency analysis is a control-dependency analysis, which collects information about which part of the arguments the value depends on.

Transition Collection. Once the size information and dependency information have been obtained, an approximate 1-step transition is collected for each reachable call in the program[1]. Then the entire approximate set of transitions is generated by taking the transitive closures of all compositions of these.

Termination Analysis. We notice that an argument that does not depend on an in situ increasing argument must be bounded (this implies that it is not in situ increasing itself). Reasoning for this: Suppose we have a variable x that does not depend on any in situ increasing variables. Then x's value can only be increased a finite number of times, namely the number of increasing operators in the program. Assume that the value of x is increased more times than there are increasing operators in the program, then the same operator must have been used more than once, which means that x must depend on a variable that is in situ increasing.

If a variable is in situ increasing then its value is potentially built recursively, thus an upper bound can not be guaranteed without additional information.

Given a well-annotated program and an approximation of the set of transitions, we ensure termination of partial evaluation by repeating the following three steps until the annotations stabilize:

1. Generalize in situ increasing arguments occurring in endotransitions without a static bounded in situ decreasing argument.

2. Update the annotations to ensure that the program is well-annotated.

[1]Assuming, as usual, that every conditional branch can go either way.

3. Redo the transition collection.

This ensures that partial evaluation of the program will only enter finitely many different configurations.

In step 2 some binding times that depend on the ones changed in step 1 must also be changed to ensure well-annotatedness. However it is not necessary to restart the binding-time analysis from scratch, since step 1 only changes binding times from static to dynamic and not vice versa.

The complexity of the termination analysis can be significantly reduced by skipping step 3 and using the initial collections of transitions instead. The price is a slighty more conservative result.

Chapter 5

Development History

In this chapter we give an account of how we developed the analysis. We focus on the size and dependency analysis and the transition collection, since they are solving the core problems. First, we give a description of the first-order analysis, which served as a basis for the development. Second, we describe how a simple extension tames the control-flow problem in higher-order programs. Third, we describe a plausible but, as it turned out, unsuccessful attempt to deal with higher-order data flow using bounded trees. Fourth, we show how grammars solve most of the problems. Finally, we show how the remaining problems can be solved by taking binding times and single-threaded closures into account.

The technical details of the simple and the grammar-based analysis are given in Chapter 7.

The First-Order Analysis. The result of the size and dependency analysis in the first order case is, for each function, a (*size*, *dep*) pair describing how the return value relates to the parameters. Informally, the *dep* value is a set of argument positions identifying which of the parameters the return value depends on (including control dependency). Again informally, the *size* is a set of "basic" size values, which come in three guises: $D(i)$, $E(i)$, and I, where i is an argument position; the ith parameter. $D(i)$ means that the return value is guaranteed to be less that the ith parameter, $E(i)$ means "less than or equal", and I means that we cannot ensure anything. The meaning of a set of basic size values is the disjunction of the size values. The size value only describes data dependency and not control dependency. Consider the following functions:

$$\begin{array}{lcl} f(x,y) & = & g(x,x,y) \\ g(x,y,z) & = & \mathit{if}\ z\ \mathit{then}\ \mathit{hd}\ x\ \mathit{else}\ \mathit{hd}\ y \end{array}$$

The function g gets the size description $\{D(1), D(2)\}$ and the dependency description $\{1, 2, 3\}$, which together denote that the return value is either smaller than x or smaller than y and that it depends on x, y, and z. Why the either-or case is interesting should be clear when considering f's size description, which becomes $\{D(1)\}$.

Once the size and dependency information have been collected, an approximate 1-step transition is collected for each reachable call in the program. The size and dependency information are needed in case of calls that pass the return value of some function as a parameter to another (e.g., $f(g(x,y),z)$). A transition consists of two function names (the caller and the callee), a size tuple, and a dependency tuple. The two tuples describe how the callee's parameters relate to the caller's parameters. For example, the call $f(x,y) = g(x,x,y)$ would give rise to the following transition: $(\langle f,g\rangle, (\langle E(1),E(1),E(2)\rangle, \langle\{1\},\{1\},\{2\}\rangle)$.

If the program contains calls between two functions with different size-dep characteristics, a 1-step transition is collected for each call. This is crucial for the precision of the analysis. Consider the following two transitions: $\langle D(1),E(2)\rangle$, and $\langle E(1),D(2)\rangle$. If we only collected one transition for each function pair it would have to be $\langle E(1),E(2)\rangle$, and we would not detect that the arguments are decreasing.

Once we have collected the 1-step transitions, we take the transitive closure of all compositions of these to get an approximation of the program's control and data flow.

Example 5.1 The functions

$$\begin{array}{lcl} f(x) & = & g(x, cons\ x\ x) \\ g(x,y) & = & h(hd\ y, hd\ x, x) \end{array}$$

give rise to the following 1-step transitions (the dependency information has been left out for readability):

$$t_1 = (\langle f,g\rangle, \langle E(1), I\rangle), \quad t_2 = (\langle g,h\rangle, \langle D(2), D(1), E(1)\rangle)$$

When t_1 and t_2 are composed we get the following transition:

$$\begin{array}{rcl} t_2 \circ t_1 & = & (\langle f,h\rangle, \langle D(2),D(1),E(1)\rangle \circ \langle E(1),I\rangle) \\ & = & (\langle f,h\rangle, \langle D(2)\circ\langle E(1),I\rangle, D(1)\circ\langle E(1),I\rangle, E(1)\circ\langle E(1),I\rangle\rangle) \\ & = & (\langle f,h\rangle, \langle I, D(1), E(1)\rangle) \end{array}$$

The transition describes a transfer of control from f to h, and shows how h's parameters relate to f's parameters. We can verify that we have composed the transitions correctly by unfolding the call to g: $f(x) = h(hd\ (cons\ x\ x), hd\ x, x)$. Note that we are unable to detect that the first argument to h is actually equal to x. However, it turns out that the techniques developed in this thesis for handling higher-order values also apply to data structures, so the example does not prove to be a problem.

Transitions represent functions from argument tuples to argument tuples, and size descriptions represent functions from argument tuples to values, thus the composition of $\langle D(2), D(1), E(1)\rangle$ with $\langle E(1), I\rangle$ is found by composing each element of the former tuple with the latter. For example, the composition $D(2) \circ \langle E(1), I\rangle$ denotes a value that is increased first and then decreased. Since the size description does not tell how much the value is increased respectively decreased the result of the composition has to be I. The other compositions are computed using similar reasoning. □

The Higher-Order Case. A major difference between first-order and higher-order programs is the complexities involved in determining the control flow. In the first-order case control flow is deterministic except for conditionals where both branches must be considered. In the higher-order case the flow at every application is undetermined, since the function can be any of a number of different abstractions in the program.

Another problem is that the data flow is obscured when calls and returns cause data to flow in and out of closures. As we saw above this also occurs when dealing with data structures (i.e., $hd\ (cons\ x\ x)$): When a (potentially complex) structure is taken apart it is difficult to tell where the different parts originate from.

The Simple Higher-Order Analysis. We started out trying to solve the control-flow problem without addressing the data-flow problem (data flowing in and out of closures). We had the hope that most flow would be determined by first-order values (e.g., the decomposition of an expression in an interpreter) so that it would not matter that the analysis was conservative in the handling of higher-order values.

Experiments with the analysis showed that it is able to detect control flow with acceptable precision, whereas more precision is needed in the handling of data flow. The following example illustrates the kind of data flow the analysis is unable to detect:

$$\begin{array}{rcl} f(x,y) & = & g(\lambda z.hd\ x, y) \\ g(c,y) & = & c\ y \end{array}$$

The return value of f becomes I, when in fact it is smaller than x. The reason is that at the application of the closure we know absolutely nothing about the free variable x (a variable in another environment), so we have to make the conservative assumption that the result is increasing.

The simple higher-order analysis uses the same domains as the first-order analysis, so we just need to extend the analysis to handle the two new language constructs: abstraction and application. The size description of an abstraction is simply I (which makes sense, since an abstraction builds something) and the dependency description consists of a list of the free variables. The size and dependency description of an application is found by taking the least upper bound of all possible calls (the closure analysis is used to determined which abstractions can flow to the application). The free variables in the calls are described by I, since the abstract domains are inadequate for obtaining more detailed information.

Similarly, at each application we collect a 1-step transition for each of the abstractions that can flow to the application, and again the free variables are described by I. Since partial evaluation evaluates under dynamic abstractions we also collect a 1-step transition at each abstraction. Notice that the simple higher-order analysis does not use the binding-time information, so it must take into account that a given abstraction might become dynamic.

Improving the Representation of Higher-Order Values. A closure is represented as a label and a tuple with the value of the free variables; in general a tree. An obvious abstraction of this gives rise to an infinite tree augmented with a size and a dependency value at each node.

We present two different finite representations of the infinite trees, namely (1) cutting off the trees at depth k and (2) approximation using grammars [Jones and Muchnick, 1981]. The depth bound was tried first, in the hope that it would be sufficiently precise and simpler to implement than grammars, however, neither turned out to be the case.

The k-Bounded Analysis. The infinite tree is made finite by cutting it off at depth k, and attributing each remaining node with an extra size-dep value, which approximates the subtree (of the infinite tree) of which it is root.

Experiments showed that that there *is* a need to handle recursively built structures (and the values they contain). Furthermore, the k-bounded analysis turned out to be more complicated to implement than the grammar-based analysis.

The Grammar-Based Analysis. The infinite tree is approximated by a grammar that contains one rule for each label in the tree. Thus, a label that appears in two different contexts in the tree will be approximated by one rule in the grammar. We have chosen this representation for simplicity. The domains in more detail:

$$\begin{array}{lcl}\text{SizeDep}^- & = & \mathbb{P}(\text{Label} \times \text{Size} \times \text{Dep}) \\ \text{Gram} & = & \text{Label} \mapsto (\text{SizeDep}^-)^* \\ \text{SizeDep} & = & \text{SizeDep}^- \times \text{Gram}\end{array}$$

A SizeDep value consists of a grammar and a set of triples $\langle \mathit{label}, \mathit{size}, \mathit{dep}\rangle$ — one for each label the described function can return ($\mathbb{P}$ is the Hoare power domain). Note that a label also can denote a first-order value. We use a power domain to be able to get a more precise description of functions that can return more than one kind of value (e.g., a first-order value and a closure). For example, the abstract value (the size-dep information has been omitted for readability)

$$\{f_1\},\ [f_1 \mapsto \langle\{FO\}, \{f_1, f_2\}\rangle,\ f_2 \mapsto \langle\rangle]$$

denotes a f_1-closure, where the first free variable of the closure is a first-order value, and the second is either an f_1- or an f_2-closure. f_2 has no free variables.

Recall that dependency (both data- and control-dependency) in the first-order analysis was represented using argument positions. To use the extra precision the grammars give us, we need a more refined way to express dependency. Therefore, we extend the size domain with the forms $D(i\ f\ j)$ and $E(i\ f\ j)$, where i refers to the ith parameter, f is a label identifying an abstraction, and j refers to the jth free variable in the abstraction. For example, $D(1\ f_1\ 2)$ denotes a function whose return value is always less than the second free variable of an f_1-closure found somewhere in the function's first argument. Notice the close correspondence between the meaning of size values and the meaning of grammars. This correspondence makes it easy to compose SizeDep values.

Using Binding-Time Information. Below is the extract of a lambda interpreter (in its unannotated form and before lambda lifting):

```
int(e,ρ) = if hd e = 'Var then ρ(tl e)
           else if hd e = 'Lam then λx.int(hd(tl(tl e)), update(hd(tl e), x, ρ))
           else int(hd(tl e), ρ) int(hd(tl(tl e)), ρ)
```

Specialization of *int* with respect to some expression e will terminate, because something gets smaller in every transition during partial evaluation. However, the analysis, as it has been developed so far, is unable to detect this. The reason is that *int* is *not* quasi-terminating under normal evaluation (consider the call sequence spawned by interpretation of the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$).

The problem can be fixed by taking the binding-times into account when collecting transitions: Since the application in the last else-branch in *int* is dynamic (see the annotated version of the program below), it does not give rise to any transitions during partial evaluation, so they can safely be ignored. Using this extra information the analysis is able to detect that partial evaluation of the interpreter indeed terminates.

```
int(e,ρ) = if hd e = 'Var then ρ(tl e)
           else if hd e = 'Lam then $\underline{\lambda x}$.int(hd(tl(tl e)), update(hd(tl e), $\underline{x}$, ρ))
           else int(hd(tl e), ρ) $\underline{@}$ int(hd(tl(tl e)), ρ)
```

Note that taking the binding times into account introduces a cyclic dependency between the binding-time analysis, the transition collection, and the termination analysis (See the overview diagram in Chapter 4).

Using Single-Threaded Information. In order to get a more precise description of the free variables in an abstraction, it is sometimes beneficial to pretend that the calls inside a given abstraction are performed directly instead of via an application later during evaluation. At the abstraction the analysis can collect a more accurate description of the free variables, whereas little is known about the lambda-bound variables. At the application the situation is the reverse.

Since we are not interested in the size-dep relationship of the dynamic variables the result of the analysis can be improved by collecting transitions inside abstractions whose lambda-bound variables are dynamic. However, it is only safe to do so if the closure can at most be applied once (i.e., if it is single threaded), otherwise an infinite loop may be overlooked if the closure is copied an unbounded number of times (consider the situation above where the interpreter is executed on the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$: The closure is inserted into the environment and copied an unbounded number of times).

Chapter 6

Experiments

In this chapter we shall report the result of applying three versions of the analysis to a number of different programs, that contain non-trivial recursion and usages of higher-order features. The analysed programs and the relevant binding times of each program are given in appendix B. The analysis is implemented in Gofer and is fully automatic except for the binding-time and single-threaded analyses.

The PE column indicates whether or not partial evaluation of the program is guaranteed to terminate; T indicates termination for all static input and N indicates non-termination for some static input.

The S, G, and G+ columns contain the results of running three different versions of the analysis, namely the simple higher-order analysis (S), the grammar based analysis (G), and finally the grammar-based analysis that also takes binding-time and single-threaded information into account (G+).

A $\surd$ indicates that the analysis detects that specialization of the program always will terminate (when that is the case), or that the analysis safely detects that it may loop (in case of the ho.letrec program). A $\div$ indicates that the analysis performs one or more unnecessary generalizations — not that the analysis produces a wrong result.

	Program	PE	S	G	G+	Description
1	myflatten	T	÷	√	√	flatten a list of lists — written in cps
2	kmp	T	√[1]	√[1]	√[1]	naive pattern matcher[2]
3	closure	T	÷	√	√	extraction of static data from a closure
4	fo	T	√	√	√	int. for a first-order language with let
5	fo.func	T	√	√	√	"fo" extended with named functions
6	goto	T	√	√	√	int. for a goto-language
7	goto.while	T	÷	÷	÷	int. for a goto-language with while
8	ho	T	÷	÷	√	lambda interpreter
9	ho.cbn	T	÷	÷	√	call-by-name lambda interpreter
10	ho.cps	T	÷	÷	√	lambda interpreter written in cps
11	ho.let	T	÷	÷	√	"ho" extended with let
12	ho.func	T	÷	÷	√	"ho.let" extended with named functions
13	ho.letrec	N	√	√	√	"ho" extended with letrec

The Simple Higher-Order Analysis. The programs kmp, fo.func and goto are all first order, but contain non-trivial recursion. For example, in the fo.func interpreter: The environment is represented as a namelist *ns* and a valuelist *vs*, and to evaluate a let expression *(Let x* e_1 e_2*)* the following call is made:

$$int(e_2,cons(x,ns),cons(v,vs),p),$$

where v is the value of e_1, and p is the entire program. To interpret a function call *(Call f (*$e_1 \ldots e_n$*))* the body and the namelist of the function is looked up, and the arguments are evaluated:

$$int(lookupbody(f,p),lookupnames(f,p),(v_1 \ldots v_n),p).$$

The first call gives rise to the 1-step transition $\langle D(1), I, \_, E(4)\rangle$, and the second call gives rise to the 1-step transition $\langle D(4), D(4), \_, E(4)\rangle$. The third argument, the valuelist, is dynamic. This example shows the necessity of collecting more than one 1-step transition between two program points. If the descriptions of the two calls were joined, too much information would be lost.

A similar situation occurs for the interpreter of the goto language. The interpreter takes a list of statements as argument $(s_1 \ldots s_n)$, evaluates the first and proceeds to evaluate the rest unless s_1 is a jump statement. In the former case the list of statements are in situ decreasing and in the latter case the list is reset to a new value (when looking up the target of the jump). So, in this case we also need to keep the descriptions of the two calls apart.

[1]Assuming we are working with natural numbers.

[2]The result of specializing the naive pattern matcher is a Knuth-Morris-Pratt matcher (originally achieved by Consel and Danvy [Consel and Danvy, 1989]).

It is encouraging that the simple anlysis is able to detect that partial evaluation of fo terminates, because fo uses higher-order features to represent the environment (a function from names to values). However, the simple analysis quickly falls short when the data flow becomes a bit more complex; like in the closure and flatten examples. (The closure example is constructed to make the simple analysis fail).

The Grammar-Based Analysis. Although the grammar-based analysis is able to handle the closure and flatten examples, it is is obvious that use of binding-time and single-threaded information is crucial for getting accurate results (see the entries in the table for programs 8 to 12).

General Comments. Note that the analyses safely detect that ho.letrec may loop (i.e., if specialized with respect to the expression (*letrec x x*)).

None of analyses are unable to detect that partial evaluation of goto.while terminates. The reason is that the interpretation of the while-construct is implemented by adding the body of the loop to the while statement and interpreting the result. In order to handle this example we would need a richer size measure.

In conclusion: The analysis is strong enough to detect that partial evaluation of non-trivial interpreters using higher-order features will terminate.

Chapter 7

Technical Details

In this chapter we shall present the technical details of the simple higher-order analysis and of the grammar-based analysis including the use of binding-time and single-threaded information. First, we define a partial evaluation semantics and a partial evaluation transition semantics, then we give the analyses as abstract versions of these.

Since this thesis focus on the practical application of the analysis, we do not give a formal proofs for the safety of analyses. We only state the safety condition.

Before reading this chapter, we recommend that you read Chapter 5, which gives the necessary background.

7.1 The Basis of the Analyses

Domains. We define the following domains:

$$\begin{array}{lcl}
\mathrm{V} & = & \mathrm{BaseValue} + \mathrm{Fname} \times \mathrm{V}^* \\
\mathrm{Func} & = & \mathrm{V}^* \to \mathrm{V} \\
\mathrm{Fenv} & = & \mathrm{Fname} \to \mathrm{Func} \\
\mathrm{Trans} & = & \mathrm{Fname} \times \mathrm{Fname} \mapsto \mathbb{P}(\mathrm{Func}^*)
\end{array}$$

An element in the value domain V can either be a base value or a closure (represented by a function name and a tuple of values). We use FO and Clo (short for "first order" and "closure") to denote a left hand respectively a right hand member of the sum.

The function domain Func consist of functions from tuples of values into values. The function environment Fenv maps function names to functions in Func.

Trans is the domain of transitions. A transition consists of two function names; the caller f and the callee g, and a tuple of functions: $\langle \varphi_1, \ldots, \varphi_n \rangle$. The transition denotes a transfer of control from f to g, where φ_i expresses the value of the ith argument to g as a function of f's variables. The tuple describes both the free and the lambda bound variables in g. The free variables are described by $\varphi_1, \ldots, \varphi_{fv(g)}$ and the lambda bound variables are described by $\varphi_{fv(g)+1}, \ldots, \varphi_{fv(g)+arity(g)}$, where $fv(g)$ and $arity(g)$ are the number of free respectively lambda-bound variables in g. We use the Hoare powerdomain so that we can collect more than one transition between the same two functions.

For notational convenience we define function composition $\circ$ on Func and Func* in the obvious way:

$$\begin{aligned} &\_ \circ \_ : \text{Func} \times \text{Func}^* \to \text{Func} \\ &\varphi \circ \langle \psi_1, \ldots, \psi_n \rangle = \lambda \rho . \varphi \ \langle \psi_1 \rho, \ldots, \psi_n \rho \rangle \end{aligned}$$

$$\begin{aligned} &\_ \circ \_ : \text{Func}^* \times \text{Func}^* \to \text{Func}^* \\ &\langle \varphi_1, \ldots, \varphi_n \rangle \circ \overline{\psi} = \langle \varphi_1 \circ \overline{\psi}, \ldots, \varphi_n \circ \overline{\psi} \rangle \end{aligned}$$

The Size of Data Objects. We assume that (V,$\sqsubseteq$) is a flat domain and the size relation (V,$\leq$) is finitely downwards closed, and that it obeys the domain ordering (i.e., $\forall x \in \mathrm{V} : \perp \leq x.$).

Structured values (i.e., S-expressions) are ordered by inclusion; that is, $\forall x \in V : x < cons(x, \ldots)$, $x < cons(\ldots, x)$ and the transitive extension hereof.

The size relation on closures is defined pointwise; that is, $\forall f \in$ Fname, $\forall x_1, \ldots, x_n, y_1, \ldots, y_n \in \mathrm{V}$: $\mathrm{Clo}(f, \langle x_1, \ldots, x_n \rangle) \leq \mathrm{Clo}(f, \langle y_1, \ldots, y_n \rangle)$ iff $\forall j \in \{1, \ldots, n\} : x_j \leq y_j$.

Since we treat closures like data structures, we also define the following relation: $\forall f \in \text{Fname}, \forall x \in \mathrm{V} : x < \mathrm{Clo}(f, \langle \ldots, x, \ldots \rangle)$ and the transitive extension hereof.

The Partial Evaluation Semantics. Figure 5 defines evaluation of annotated expressions. $\perp$ denotes that the function is undefined, which means that it either loops, terminates abnormally with an error, or that the return value is dynamic. We do not distinguish between the three.

A more detailed partial evaluation semantics would generate code for residual expressions; either as a side effect or as the return value; for example, λ-mix [Jones et al., 1993, Chapter 8].

For static expressions the function $\mathcal{PE}$ defines evaluation in the usual way. To emphasize that we are interested in the *function* (from environment to value) an expression represents, we have moved ρ to the righthand side of the equality sign.

The meaning of a variable $x_{i,j}$ is the selector function π_j, which returns the jth element of a tuple. Base operators are given meaning by the oracle $\mathcal{O}$, and the use of a base operator o is the composition of $\mathcal{O}(o)$ with the tuple of functions describing the operands. The remaining expressions are handled in the usual way.

$\mathcal{PE} : \text{Exp} \to \text{Fenv} \to \text{Func}$

$$\begin{array}{lcl}
\mathcal{PE}[\![c]\!]\phi & = & \lambda\rho.FO(c) \\
\mathcal{PE}[\![x_{i,j}]\!]\phi & = & \pi_j \\
\mathcal{PE}[\![o(e_1,\ldots,e_n)]\!]\phi & = & \mathcal{O}(o) \circ \langle \mathcal{PE}[\![e_1]\!]\phi, \ldots, \mathcal{PE}[\![e_n]\!]\phi \rangle \\
\mathcal{PE}[\![\mathit{Clo}(f_i, \langle e_1,\ldots,e_n\rangle)]\!]\phi & = & \lambda\rho.\text{Clo}(f_i, \langle \mathcal{PE}[\![e_1]\!]\phi\rho, \ldots, \mathcal{PE}[\![e_n]\!]\phi\rho \rangle) \\
\mathcal{PE}[\![\mathit{if}\ e_1\ e_2\ e_3]\!]\phi & = & \lambda\rho.\mathit{if}\ \mathcal{PE}[\![e_1]\!]\phi\rho\ \mathit{then}\ \mathcal{PE}[\![e_2]\!]\phi\rho\ \mathit{else}\ \mathcal{PE}[\![e_3]\!]\phi\rho \\
\mathcal{PE}[\![e\ (e_1,\ldots,e_n)]\!]\phi & = & \lambda\rho.\phi(f_i)\ \langle v_1,\ldots,v_m, \mathcal{PE}[\![e_1]\!]\phi\rho, \ldots, \mathcal{PE}[\![e_n]\!]\phi\rho \rangle, \\
 & & \quad \text{where Clo}(f_i, \langle v_1,\ldots,v_m\rangle) = \mathcal{PE}[\![e]\!]\phi\rho \\
\mathcal{PE}[\![\mathit{lift}\ e]\!]\phi & = & \bot \\
\mathcal{PE}[\![\underline{x_{i,j}}]\!]\phi & = & \bot \\
\mathcal{PE}[\![\underline{o}(e_1,\ldots,e_n)]\!]\phi & = & \bot \\
\mathcal{PE}[\![\underline{\mathit{Clo}}(f_i, \langle e_1,\ldots,e_n\rangle)]\!]\phi & = & \bot \\
\mathcal{PE}[\![\underline{\mathit{if}}\ e_1\ e_2\ e_3]\!]\phi & = & \bot \\
\mathcal{PE}[\![e\ \underline{@}\ (e_1,\ldots,e_n)]\!]\phi & = & \bot
\end{array}$$

$$pe_p = \mathit{fix}\ \lambda\phi.[f_i \mapsto \mathcal{PE}[\![e_i]\!]\phi \mid f_i(\ldots)(\ldots) = e_i \in p]$$

Figure 5: Partial Evaluation Semantics

The Partial Evaluation Transition Semantics. The function $\mathcal{CPE}_{f_i}$ (Collecting Partial Evaluation) in Figure 6 will, given the (annotated) body of f_i, the function environment and the variable environment, collect the 1-step transitions that originate from f_i.

The interesting entries are the rules for the creation of closures, conditionals, and applications.

- During partial evaluation of a residual creation of a closure $\underline{\mathit{Clo}}(f_j, \ldots)$, a transition to f_j occurs, because the specializer evaluates the body of f_j (yielding a reduced expression). To model this behavior we collect a transition from the calling function to f_j in which the lambda-bound variables are described by $\bot$ (because they are dynamic; confer with the DynClo rule in Figure 3 in Chapter 2).

- Partial evaluation of a static creation of a closure does not immediately lead to a transition, because the body of the closure is not evaluated until the closure is applied.

- When a static conditional is specialized, only one of the branches will be executed, whereas both branches of a residual conditional are executed.

- Specialization of a static application involves a transfer of control, whereas the specialization of a residual application does not.

$$
\begin{array}{lll}
\mathcal{CPE}_* : \text{Exp} \rightarrow \text{Fenv} \rightarrow \text{V}^* \rightarrow \text{Trans} & & \\
& & \\
\mathcal{CPE}_{f_i}[\![c]\!]\phi\rho & = & \{\} \\
\mathcal{CPE}_{f_i}[\![\mathit{lift}\ e]\!]\phi\rho & = & \mathcal{CPE}_{f_i}[\![e]\!]\phi\rho \\
\mathcal{CPE}_{f_i}[\![x_{i,j}]\!]\phi\rho & = & \{\} \\
\mathcal{CPE}_{f_i}[\![\underline{x}_{i,j}]\!]\phi\rho & = & \{\} \\
\mathcal{CPE}_{f_i}[\![o(e_1,\ldots,e_n)]\!]\phi\rho & = & \bigsqcup_j \mathcal{CPE}_{f_i}[\![e_j]\!]\phi\rho \\
\mathcal{CPE}_{f_i}[\![\underline{o}(e_1,\ldots,e_n)]\!]\phi\rho & = & \bigsqcup_j \mathcal{CPE}_{f_i}[\![e_j]\!]\phi\rho \\
\mathcal{CPE}_{f_i}[\![\mathit{Clo}(f_j,\langle e_1,\ldots,e_n\rangle)]\!]\phi\rho & = & \bigsqcup_k \mathcal{CPE}_{f_i}[\![e_k]\!]\phi\rho \\
\mathcal{CPE}_{f_i}[\![\underline{\mathit{Clo}}(f_j,\langle e_1,\ldots,e_n\rangle)]\!]\phi\rho & = & \\
\qquad \bigsqcup_k \mathcal{CPE}_{f_i}[\![e_k]\!]\phi\rho\ \sqcup [\langle f_i,f_j\rangle \mapsto \langle \mathcal{PE}[\![e_1]\!]\phi,\ldots,\mathcal{PE}[\![e_n]\!]\phi,\bot,\ldots,\bot\rangle] & & \\
\mathcal{CPE}_{f_i}[\![\mathit{if}\ e_1\ e_2\ e_3]\!]\phi\rho & = & \\
\qquad \mathcal{CPE}_{f_i}[\![e_1]\!]\phi\rho\ \sqcup\ \mathit{if}\ \mathcal{PE}[\![e_1]\!]\phi\rho\ \mathit{then}\ \mathcal{CPE}_{f_i}[\![e_2]\!]\phi\rho\ \mathit{else}\ \mathcal{CPE}_{f_i}[\![e_3]\!]\phi\rho & & \\
\mathcal{CPE}_{f_i}[\![\underline{\mathit{if}}\ e_1\ e_2\ e_3]\!]\phi\rho & = & \bigsqcup_j \mathcal{CPE}_{f_i}[\![e_j]\!]\phi\rho \\
\mathcal{CPE}_{f_i}[\![e\ (e_1,\ldots,e_n)]\!]\phi\rho & = & \\
\qquad \mathcal{CPE}_{f_i}[\![e]\!]\phi\rho\ \sqcup\ \bigsqcup_j \mathcal{CPE}_{f_i}[\![e_j]\!]\phi\rho & & \\
\qquad \sqcup[\langle f_i,f_j\rangle \mapsto \langle \lambda\rho.v_1,\ldots,\lambda\rho.v_m,\mathcal{PE}[\![e_1]\!]\phi,\ldots,\mathcal{PE}[\![e_n]\!]\phi\rangle] & & \\
\qquad \text{where Clo}(f_j,\langle v_1,\ldots,v_m\rangle) = \mathcal{PE}[\![e]\!]\phi\rho & & \\
\mathcal{CPE}_{f_i}[\![e\ \underline{@}\ (e_1,\ldots,e_n)]\!]\phi\rho & = & \mathcal{CPE}_{f_i}[\![e]\!]\phi\rho\ \sqcup\ \bigsqcup_j \mathcal{CPE}_{f_i}[\![e_j]\!]\phi\rho
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{T}_{f_i}(\phi,\rho) = & T \sqcup \bigsqcup\{\mathcal{T}_{f_j}(\phi,\langle\varphi_1\rho,\ldots,\varphi_n\rho\rangle) \mid [\langle f_i,f_j\rangle \mapsto t] \in T, \langle\varphi_1,\ldots,\varphi_n\rangle \in t\} \\
& \text{where } T = \mathcal{CPE}_{f_i}[\![e_i]\!]\phi\rho \text{ and } f_i(\ldots)(\ldots) = e_i
\end{array}
$$

$$
\begin{array}{lll}
\mathit{trans}_{p,\rho} & = & \mathit{fix}\ \lambda T.(\mathcal{T}_{f_1}(pe_p,\rho)\ \sqcup\ [\langle f_i,f_k\rangle \mapsto \overline{\psi}\circ\overline{\varphi} \mid \\
& & \qquad [\langle f_i,f_j\rangle \mapsto t_1],[\langle f_j,f_k\rangle \mapsto t_2] \in T, \overline{\varphi} \in t_1, \overline{\psi} \in t_2]) \\
\mathit{trans}_p & = & \bigsqcup_{\rho\in(\text{V}\setminus\{\bot\})^n} \mathit{trans}_{p,\rho}
\end{array}
$$

Figure 6: Transition Semantics for Partial Evaluation

The function $\mathcal{T}_{f_i}$ will, given the function environment and the variable environment, collect the 1-step transitions from f_i and its descendents. Thus $\mathcal{T}_{f_1}(pe_p,\rho)$ is the set of 1-step transitions for the input ρ, and $\mathit{trans}_{p,\rho}$ is the transitive closure of these. The set of all transitions defined by the program (trans_p) is found by taking the union of $\mathit{trans}_{p,\rho}$ for all input.

7.2 The Simple Higher-Order Analysis

In this section we define the simple higher-order analysis. We start by defining the abstract domains, then the concretization functions, and finally an abstract version of the partial evaluation semantics and of the transition semantics.

Dependency. The domain Dep, which will be used to describe general dependency (both data dependency and control dependency) is built of on top of De_n:

$$\mathrm{De}_n = \{\bot\} \cup \{1, \ldots, n\}$$

De_n is a flat domain. For a given program, that uses the arities $i_1, \ldots, i_n$, the domain Dep is defined as the sum of the Hoare powerdomains of the relevant De_*'s:

$$\mathrm{Dep} = \mathbb{P}(\mathrm{De}_{i_1}) + \ldots + \mathbb{P}(\mathrm{De}_{i_n}),$$

Size. Similarly to the Dep domain, Size is defined on top of Si_n which in turn is built on top of De_n:

$$\mathrm{Si}_n = \{\bot\} \cup \{\mathrm{D}_n(i) \mid i \in \mathrm{De}_n \setminus \{\bot\}\} \cup \{\mathrm{E}_n(i) \mid i \in \mathrm{De}_n \setminus \{\bot\}\} \cup \{\mathrm{I}_n\}$$

Si_n has the following ordering: $\forall i \in \mathrm{De}_n \setminus \{\bot\} : \bot \sqsubset \mathrm{D}_n(i) \sqsubset \mathrm{E}_n(i) \sqsubset \mathrm{I}_n$. For a given program, that uses the arities $i_1, \ldots, i_n$, we define the domain Size as follows:

$$\mathrm{Size} = \mathbb{P}(\mathrm{Si}_{i_1}) + \ldots + \mathbb{P}(\mathrm{Si}_{i_n}),$$

In the following we leave out the subscripts on E, D, and I where the meaning is clear from the context.

AbsTrans. Like the name suggest AbsTrans abstract the Trans domain. Recall that a transition is a tuple of functions $\langle \varphi_1, \ldots, \varphi_n \rangle$. An abstract transition consists of a two tuples, a size and a dependency tuple, which abstracts the functions φ_1 to φ_n.

$$\mathrm{AbsTrans} = \mathrm{Fname} \times \mathrm{Fname} \mapsto \mathbb{P}(\mathrm{Size}^* \times \mathrm{Dep}^*)$$

Concretization Functions. First, we define the selection function *sel*, which takes a dependency descriptor ($i \in \mathrm{De}_n$) and a value tuple $\overline{x}$ and returns the set of values in $\overline{x}$ which i refers to.

$$\begin{array}{lcl} \multicolumn{3}{l}{sel : \mathrm{De}_n \times \mathrm{V}^n \to \mathcal{P}(\mathrm{V})} \\ sel(\bot, \overline{x}) & = & \{\} \\ sel(i, \overline{x}) & = & \{\pi_i(\overline{x})\} \end{array}$$

Concretization of a Size value is the least upper bound of the concretization of the maximal elements:

$$\begin{array}{l}
\mathit{ConSize} : \mathrm{Size} \to \mathcal{P}(\mathrm{V}^* \to \mathrm{V}) \\
\mathit{ConSize}(s) = \bigcup_{\sigma \in s} \mathit{ConS}(\sigma)
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{ConS} : \mathrm{Si}_n \to \mathcal{P}(\mathrm{V}^n \to \mathrm{V})} \\
\mathit{ConS}(\bot) & = & \{\} \\
\mathit{ConS}(\mathrm{D}_n(i)) & = & \{\phi \in \mathrm{V}^n \to V \mid \forall \overline{x} \in (\mathrm{V} \setminus \{\bot\})^n : \forall y \in \mathit{sel}(i, \overline{x}) : \phi(\overline{x}) < y\} \\
\mathit{ConS}(\mathrm{E}_n(i)) & = & \{\phi \in \mathrm{V}^n \to V \mid \forall \overline{x} \in (\mathrm{V} \setminus \{\bot\})^n : \forall y \in \mathit{sel}(i, \overline{x}) : \phi(\overline{x}) \leq y\} \\
\mathit{ConS}(\mathrm{I}_n) & = & \mathrm{V}^n \to \mathrm{V}
\end{array}$$

Similarly for the concretization of Dep values:

$$\begin{array}{l}
\mathit{ConDep} : \mathrm{Dep} \to \mathcal{P}(\mathrm{V}^n \to \mathrm{V}) \\
\mathit{ConDep}(d) = \bigcup_{\delta \in d} \mathit{ConD}(\delta)
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{ConD} : \mathrm{De}_n \to \mathcal{P}(\mathrm{V}^n \to \mathrm{V})} \\
\mathit{ConD}(\bot) & = & \{\} \\
\mathit{ConD}(i) & = & \{\phi \in \mathrm{V}^n \to V \mid \forall \overline{x} \in (V \setminus \{\bot\})^n : \phi(\overline{x}) \text{ does not depend} \\
& & \qquad\qquad\qquad \text{on any value in } \overline{x} \text{ except those in } \mathit{sel}(i, \overline{x})\}
\end{array}$$

Concretization of an abstract transition is defined as follows:

$$\begin{array}{l}
\mathit{ConAbsT} : \mathrm{AbsTrans} \to \mathrm{Trans} \\
\mathit{ConAbsT}(\theta) = \bigsqcup [\langle f, g \rangle \mapsto \langle \varphi_1, \ldots, \varphi_n \rangle \mid [\langle f, g \rangle \mapsto \tau] \in \theta, \\
\qquad\qquad\qquad\qquad\qquad\qquad (\langle s_1, \ldots, s_n \rangle, \langle d_1, \ldots, d_n \rangle) \in \tau, \\
\qquad\qquad\qquad\qquad\qquad\qquad \varphi_i \in \mathit{ConSize}(s_i) \cap \mathit{ConDep}(d_i)]
\end{array}$$

Simple Size Analysis. The simple size analysis, which is an abstract version of the partial evaluation semantics, is given in Figure 7. Since the analysis does not take the binding times of the program into account, it is defined on unannotated expressions. When comparing the abstract version with the original one, you just have to keep in mind that every entry (except constant) abstracts two cases. This is safe, if imprecise (recall that residual expressions evaluate to $\bot$).

A constant's relation to the variables is in general unknown, thus it is given the size description I. The variable $x_{i,j}$ is equal to the jth element in the environment, and is therefore described by $\mathrm{E}(j)$. The size behavior of a base operator is looked up in $\mathcal{OS}$ and composed with the operand tuple. The definition of $\mathcal{OS}$ is straightforward: $\mathcal{OS}(\mathit{cons}) = \mathrm{I}$, $\mathcal{OS}(\mathit{hd}) = \mathrm{D}(1)$, etc. The creation of a closure is described by I, which is natural since it builds something. For conditionals the least upper bound of the two branches are taken as usual.

For applications: Given the set of possible closures the lefthand expression can evaluate to, the least upper bound of the closure bodies composed with a tuple describing the variables in the body are taken. The first elements of the

$$\mathcal{S} : \text{Exp} \to (\text{Fname} \to \text{Size}) \to \text{Size}$$

$$\begin{array}{lcl}
\mathcal{S}[\![c]\!]\phi & = & \text{I} \\
\mathcal{S}[\![x_{i,j}]\!]\phi & = & \text{E}(j) \\
\mathcal{S}[\![o(e_1,\ldots,e_n)]\!]\phi & = & \mathcal{OS}(o) \circ^\sharp \langle \mathcal{S}[\![e_1]\!]\phi,\ldots,\mathcal{S}[\![e_n]\!]\phi\rangle \\
\mathcal{S}[\![\mathit{Clo}(f,\langle e_1,\ldots,e_n\rangle)]\!]\phi & = & \text{I} \\
\mathcal{S}[\![\mathit{if}\ e_1\ e_2\ e_3]\!]\phi & = & \mathcal{S}[\![e_2]\!]\phi\ \sqcup\ \mathcal{S}[\![e_3]\!]\phi \\
\mathcal{S}[\![e\ (e_1,\ldots,e_n)]\!]\phi & = & \bigsqcup\{\phi(f) \circ^\sharp \langle \text{I},\ldots,\text{I},\mathcal{S}[\![e_1]\!]\phi,\ldots,\mathcal{S}[\![e_n]\!]\phi\rangle \mid \\
 & & \quad f \in \mathit{labels}(e), f \neq \mathit{FO}, \mathit{arity}(f) = n\}
\end{array}$$

$$\mathit{size}_p = \mathit{fix}\lambda\phi.[f_i \mapsto \mathcal{S}[\![e_i]\!]\phi \mid f_i(\ldots)(\ldots) = e_i \in p]$$

Figure 7: Simple Size Analysis

tuple describe the free variables in the closure. The rest of the tuple describes the lambda-bound variables. The free variable's relation to the current environment is in general unknown, so they are described by I. We ignore the possible application of closures of the wrong arity (and the application of first-order values) since this error situation can be caught at specialization time. The function *labels* returns the labels a given expression may evaluate to. It is defined by the closure analysis.

Composition ($\circ^\sharp$) of Size values is defined as follows:

$$\_ \circ^\sharp \_ : \text{Size}^* \times \text{Size}^* \to \text{Size}^*$$
$$\langle s_1,\ldots,s_n\rangle \circ^\sharp S = \langle s_1 \circ^\sharp S,\ldots,s_n \circ^\sharp S\rangle$$

$$\_ \circ^\sharp \_ : \text{Size} \times \text{Size}^* \to \text{Size}$$
$$s \circ^\sharp S = \textstyle\bigsqcup_{\sigma\in s} \sigma \circ^\sharp S$$

$$\_ \circ^\sharp \_ : \text{Si}_n \times \text{Size}^n \to \text{Size}$$
$$\sigma \circ^\sharp S = \begin{cases} \bot, & \text{if } \sigma = \bot \text{ or } \exists i : \pi_i(S) = \bot \\ \bigsqcup\{\mathit{decrease}(\sigma') \mid \sigma' \in \pi_i(S)\}, & \text{if } \sigma = \text{D}(i) \\ \pi_i(S), & \text{if } \sigma = \text{E}(i) \\ \text{I}, & \text{if } \sigma = \text{I} \end{cases}$$

$$\mathit{decrease} : \text{Si}_n \to \text{Si}_n$$
$$\mathit{decrease}(\sigma) = \begin{cases} \text{D}(\delta), & \text{if } \sigma = \text{E}(\delta) \\ \sigma, & \text{otherwise} \end{cases}$$

Simple Dependency Analysis. The simple dependency analysis is given in Figure 8. Constants do not depend on any variables, a variable depends on itself, base operators are treated like in the size analysis, the creation of a closure $Clo(f, \langle e_1, \ldots, e_n \rangle)$ depends on the variables found in the expressions e_1 to e_n, conditionals depend on all the variable found in the expression, and applications are treated like in the size analysis, except that the free variables are described by the dependency of the lefthand expression.

$$\mathcal{D} : \text{Exp} \to (\text{Fname} \to \text{Dep}) \to \text{Dep}$$

$$\begin{array}{lcl}
\mathcal{D}[\![c]\!]\psi & = & \{\} \\
\mathcal{D}[\![x_{i,j}]\!]\psi & = & \{j\} \\
\mathcal{D}[\![o(e_1, \ldots, e_n)]\!]\psi & = & \mathcal{OD}(o) \circ^\sharp \langle \mathcal{D}[\![e_1]\!]\psi, \ldots, \mathcal{D}[\![e_n]\!]\psi \rangle \\
\mathcal{D}[\![Clo(f, \langle e_1, \ldots, e_n \rangle)]\!]\psi & = & \bigsqcup_i \mathcal{D}[\![e_i]\!]\psi \\
\mathcal{D}[\![if\ e_1\ e_2\ e_3]\!]\psi & = & \mathcal{D}[\![e_1]\!]\psi \sqcup \mathcal{D}[\![e_2]\!]\psi \sqcup \mathcal{D}[\![e_3]\!]\psi \\
\mathcal{D}[\![e\ (e_1, \ldots, e_n)]\!]\psi & = &
\end{array}$$

$$\bigsqcup\{\psi(f) \circ^\sharp \langle \mathcal{D}[\![e]\!]\psi, \ldots, \mathcal{D}[\![e]\!]\psi, \mathcal{D}[\![e_1]\!]\psi, \ldots, \mathcal{D}[\![e_n]\!]\psi \rangle \mid f \in labels(e), f \neq FO, arity(f) = n\}$$

$$dep_p = fix \lambda\psi.[f_i \mapsto \mathcal{D}[\![e_i]\!]\psi \mid f_i(\ldots)(\ldots) = e_i \in p]$$

Figure 8: Simple Dependency Analysis

Composition of Dep values is straightforward:

$$\_ \circ^\sharp \_ : \text{Dep*} \times \text{Dep*} \to \text{Dep*}$$
$$\langle d_1, \ldots, d_n \rangle \circ^\sharp D = \langle d_1 \circ^\sharp D, \ldots, d_n \circ^\sharp D \rangle$$

$$\_ \circ^\sharp \_ : \text{Dep} \times \text{Dep}* \to \text{Dep}$$
$$d \circ^\sharp D = \bigsqcup_{\delta \in d} \delta \circ^\sharp D$$

$$\_ \circ^\sharp \_ : \text{De}_n \times \text{Dep}^n \to \text{Dep}$$
$$\delta \circ^\sharp \langle d_1, \ldots, d_n \rangle = \begin{cases} \bot, & \text{if } \delta = \bot \text{ or } \exists i : d_i = \bot \\ d_\delta, & \text{otherwise} \end{cases}$$

Simple Transition Collection. The simple transition collection is shown in Figure 9. The creation of a closure gives rise to a transition because the specializer evaluates the body of dynamic abstractions. An application gives rise to a transition for each closure that can flow to the expression (again we ignore

$$\mathcal{CS}_* : \text{Exp} \to (\text{Fname} \to \text{Size}) \to (\text{Fname} \to \text{Dep}) \to \text{AbsTrans}$$

$$\begin{array}{lcl}
\mathcal{CS}_{f_i}[\![c]\!]\phi\psi & = & \{\} \\
\mathcal{CS}_{f_i}[\![o(e_1,\ldots,e_n)]\!]\phi\psi & = & \bigsqcup_k \mathcal{CS}_{f_i}[\![e_k]\!]\phi\psi \\
\mathcal{CS}_{f_i}[\![\mathit{Clo}(f_j,\langle e_1,\ldots,e_n\rangle)]\!]\phi\psi & = & \\
\qquad \bigsqcup_k \mathcal{CS}_{f_i}[\![e_k]\!]\phi\psi & & \\
\qquad \sqcup [\langle f_i, f_j\rangle \mapsto (\langle \mathcal{S}[\![e_1]\!]\phi,\ldots,\mathcal{S}[\![e_n]\!]\phi,\bot,\ldots,\bot\rangle, & & \\
\qquad\qquad \langle \mathcal{D}[\![e_1]\!]\psi,\ldots,\mathcal{D}[\![e_n]\!]\psi,\bot,\ldots,\bot\rangle)] & & \\
\mathcal{CS}_{f_i}[\![\mathit{if}\ e_1\ e_2\ e_3]\!]\phi\psi & = & \bigsqcup_k \mathcal{CS}_{f_i}[\![e_k]\!]\phi\psi \\
\mathcal{CS}_{f_i}[\![e\ (e_1,\ldots,e_n)]\!]\phi\psi & = & \\
\qquad \mathcal{CS}_{f_i}[\![e]\!]\phi\psi \ \sqcup\ \bigsqcup_k \mathcal{CS}_{f_i}[\![e_k]\!]\phi\psi \ \sqcup & & \\
\qquad \bigsqcup [\langle f_i, f_j\rangle \mapsto (\langle \mathrm{I},\ldots,\mathrm{I},\mathcal{S}[\![e_1]\!]\phi,\ldots,\mathcal{S}[\![e_n]\!]\phi\rangle, & & \\
\qquad\qquad \langle \mathcal{D}[\![e]\!]\psi,\ldots,\mathcal{D}[\![e]\!]\psi,\mathcal{D}[\![e_1]\!]\psi,\ldots,\mathcal{D}[\![e_n]\!]\psi\rangle) \mid & & \\
\qquad\quad f_j \in \mathit{labels}(e), f_j \neq \mathit{FO}, \mathit{arity}(f_j) = n] & &
\end{array}$$

$$\begin{array}{lcl}
\mathit{1\text{-}atrans}_p & = & \bigsqcup_i \mathcal{CS}_{f_i}[\![e_i]\!]\ \mathit{size}_p\ \mathit{dep}_p, \\
 & & \text{where } f_i(\ldots)(\ldots) = e_i \in p, \text{ and } f_i \text{ is reachable} \\
\mathit{atrans}_p & = & \mathit{fix}\lambda\theta.([\langle f_i, f_k\rangle \mapsto SD_2 \circ^\sharp SD_1 \mid [\langle f_i, f_j\rangle \mapsto \tau_1], [\langle f_j, f_k\rangle \mapsto \tau_2] \in \theta, \\
 & & \qquad SD_1 \in \tau_1, SD_2 \in \tau_2] \sqcup \mathit{1\text{-}atrans}_p)
\end{array}$$

Figure 9: Simple Transition Collection

closures of the wrong arity). The transition's size and dependency tuples are defined precisely as in the size and the dependency analysis.

1-atrans$_p$ is an approximation of the 1-step transitions in the program (for any input), and *atrans*$_p$ is the transitive closure of these. Composition of two size-dep value is defined as follows:

$$\begin{array}{l}
\_ \circ^\sharp \_ : (\text{Size}^* \times \text{Dep}^*) \times (\text{Size}^* \times \text{Dep}^*) \to (\text{Size}^* \times \text{Dep}^*) \\
(S_1, D_1) \circ^\sharp (S_2, D_2) = (S_1 \circ^\sharp S_2, D_1 \circ^\sharp D_2)
\end{array}$$

Simple Termination Analysis. We use the result of the simple transition collection to determine which variables may be in situ increasing and which are guaranteed to be in situ decreasing: Given the endotransitions $\tau \in \mathit{atrans}_p(f_i, f_i)$ we classify the variable $x_{i,j}$ as

- in situ increasing, if $\exists (S,D) \in \tau,\ \exists I \in \pi_j(S),$ and $\exists j \in \pi_j(D)$
- in situ decreasing, if $\forall (S,D) \in \tau, \forall \sigma \in \pi_j(S) : \sigma \sqsubseteq \mathrm{D}(j)$.

Based on this approximation we can apply the three step algorithm described in Chapter 4 to generalize "offending" variables.

Safety. The condition for checking that the simple transition collection is safe with respect to the transition semantics is given below:

$$trans_p \sqsubseteq ConAbsT(atrans_p)$$

Verifying the condition amounts to checking each entry in the simple transition collection against the transition semantics for partial evaluation, and to check that $size_p$ and dep_p safely abstracts pe_p; that is, for each function f in the program:

$$\begin{array}{rcl} pe_p(f) & \in & ConSize(size_p(f)) \\ pe_p(f) & \in & ConDep(dep_p(f)) \end{array}$$

This, in turn, amounts to checking each entry in the simple size and simple dependency analyses against the partial evaluation semantics, and to check that $\circ^\sharp$ is a safe abstraction of $\circ$.

7.3 The Grammar-Based Analysis

In this section we define the grammar-based analysis. We do this by redefining the abstract domains and some of the semantic functions given in the previous section.

Dependency. The De_* domains are extended with selectors of the form $(i\ f\ j)$, where i is an argument position, f is a lambda label, and j refers to the jth free variable in f.

$$\mathrm{De}_n = \{\bot\} \ \cup\ \{i, \ldots, n\} \ \cup\ \{(i\ f\ j) \mid 1 \leq i \leq n,\ f \in \mathrm{Fname},\ 1 \leq j \leq fv(f)\}$$

De_n is ordered as follows: $\forall i \in \{1, \ldots, n\}, \forall f \in \mathrm{Fname}, \forall j \in \{1, \ldots, fv(f)\}$: $\bot \sqsubset (i\ f\ j) \sqsubset i$. Dep is still defined as the sum of the Hoare powerdomains of the relevant De_*'s:

$$\mathrm{Dep} = \mathbb{P}(\mathrm{De}_{i_1}) + \ldots + \mathbb{P}(\mathrm{De}_{i_n}),$$

Size. The definition of Si_n is unchanged, but the domain is different:

$$\mathrm{Si}_n = \{\bot\} \ \cup\ \{\mathrm{D}_n(\delta) \mid d \in \mathrm{De}_n \setminus \{\bot\}\} \ \cup\ \{\mathrm{E}_n(\delta) \mid d \in \mathrm{De}_n \setminus \{\bot\}\} \ \cup\ \{\mathrm{I}_n\}$$

Si_n has the following ordering: $\forall i \in \{1, \ldots, n\}, \forall f \in \mathrm{Fname}, \forall j \in \{1, \ldots, fv(f)\}$:

$$\begin{tikzcd}
 & \mathrm{I}_n \arrow[d, dash] & \\
 & \mathrm{E}_n(i) \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{D}_n(i) \arrow[dr, dash] & & \mathrm{E}_n(i\ f\ j) \arrow[dl, dash] \\
 & \mathrm{D}_n(i\ f\ j) \arrow[d, dash] & \\
 & \bot &
\end{tikzcd}$$

The definition of Size is unchanged:

$$\mathrm{Size} = \mathbb{P}(\mathrm{Si}_{i_1}) + \ldots + \mathbb{P}(\mathrm{Si}_{i_n}),$$

Grammars. The domains used in the grammar-based analysis are defined as follows (the domains are described in Chapter 5):

$$\begin{array}{lcl}
\mathrm{Label} & = & \mathrm{Fname} \ \cup \ \{FO\} \\
\mathrm{SizeDep}^{-} & = & \mathbb{P}(\mathrm{Label} \times \mathrm{Size} \times \mathrm{Dep}) \\
\mathrm{Gram} & = & \mathrm{Label} \mapsto (\mathrm{SizeDep}^{-})^{*} \\
\mathrm{SizeDep} & = & \mathrm{SizeDep}^{-} \times \mathrm{Gram} \\
\mathrm{AbsTrans} & = & \mathrm{Fname} \times \mathrm{Fname} \mapsto \mathbb{P}(\mathrm{SizeDep})
\end{array}$$

Concretization Functions. We redefine *sel* to match the new definition of the De_* domain:

$$\begin{array}{lcl}
\multicolumn{3}{l}{sel : \mathrm{De}_n \times \mathrm{V}^n \to \mathcal{P}(\mathrm{V})} \\
sel(\bot, \overline{x}) & = & \{\} \\
sel(i, \overline{x}) & = & \{\pi_i(\overline{x})\} \\
sel(i\ f\ j, \overline{x}) & = & \{y_j \mid \text{for all } \mathrm{Clo}(f, \langle y_1, \ldots, y_n \rangle) \text{ in } \pi_i(\overline{x})\}
\end{array}$$

That one value x is "in" another value y, means that x is either equal to y or a substructure hereof.[1]

[1] The relation "in" is the same as the "less than or equal" ($\leq$) relation. However, we define *sel* based on "in", so that we can change the $<$ relation without affecting *sel*.

The concretization function *Con* from SizeDep to $\mathcal{P}$(Func) is defined as follows (α denotes a SizeDep$^-$ value):

$$Con : \text{SizeDep} \to \mathcal{P}(\text{V}^* \to \text{V})$$

$$\begin{array}{lll} Con(\alpha, G) & = & Con'(\alpha,\ \mathit{fix}\ ConG(G)) \\ Con'(\alpha, \Phi) & = & \bigcup_{\langle l,s,d\rangle \in \alpha}(ConSize(s)\ \cap\ ConDep(d)\ \cap\ \Phi(l)) \end{array}$$

$$ConG : \text{Gram} \to (\text{Label} \mapsto \mathcal{P}(\text{V}^n \to \text{V})) \to (\text{Label} \mapsto \mathcal{P}(\text{V}^n \to \text{V}))$$

$$ConG([l_1 \mapsto \langle \ldots \rangle,\ \ldots,\ l_n \mapsto \langle \ldots \rangle])\Phi = \bigsqcup_i ConG'(l_i \mapsto \langle \ldots \rangle)\Phi$$

$$\begin{array}{l} ConG'(FO \mapsto \langle\rangle)\Phi = [FO \mapsto \{\phi \in \text{V}^n \to \text{V} \mid \forall \overline{x} \in (\text{V} \setminus \{\bot\})^n : \phi(\overline{x}) = \text{FO}(\_)\}] \\ ConG'(f \mapsto \langle \alpha_1, \ldots, \alpha_m \rangle)\Phi = \\ \qquad [f \mapsto \{\phi \in \text{V}^n \to \text{V} \mid \exists \psi_i \in Con'(\alpha_i, \Phi) : \forall \overline{x} \in (\text{V} \setminus \{\bot\})^n : \\ \qquad\qquad \phi(\overline{x}) = \text{Clo}(f, \langle \psi_1(\overline{x}), \ldots, \psi_m(\overline{x}) \rangle)\}] \end{array}$$

Concretization of a SizeDep value is defined as taking the union of the concretizations of the (Label, Size, Dep) triples found in the value. Concretization of a given triple is the intersection of the functions produced by the size information, the dependency information, and the grammar.

The definition of *ConSize* and *ConDep* are the same as in the previous section. Recall that they are based on the *sel* function.

For a given grammar G we define the set of functions it represents as a fixpoint of the sequence $\Phi_0, \Phi_1, \ldots$, where $\Phi_i(l)$ for some label l is the set of functions that return an l-value whose maximum closure-nesting depth is i.

ConG processes each production in the grammar in turn and joins the results. $ConG'$ takes one production $l \mapsto \ldots$ and returns a mapping from l to the set of functions it may produce. In the case of a *FO* label, $ConG'$ returns a mapping from *FO* to the set of functions that return a first order value. In the case a mapping from a closure label to a description of its free variables ($f \mapsto \langle \alpha_1, \ldots, \alpha_m \rangle$), $ConG'$ returns those functions that return an f-closure $\text{Clo}(f, \langle y_1, \ldots, y_m \rangle)$, where y_i is the return value of some function in $Con(\alpha_i, \Phi)$.

Example 7.1 Consider again the abstract value from Chapter 5 (stripped of size and dependency information):

$$\{l_1\},\ [l_1 \mapsto \langle \{FO\}, \{l_1, l_2\} \rangle,\ l_2 \mapsto \langle\rangle]$$

The first few iterations of the fixpoint are shown below (without size and dependency information):

$$\begin{array}{rcl}
\Phi_0 & = & [FO \mapsto \phi_{fo},\ l_2 \mapsto \phi_2] \\
\Phi_1 & = & [FO \mapsto \phi_{fo},\ l_2 \mapsto \phi_2, \\
& & \quad l_1 \mapsto \{\phi \in \mathrm{V}^n \to \mathrm{V} \mid \exists \psi_1 \in \Phi_0(FO),\ \exists \psi_2 \in \Phi_0(l_2) : \\
& & \qquad\qquad \forall \overline{x} \in (\mathrm{V} \setminus \{\bot\})^n : \phi(\overline{x}) = \mathrm{Clo}(l_1, \langle \psi_1(\overline{x}), \psi_2(\overline{x})\rangle)\}] \\
\Phi_2 & = & [FO \mapsto \phi_{fo},\ l_2 \mapsto \phi_2, \\
& & \quad l_1 \mapsto \{\phi \in \mathrm{V}^n \to \mathrm{V} \mid \exists \psi_1 \in \Phi_1(FO),\ \exists \psi_2 \in \Phi_1(l_2) \cup \Phi_1(l_1) : \\
& & \qquad\qquad \forall \overline{x} \in (\mathrm{V} \setminus \{\bot\})^n : \phi(\overline{x}) = \mathrm{Clo}(l_1, \langle \psi_1(\overline{x}), \psi_2(\overline{x})\rangle)\}] \\
\Phi_3 & = & \ldots,
\end{array}$$

where

$$\begin{array}{rcl}
\phi_{fo} & = & \{\phi \in \mathrm{V}^n \to \mathrm{V} \mid \forall \overline{x} \in (\mathrm{V} \setminus \{\bot\})^n : \phi(\overline{x}) = \mathrm{FO}(\_)\} \\
\phi_2 & = & \{\phi \in \mathrm{V}^n \to \mathrm{V} \mid \forall \overline{x} \in (\mathrm{V} \setminus \{\bot\})^n : \phi(\overline{x}) = \mathrm{Clo}(l_2, \langle\rangle)\}
\end{array}$$

□

Concretization of an AbsTrans value is defined as follows:

$$\begin{array}{l}
ConAbsT : \mathrm{AbsTrans} \to \mathrm{Trans} \\
ConAbsT(\theta) = \bigsqcup [\langle f, g\rangle \mapsto \langle \varphi_1, \ldots, \varphi_n\rangle \mid [\langle f, g\rangle \mapsto \langle x_1, \ldots, x_n\rangle] \in \theta, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \varphi_i \in Con(x_i)]
\end{array}$$

Size and Dependency Analysis. First we define an abstract analogue to the selection function *sel* given above:

$$\begin{array}{lcl}
sel^\sharp : \mathrm{De}_n \times \mathrm{SizeDep}^n \to \mathrm{SizeDep} & & \\
sel^\sharp(\bot, X) & = & \bot \\
sel^\sharp(i, X) & = & \pi_i(X) \\
sel^\sharp(i\ f\ j, X) & = & normalize(\alpha_{(ifj)}, G_i) \\
& & \text{where } \ \langle(\alpha_1, G_1), \ldots, (\alpha_n, G_n)\rangle = X \\
& & \qquad [\ldots, f \mapsto \langle \alpha_{(if1)}, \ldots, \alpha_{(ifm)}\rangle, \ldots] = G_i
\end{array}$$

$$\begin{array}{lcl}
normalize : \mathrm{SizeDep} \to \mathrm{SizeDep} & & \\
normalize(\alpha, G) & = & (\alpha, G|_L) \\
& & \text{where } L = \ \{l \mid \langle l, \_, \_\rangle \in \alpha\} \cup \\
& & \qquad\qquad \{l' \mid l \in L, \langle \alpha_1, \ldots, \alpha_n\rangle = G(l), \langle l', \_, \_\rangle \in \alpha_i\}
\end{array}$$

$sel^\sharp(i\ f\ j, X)$ selects the $\mathrm{SizeDep}^-$ value, which is the jth element of the tuple $G_i(f)$, where G_i is the grammar of the ith element of X. The $\mathrm{SizeDep}^-$ value is paired with a normalized version of G_i. The function *normalize* selects the labels from the grammar that are reachable, directly or indirectly. The notation $G|_L$ means the grammar restricted to the labels in L.

Figure 10 defines the grammar-based size and dependency analysis. The size and the dependency analyses are performed simultaneously, because it is easier

$$\mathcal{SD} : \text{Exp} \to (\text{Fnames} \to \text{SizeDep}) \to \text{SizeDep}$$

$$\begin{array}{lcl}
\mathcal{SD}[\![c]\!]\phi & = & \langle FO, \text{I}, \{\}\rangle, [FO \mapsto \langle\rangle] \\
\mathcal{SD}[\![x_{i,j}]\!]\phi & = & id(x_{i,j}) \\
\mathcal{SD}[\![o(e_1, \ldots, e_n)]\!]\phi & = & \mathcal{OSD}(o) \circ^\sharp \langle(\mathcal{SD}[\![e_1]\!]\phi)|_{\text{FO}}, \ldots, (\mathcal{SD}[\![e_n]\!]\phi)|_{\text{FO}}\rangle \\
\mathcal{SD}[\![Clo(f_i, \langle e_1, \ldots e_n\rangle)]\!]\phi & = & \langle f_i, \text{I}, \bigsqcup_{\langle l,s,d\rangle \in \alpha} d\rangle, \\
 & & G \sqcup \\
 & & [f_i \mapsto \langle \mathcal{SD}[\![e_1]\!]\phi\downarrow_{\text{SizeDep}^-}, \ldots \mathcal{SD}[\![e_n]\!]\phi\downarrow_{\text{SizeDep}^-}\rangle], \\
 & & \text{where } (\alpha, G) = \bigsqcup_j \mathcal{SD}[\![e_j]\!]\phi \\
\mathcal{SD}[\![if\ e_1\ e_2\ e_3]\!]\phi & = & (\bigsqcup\{\langle l, s, d_1 \sqcup d_2\rangle \mid \langle l, s, d_1\rangle \in \alpha_1\}), G_1 \\
 & & \text{where} \\
 & & (\alpha_1, G_1) = \mathcal{SD}[\![e_2]\!]\phi \ \sqcup\ \mathcal{SD}[\![e_3]\!]\phi \\
 & & (\alpha_2, G_2) = \mathcal{SD}[\![e_1]\!]\phi \ \sqcup\ \mathcal{SD}[\![e_2]\!]\phi \ \sqcup\ \mathcal{SD}[\![e_3]\!]\phi \\
 & & d_2 = \bigsqcup_{\langle \_,\_,d\rangle \in \alpha_2} d \\
\mathcal{SD}[\![e\ (e_1, \ldots, e_n)]\!]\phi & = & \bigsqcup\{\ \phi(l) \circ^\sharp \langle x_{l1}, \ldots, x_{lm}, y_1, \ldots, y_n\rangle \mid \\
 & & \qquad \langle l, \_, \_\rangle \in \alpha,\ l \neq FO, arity(l) = n\} \\
 & & \text{where } \begin{array}{lcl} (\alpha, G) & = & \mathcal{SD}[\![e]\!]\phi \\ \langle \alpha_{l1}, \ldots, \alpha_{lm}\rangle & = & G(l) \\ x_{li} & = & normalize(\alpha_{li}, G) \\ y_i & = & \mathcal{SD}[\![e_i]\!]\phi \end{array} \\
\mathcal{SD}[\![\underline{\bullet}]\!]\phi & = & \bot
\end{array}$$

$$sizedep_p = fix\ \lambda\phi.[f_i \mapsto \mathcal{SD}[\![e_i]\!]\phi \mid f_i(\ldots)(\ldots) = e_i \in p]$$

Figure 10: Combined Size and Dependency Analysis

to maintain one grammar augmented with size and dependency information than two grammars with size respectively dependency information.

Like in the simple analysis constants are marked as increasing and depending on no variables.

In the partial evaluation semantics a variable is interpreted as a selector picking a value from the environment tuple. In this analysis a variable is interpreted as an abstraction of this projection as well as a description of the structure of the value; basically it is a grammar where each part describes that it is equal to itself. For the jth variable in the ith function, id builds this SizeDep value as follows:

$$\begin{array}{lcl}
id(x_{i,j}) & = & normalize(\alpha, G_{all}) \\
 & & \text{where } \begin{array}{lcl} \alpha & = & \{\langle l, E(j), j\rangle \mid l \in labels(x_{i,j})\} \\ G_{all} & = & \bigsqcup[l_k \mapsto \langle \alpha_{k1}, \ldots, \alpha_{km}\rangle \mid \text{for all labels } l_k] \\ \alpha_{kh} & = & \{\langle l, E(j\ l_k\ h), (j\ l_k\ h)\rangle \mid l \in labels(x_{k,h})\} \end{array}
\end{array}$$

In the rule for base operators the notation $|_{\mathrm{FO}}$ is used to restrict SizeDep values to first-order values; that is, $(\alpha, G)|_{\mathrm{FO}} \overset{\text{def}}{=} (\{\langle l, s, d\rangle \mid \langle l, s, d\rangle \in \alpha, l = \mathrm{FO}\}, [\mathrm{FO} \mapsto \langle\rangle])$. $\mathcal{OSD}$ maps operators to their SizeDep descriptions.

The creation of a f_i-closure is described by a SizeDep value whose grammar maps f_i to a tuple of SizeDep$^-$ values describing the free variables. Since the free variables may contain closures as well, their grammars are collected.

For conditionals size information are collected from the branches and dependency information are collected from all three subexpressions.

For an application $e(e_1, \ldots, e_n)$ we apply each possible closure e can evaluate to to the value of the argument tuple $(e_1, \ldots, e_n)$ and take the least upper bound of the results. Each closure consist of a label and a description of its free variables. The label is looked up in the function environment and composed with a tuple describing both the free and the bound variables.

Since the grammar-based analysis take the binding-times into account, all residual expressions (including *lift*) evaluates to $\bot$.

Composition of SizeDep values is defined as follows:

$$\begin{array}{l}
\circ^\sharp : \mathrm{SizeDep}^* \times \mathrm{SizeDep}^* \to \mathrm{SizeDep}^* \\
\langle x_1, \ldots, x_n\rangle \circ^\sharp Y = \langle x_1 \circ^\sharp Y, \ldots, x_n \circ^\sharp Y\rangle
\end{array}$$

$$\begin{array}{l}
\circ^\sharp : \mathrm{SizeDep} \times \mathrm{SizeDep}^* \to \mathrm{SizeDep} \\
x \circ^\sharp Y = \begin{cases} \bot, & \text{if } x = \bot \text{ or } \exists i : \pi_i(Y) = \bot \\ (\mathit{compSD}(x{\downarrow_{\mathrm{SizeDep}^-}}, Y), \mathit{compG}(x, Y)), & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathit{compSD} : \mathrm{SizeDep}^- \times \mathrm{SizeDep}^* \to \mathrm{SizeDep}^- \\
\mathit{compSD}(\alpha, X) = \bigsqcup\{\langle l, \sqcup_{\sigma \in s} \mathit{compS}(\sigma, X), \sqcup_{\delta \in d} \mathit{compD}(\delta, X)\rangle \mid \langle l, s, d\rangle \in \alpha\}
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{compS} : \mathrm{Si}_n \times \mathrm{SizeDep}^n \to \mathbb{P}(\mathrm{Si}_m)} \\
\mathit{compS}(\bot, X) & = & \bot \\
\mathit{compS}(\mathrm{D}(\delta), X) & = & \bigsqcup\{\mathit{decrease}(\sigma) \mid \langle \_, s, \_\rangle \in \mathit{sel}^\sharp(\delta, X){\downarrow_{\mathrm{SizeDep}^-}}, \sigma \in s\} \\
\mathit{compS}(\mathrm{E}(\delta), X) & = & \bigsqcup\{s \mid \langle \_, s, \_\rangle \in \mathit{sel}^\sharp(\delta, X){\downarrow_{\mathrm{SizeDep}^-}}\} \\
\mathit{compS}(\mathrm{I}, X) & = & \{\mathrm{I}\}
\end{array}$$

$$\begin{array}{l}
\mathit{compD} : \mathrm{De}_n \to \mathrm{SizeDep}^n \to \mathbb{P}(\mathrm{De}_m) \\
\mathit{compD}(\delta, X) = \bigsqcup\{d \mid \langle \_, \_, d\rangle \in \mathit{sel}^\sharp(\delta, X){\downarrow_{\mathrm{SizeDep}^-}}\}
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{compG} : \mathrm{SizeDep} \times \mathrm{SizeDep}^* \to \mathrm{Gram}} \\
\mathit{compG}((\alpha, G), X) & = & [\ldots, l \mapsto \langle y_{l1}, \ldots, y_{lm}\rangle, \ldots] \ \sqcup \\
& & \mathit{compG}'(\alpha, X) \ \sqcup \ \mathit{compG}'(\alpha_{li}, X) \\
& & \text{where } \ y_{li} = \mathit{compSD}(\alpha_{li}, X) \\
& & \qquad\quad [\ldots, l \mapsto \langle \alpha_{l1}, \ldots, \alpha_{lm}\rangle, \ldots] = G \\
\mathit{compG}'(\alpha, X) & = & \bigsqcup\{\mathit{sel}^\sharp(\delta, X){\downarrow_{\mathrm{Gram}}} \mid \langle \_, \_, d\rangle \in \alpha, \delta \in d\}
\end{array}$$

Composition is split up into composition of size (*compS*), dependency (*compD*) and grammars (*compG*).

Composition of the Si_* value $\mathrm{E}(\delta)$ with a SizeDep tuple, as defined by *compS*, is the least upper bound of the size elements that δ select from the tuple. $\mathrm{D}(\delta)$ is treated similarly except that weakly decreasing functions in the SizeDep tuple becomes decreasing instead of remaining weakly decreasing.

The function *compD* simply takes the least upper bound of the dependency values that the De_* value selects.

The *compG* function builds a grammar that describes the result of the composition. The grammar has the same *shape* as the grammar describing the result of the function, but where the grammar of the function described the result relative to the arguments of the function the grammar for the result of the application must describe the result relative to the environment at the call site, so all the sizes and the dependencies are adjusted by composing them with the argument tuple. To this we add the grammars from the parts of the arguments that the result is build from.

Transition Collection. Figure 11 defines the transition collection used in the grammar-based analysis.

The interesting entries in the abstract semantics are those for the generation of closures and applications. The rest are straightforward.

Notice how static closure creation match static application, and residual closure creation match residual application. A static closure creation gives rise to transitions at application time and not at creation time, where as residual closure creation contribute at creation time but not when applied.

Residual closure creation yields a transition where the free variables are described by $\mathcal{SD}[\![e_i]\!]\phi$, and the lambda-bound, since they are dynamic, are described by $\perp$.

A static application gives rise to a transition for each closure of the right arity. The description of the free variables are taken from the grammar describing the closure and the description of the lambda-bound variables from the actual arguments.

Single-Threaded Lambdas. Extending the analysis to use information about single-threaded lambdas is easy. Given the set of single-threaded lambdas whose lambda-bound variables are dynamic *st*, we can improve the result of the analysis as follows: First, we annotate the lambdas in *st* as being single threaded, then we update the transition collection to use the residual closure rule for these lambdas as well (incidentally the residual closure rule does the job – it collects the calls under the lambda). Finally, we update the static application rule to ignore the

$$\mathcal{CS}_* : \text{Exp} \to (\text{Fname} \to \text{SizeDep}) \to \text{AbsTrans}$$

$$\begin{array}{lcl}
\mathcal{CS}_{f_i}[\![c]\!]\phi & = & \{\} \\
\mathcal{CS}_{f_i}[\![\mathit{lift}\ e]\!]\phi & = & \mathcal{CS}_{f_i}[\![e]\!]\phi \\
\mathcal{CS}_{f_i}[\![x_{i,j}]\!]\phi & = & \{\} \\
\mathcal{CS}_{f_i}[\![\underline{x}_{i,j}]\!]\phi & = & \{\} \\
\mathcal{CS}_{f_i}[\![o(e_1,\ldots,e_n)]\!]\phi & = & \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi \\
\mathcal{CS}_{f_i}[\![\underline{o}(e_1,\ldots,e_n)]\!]\phi & = & \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi \\
\mathcal{CS}_{f_i}[\![\mathit{Clo}(f_j,\langle e_1,\ldots,e_n\rangle)]\!]\phi & = & \bigsqcup_k \mathcal{CS}_{f_i}[\![e_k]\!]\phi \\
\mathcal{CS}_{f_i}[\![\underline{\mathit{Clo}}(f_j,\langle e_1,\ldots,e_n\rangle)]\!]\phi & = & \bigsqcup_k \mathcal{CS}_{f_i}[\![e_k]\!]\phi \sqcup \\
 & & [\langle f_i,f_j\rangle \mapsto \langle \mathcal{SD}[\![e_1]\!]\phi,\ldots,\mathcal{SD}[\![e_n]\!]\phi,\bot,\ldots,\bot\rangle] \\
\mathcal{CS}_{f_i}[\![\mathit{if}\ e_1\ e_2\ e_3]\!]\phi & = & \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi \\
\mathcal{CS}_{f_i}[\![\underline{\mathit{if}}\ e_1\ e_2\ e_3]\!]\phi & = & \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi \\
\mathcal{CS}_{f_i}[\![e\ (e_1,\ldots,e_n)]\!]\phi & = & \mathcal{CS}_{f_i}[\![e]\!]\phi \ \sqcup\ \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi \sqcup \\
 & & \bigsqcup[\ \langle f_i,l\rangle \mapsto \langle x_{l1},\ldots,x_{lm},y_1,\ldots,y_n\rangle \mid \\
 & & \quad \langle l,\_,\_\rangle \in \alpha,\ l \neq \mathit{FO}, \mathit{arity}(l)=n] \\
 & & \text{where } \begin{array}{lcl} (\alpha,G) & = & \mathcal{SD}[\![e]\!]\phi \\ \langle \alpha_{l1},\ldots,\alpha_{lm}\rangle & = & G(l) \\ x_{li} & = & \mathit{normalize}(\alpha_{li},G) \\ y_i & = & \mathcal{SD}[\![e_i]\!]\phi \end{array} \\
\mathcal{CS}_{f_i}[\![e\ \underline{@}\ (e_1,\ldots,e_n)]\!]\phi & = & \mathcal{CS}_{f_i}[\![e]\!]\phi \ \sqcup\ \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi
\end{array}$$

$$\begin{array}{lcl}
\mathit{1\text{-}atrans}_p & = & \bigsqcup_i \mathcal{CS}_{f_i}[\![e]\!]\mathit{sizedep}_p, \text{where } f_i(\ldots)(\ldots)=e\in p, \text{ and } f_i \text{ is reachable} \\
\mathit{atrans}_p & = & \mathit{fix}\ \lambda\theta.([\langle f_i,f_k\rangle \mapsto \mathit{SD}_2 \circ^\sharp \mathit{SD}_1 \mid [\langle f_i,f_j\rangle \mapsto \tau_1],[\langle f_j,f_k\rangle \mapsto \tau_2] \in \theta, \\
 & & \qquad \mathit{SD}_1 \in \tau_1,\ \mathit{SD}_2 \in \tau_2]\ \sqcup\ \mathit{1\text{-}atrans}_p)
\end{array}$$

Figure 11: Transition Collection

lambdas in *st*:

$$\begin{array}{lcl}
\mathcal{CS}_{f_i}[\![e\ (e_1,\ldots,e_n)]\!]\phi & = & \mathcal{CS}_{f_i}[\![e]\!]\phi \ \sqcup\ \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi \sqcup \\
 & & \bigsqcup[\ \langle f_i,l\rangle \mapsto \langle x_{l1},\ldots,x_{lm},y_1,\ldots,y_n\rangle \mid \\
 & & \quad \langle l,\_,\_\rangle \in \alpha,\ l \neq \mathit{FO},\ l \notin \mathit{st},\ \mathit{arity}(l)=n] \\
 & & \text{where } \ldots
\end{array}$$

Termination Analysis. The abstract transitions collected above is used to classify variables as in situ decreasing respectively in situ increasing. Given the endotransition $\tau \in \mathit{atrans}_p(f_i,f_i)$ we classify the variable $x_{i,j}$ as

- in situ increasing, if $\exists \mathit{SD} \in \tau \wedge \exists \langle \_, s, d\rangle \in \pi_j(\mathit{SD}),$ where $\mathrm{I} \in s \wedge j \in d$.
- as in situ decreasing, if $\forall \mathit{SD} \in \tau,\ \forall \langle \_, s, \_\rangle \in \pi_j(\mathit{SD}), \forall \sigma \in s :$

$$\sigma \sqsubseteq \mathrm{D}(j)\ \vee\ \exists f \in \text{Fname}, k \in \{1,\ldots,\mathit{fv}(f)\} : \sigma \sqsubseteq \mathrm{E}(j\ f\ k).$$

It is obvious that $x_{i,j}$ is in situ decreasing if it is described by D(j). It is less obvious that the abstract value E(j f k) also guarantees this. Recall that E(j f k) means that value is weakly decreasing of the kth free variable in an f-closure taken from $x_{i,j}$. Since the size ordering ($<$) treats closure as data structures, E(j f k) describes a substructure of $x_{i,j}$, which implies that $x_{i,j}$ is in situ decreasing.

Based on this approximation we can apply the three step algorithm described in Chapter 4 to generalize "offending" variables.

Safety. The condition for checking the safety of the grammar-based transition collection with respect to the transition semantics is given below:

$$trans_p \sqsubseteq ConAbsT(atrans_p)$$

7.4 Summary

We have defined a partial evaluation semantics, which defines the evaluation of static expressions, and a transition semantics for partial evaluation, which describes the data flow during partial evaluation of an annotated program given the values of static input.

We have given the details of the simple analysis and of the grammar-based analysis. The former includes a size analysis, a dependency analysis, a transition collection, and a termination analysis. The latter includes a combined size and dependency analysis, a transition collection, and a termination analysis.

We have showed how information about binding times and single threadedness can be used to improve the result of the analysis.

Chapter 8

Conclusion

8.1 Related Work

Offline Approaches. Jones, Gomard and Sestoft [Jones et al., 1993] present a termination analysis for a flowchart language, and [Glenstrup and Jones, 1996] give efficient algorithms for implementing a termination analysis for a tail-recursive first-order language. Both analyses use techniques similar to ours to reason about increasing and decreasing variables. However, where we classify variables that may be unbounded as dynamic until no changes occur, they start by a division of dubious and dynamic variables, and classify dubious variables as static when they can be guaranteed to be bounded, and in the end classify the remaining dubious variables as dynamic. Loosely, one can say that they approach the fixpoint from the bottom, where we approach it from the top. It is unclear whether we end up with the same fixpoint.

Online Approaches. Most online strategies work by comparing the static data at recursive calls (or the equivalent in other languages) with the values previously seen at the same program point. If the analysis cannot guarantee that specialization with respect to the static data will terminate then one or more generalizations are made on the fly.

Online termination strategies are used in supercompilation [Turchin, 1986, Sørensen and Glück, 1995], partial evaluation of higher-order functional languages [Katz and Weise, 92], and partial deduction [Leuschel and Martens, 1996], [Gallagher and Lafave, 1996] – to mention a few.

The advantage of using an online approach instead of an offline, is that you can take the actual values of the static data into account, and that you do not have to consider both branches of static conditionals. However, the disadvantage is that global flow-information is much harder to obtain.

It might be worthwhile to combine an offline and an online approach; that is, the offline analysis points out "dangerous" variables, which the online analysis monitors and generalizes if they seem to grow unboundedly.

8.2 Conclusion

We have extended the first-order analysis of [Holst, 1991] to the higher order case, thereby taking an step towards fully automatic partial evaluation of higher-order functional languages. Our analysis is strong enough to handle values flowing in and out of closures, however the analysis sometimes fail to recognize in situ decreasing parameters due to the inevitable aliasing, which is necessary to obtain a finite description.

The analysis has been developed hand in hand with our experimental implementation of the analysis. This has made it possible for us to focus on practical usefulness, in the sense that the analysis should be strong enough to handle a large class of interesting programs. The focus has not been on speed, elegance, or an extensive correctness proof. In our opinion this has been an essential choice. We had to go through four major revisions of the analysis before our implementation was capable of handling a sufficiently large class of interesting programs to be of interest in a real partial evaluator. Our experiments with the implementation of the analysis on interpreters written in different styles indicates, that the analysis is precise enough. The current implementation is too slow to be of use on programs of realistic size, but in our opinion we are not up against any inherent complexity problem; just a slow implementation.

8.3 Future Work

The techniques presented in this thesis rely heavily on the use of finitely downwards closed domains in the subject program, and it is not clear how they can be extended to domains, that do not have a natural well-founded size ordering.

The extension to structured data-types is straightforward, e.g., for pairs simply add a label for each cons in the program and collect the size information using the same techniques as for closures.

Before the analysis can be integrated into Similix efficient algorithms must be developed. We expect that the algorithms of Jones and Glenstrup can be extended to serve this purpose.

Bibliography

[Andersen, 1993] Lars Ole Andersen. Binding-time analysis and the taming of C pointers. In David Schmidt, editor, *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93*, pages 47–58, 1993.

[Andersen, 1994] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[Andersen and Holst, 1996] Peter Holst Andersen and Carsten Kehler Holst. Termination analysis for offline partial evaluation of a higher order functional language. In *Proceedings of the Third International Static Analysis Symposium (SAS)*, 1996.

[Bondorf, 1991] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Selected papers of ESOP '90, the 3rd European Symposium on Programming.

[Bondorf and Jørgensen, 1993] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation: extended version. Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993.

[Consel, 1993] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David Schmidt, editor, *ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 145–154, June 1993.

[Consel and Danvy, 1989] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.

[Davey and Priestley, 1990] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.

[Gallagher and Lafave, 1996] John Gallagher and Laura Lafave. Regular approximation of computation paths in logic and functional languages. In *Partial Evaluation, International Seminar, Dagsthul Castle, Germany*, volume 1110, pages 263–283. Lecture Notes in Computer Science, February 1996.

[Glenstrup and Jones, 1996] Arne J. Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Andrei Ershov Second International Conference "Perspectives of System Informatics"*. Lecture Notes in Computer Science, 1996. Upcoming.

[Holst, 1991] Carsten Kehler Holst. Finiteness analysis. In John Hughes, editor, *Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 473–495, Cambridge, Massachusetts, USA, August 1991. ACM, Springer-Verlag.

[Jones and Muchnick, 1981] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[Jones and Nielson, 1994] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[Jones et al., 1993] Neil D. Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* C.A.R. Hoare, Series Editor. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[Katz and Weise, 92] Morry Katz and Daniel Weise. Towards a new perspective on partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, San Francisco, June 92. ACM Press.

[Leuschel and Martens, 1996] Michael Leuschel and Bern Martens. Global control for partial deduction through characteristic atoms and global trees. In *Partial Evaluation, International Seminar, Dagsthul Castle, Germany*, volume 1110, pages 263–283. Lecture Notes in Computer Science, February 1996.

[Milner, 1978] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Rice, 1953] H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Transaction of the AMS*, 89:25–59, 1953.

[Sestoft, 1988] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988. 107 pages.

[Sim, 1995] Similix, 1995. Version 5.1. `ftp://ftp.diku.dk/diku/users/anders/Similix.tar.Z`.

[Sørensen and Glück, 1995] Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.

[Turchin, 1986] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Turner et al., 1995] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture*, pages 1–11, La Jolla, California, June 1995. ACM Press.

Appendix A

Notational Conventions

This appendix describes (some of) the notational conventions used in this thesis. They are mainly included for the author's own reference. Note that some symbols have more than one use.

Program texts are written in *italic*. Residual expressions are <u>underlined</u>.

x denotes a variable. $x_{i,j}$ denotes the jth argument to the ith function in a program. e denotes an expression. f denotes a function name, which at the same time serves as a lambda label. l denotes a label, which can either be FO or a function name. o denotes a base operator. c denotes a constant. p denotes a program.

τ denotes a binding-time environment and t denotes a binding-time type. C denotes a configuration.

x, y	$\in$	V	φ, ψ	$\in$	Func
$\overline{x}$	$\in$	V*	$\overline{\varphi}, \overline{\psi}$	$\in$	Func*
t	$\in$	$\mathbb{P}$(Func*)	T	$\in$	Trans
σ	$\in$	Si_*	δ	$\in$	De_*
s	$\in$	Size	d	$\in$	Dep
S	$\in$	Size*	D	$\in$	Dep^*
α	$\in$	$\mathrm{SizeDep}^-$	G	$\in$	Gram
x, y	$\in$	SizeDep	X, Y	$\in$	SizeDep*
τ	$\in$	$\mathbb{P}$(Size* $\times$ Dep*)	τ	$\in$	$\mathbb{P}$(SizeDep*)
θ	$\in$	AbsTrans			

Appendix B

Programs

This appendix lists the programs used in the experiments. Each program is accompanied by a brief description, the binding-time division of the goal function's parameters (the goal function is the first function), and information about the binding times and single threadedness of the abstractions and applications. The programs are shown in their unannotated form and before lambda lifting has been performed.

Program 1. Myflatten. Flatten a list of lists — written in continuation passing style.

Initial binding-time division: *e: S*. Both lambdas are single threaded.

```
flatten(e)     = f(e,λc.c)
f(e,c)         = if eq(e,'nil) then c 'nil
                 else f(hd e,λc2.c (append(hd e,c2)))
append(xs,ys)  = if eq(xs,'nil) then ys
                 else cons(hd xs,append(tl xs,ys))
```

Program 2. Kmp. A naive pattern matcher, which will produce a Knuth-Morris-Pratt matcher when specialized.

Initial binding-time division: *p: S, d: D*. The program is first order.

```
kmp(p,d)            = loop(p,d,p)
loop(p,d,pp)        = if eq(p,'nil) then 'yes
                        else if eq(d,'nil) then 'no
                        else if eq(hd p,hd d) then loop(tl p,tl d,pp)
                        else if eq(p,pp) then kmp(p,tl d)
                        else loop1(p,d,pp,
                                    statickmp(pp,tl pp,length(tl pp) + length(p)))
loop1(p,d,pp,np)    = if eq(np,pp) then kmp(pp,tl d)
                        else loop(np,d,pp)
```

```
staticmp(p,d,n)             = staticloop(p,d,n,p,d,n)
staticloop(p,d,n,pp,dd,nn) = if eq(n,0) then
                                 if and(eq(nn,0),eq(hd p,tl d)) then
                                   staticmp(pp,tl dd,sub1 nn)
                                 else
                                   p
                               else
                                 if eq(hd p,hd d) then
                                   staticloop(hd p,hd d,sub1 n,pp,dd,nn)
                                 else
                                   staticmp(pp,tl dd,sub1 nn)
length(xs)                  = if eq(xs,'nil) then 0
                               else 1 + length(tl xs)
```

Program 3. Closure. Extracting a static value from a closure.

Initial binding-time division: *x: S, y: D*. None of the applications are dynamic, the abstraction is static and single threaded.

```
f(x,y) = g(λz.hd x,y)
g(c,y) = c y
h(x)   = h(f(x,42))
```

Program 4. Fo. Interpreter for a first-order strict language with constants, variables, conditionals, and let expressions. The environment is represented as a function.

Initial binding-time division: *e: S, v: D*. All the abstractions and applications are static. None of the lambdas are single threaded.

```
run(e,v)   = int(e,λm.v)
int(e,r)   = if eq(hd e,'Const) then hd(tl e)
             else if eq(hd e,'Var) then r (hd(tl e))
             else if eq(hd e,'Cons) then
               cons(int(hd(tl e),r),int(hd(tl(tl e)),r))
             else if eq(hd e,'If) then
               if int(hd(tl e),r) then int(hd(tl(tl e)),r) else int(hd(tl(tl(tl e))),r)
             else if eq(hd e,'Let) then — Let x e1 e2
               int(hd(tl(tl(tl e))),upd(hd(tl e),int(hd(tl(tl e)),r),r))
             else
               'Error
upd(n,v,r) = λm.if eq(n,m) then v else r m
```

Program 5. Fo.func. Interpreter for a first-order strict language with constants, variables, conditionals, let expressions, and functions. The environment is represented as a name list and a value list. The program is a list of functions: $pgm := ((f_1\ (x_{11}\ \ldots\ x_{1n})\ e_1)\ \ldots\ (f_m\ (x_{m1}\ \ldots\ x_{mn})\ e_m))$

Initial binding-time division: *e: S, p: S, v: D*. The program is first order.

```
run(e,p,v)          = int(e,cons('x,'nil),cons(v,'nil),p)
int(e,ns,vs,p)      = if eq(hd e,'Const) then hd(tl e)
                      else if eq(hd e,'Var) then lookupvar(hd(tl e),ns,vs)
                      else if eq(hd e,'Cons) then
                        cons(int(hd(tl e),ns,vs,p),int(hd(tl(tl e)),ns,vs,p))
                      else if eq(hd e,'If) then
                        if int(hd(tl e),ns,vs,p) then
                          int(hd(tl(tl e)),ns,vs,p)
                        else
                          int(hd(tl(tl(tl e))),ns,vs,p)
                      else if eq(hd e,'Let) then — (Let x e1 e2)
                        int(hd(tl(tl(tl e))),
                             cons(hd(tl e),ns),
                             cons(int(hd(tl(tl e)),ns,vs,p),vs),
                             p)
                      else if eq(hd e,'Call) then — (Call f (e1 e2 ... en))
                        int(lookupbody(hd(tl e),p),
                             lookupnames(hd(tl e),p),
                             intlist(hd(tl(tl e)),ns,vs,p),
                             p)
                      else
                        'Error
lookupvar(n,ns,vs)= if eq(n,hd ns) then hd vs
                      else lookupvar(n,tl ns,tl vs)
lookupbody(f,p)     = if eq(f,hd(hd p)) then hd(tl(tl(hd p)))
                      else lookupbody(f,tl p)
lookupnames(f,p) = if eq(f,hd(hd p)) then hd(tl(hd p))
                      else lookupnames(f,tl p)
intlist(es,ns,vs,p) = if eq(es,'nil) then 'nil
                      else cons(int(hd es,ns,vs,p),intlist(tl es,ns,vs,p))
```

Program 6. Goto. An interpreter for a small goto-language consisting of the following expressions: *y, x=x+1, y=y+1, goto N, goto x, if x=0 then goto N, if y=0 then goto N*. The second argument to *int* is a list of expressions to be evaluated.

Initial binding-time division: *pgm: S, x: D, y: D*. The program is first order.

```
run(pgm,x,y)            = int(pgm,pgm,x,y)
int(pgm,e,x,y)          = if eq(hd(hd e),'y) then y
```

```
                            else if eq(hd(hd e), 'x=x+1) then int(pgm,tl e,cons(x,1),y)
                            else if eq(hd(hd e), 'y=y+1) then int(pgm,tl e,x,cons(y,1))
                            else if eq(hd(hd e), 'goto) then int(pgm,nth(pgm,hd(tl(hd e)),x,y)
                            else if eq(hd(hd e), 'gotox) then gotox(pgm,x,y)
                            else if eq(hd(hd e), 'ifx=0then) then
                              if eq(x,0) then int(pgm,nth(pgm,hd(tl(hd e)),x,y)
                                         else int(pgm,tl e,x,y)
                            else if eq(hd(hd e), 'ify=0then) then
                              if eq(y,0) then int(pgm,nth(pgm,hd(tl(hd e)),x,y)
                                         else int(pgm,tl e,x,y)
                            else
                              'Error
nth(xs,n)                 = if eq(xs,'nil) then
                              xs
                            else
                              if eq(n,0) then
                                xs
                              else
                                nth(tl xs,n-1)
gotox(pgm,x,y)            = gotox1(pgm,pgm,x,y,1)
gotox1(pgm,ptail,x,y,n)   = if eq(ptail,'nil) then
                              'Error
                            else
                              if eq(x,n) then
                                int(pgm,ptail,x,y)
                              else
                                gotox1(pgm,tl ptail,x,y,n+1)
```

Program 7. Goto.while An interpreter for the goto-language extended with a loop construct: *while x=0 do e*.

Initial binding-time division: *pgm: S, x: D, y: D*. The program is first order.

```
run(pgm,x,y)              = int(pgm,pgm,x,y)
int(pgm,e,x,y)            = if eq(hd(hd e),'y) then y
                            else if eq(hd(hd e),'x=x+1) then int(pgm,tl e,cons(x,1),y)
                            else if eq(hd(hd e),'y=y+1) then int(pgm,tl e,x,cons(y,1))
                            else if eq(hd(hd e),'goto) then int(pgm,tl e,nth(pgm,tl e),x,y)
                            else if eq(hd(hd e),'gotox) then gotox(pgm,x,y)
                            else if eq(hd(hd e),'ifx=0then) then
                              if eq(x,0) then int(pgm,nth(pgm,tl e),x,y)
                                         else int(pgm,tl e,x,y)
                            else if eq(hd(hd e),'ify=0then) then
```

```
                          if eq(y,0) then int(pgm,nth(pgm,tl e),x,y)
                                      else int(pgm,tl e,x,y)
                        else if eq(hd(hd e),'whilex=0do) then — ((whilex=0do e1) ...)
                          if eq(x,0) then int(pgm,cons(hd(tl(hd e)),e),x,y)
                                      else int(pgm,tl e,x,y)
                        else
                          'Error
nth(xs,n)             = if eq(xs,'nil) then
                          xs
                        else
                          if eq(n,0) then
                            xs
                          else
                            nth(tl xs,n-1)
gotox(pgm,x,y)        = gotox1(pgm,pgm,x,y,1)
gotox1(pgm,ptail,x,y,n) = if eq(ptail,'nil) then
                          'Error
                        else
                          if eq(x,n) then
                            int(pgm,ptail,x,y)
                          else
                            gotox1(pgm,tl ptail,x,y,n+1)
```

Program 8. Ho. An interpreter for a language with constants, variables, cons, conditionals, abstraction, and application.

Initial binding-time division: *e: S, v: D*. The application and abstraction used to implement the environment are static. The abstraction and application in the last two cases in *int* are dynamic. None of the lambdas are single threaded.

```
run(e,v)    = int(e,λm.v)
int(e,r)    = if eq(hd e,'Const) then hd(tl e)
              else if eq(hd e,'Var) then r (hd(tl e))
              else if eq(hd e,'Cons) then cons(int(hd(tl e),r),int(hd(tl(tl e)),r))
              else if eq(hd e,'If) then
                if int(hd(tl e),r) then int(hd(tl(tl e)),r) else int(hd(tl(tl(tl e))),r)
              else if eq(hd e,'Abs) then λx.int(hd(tl(tl e),upd(hd(tl e),x,r))
              else (int(hd e,r)) (int(hd e,r))
upd(n,v,r) = λm.if eq(n,m) then v else r m
```

Program 9. Ho.cbn. An interpreter implementing call-by-name evaluation.

Initial binding-time division: *e: S, v: D*. The application and abstraction used to implement the environment are static. The rest of the abstractions and applications in the program are dynamic. None of the lambdas are single threaded.

```
run(e,v)    = int(e,λm.v)
int(e,r)    = if eq(hd e,'Const) then hd(tl e)
                else if eq(hd e,'Var) then (r (hd(tl e))) 'foo
                else if eq(hd e,'Cons) then cons(int(hd(tl e),r),int(hd(tl(tl e)),r))
                else if eq(hd e,'If) then
                  if int(hd(tl e),r) then int(hd(tl(tl e)),r) else int(hd(tl(tl(tl e))),r)
                else if eq(hd e,'Abs) then λx.int(hd(tl(tl e)),upd(hd(tl e),x,r))
                else (int(hd(tl e),r)) λfoo.(int(hd(tl(tl e)),r))
upd(n,v,r) = λm.if eq(n,m) then v else r m
```

Program 10. Ho.cps. An interpreter for the same language used in "ho" — written in continuation passing style.

Initial binding-time division: *e: S, v: D*. The application and abstraction used to implement the environment are static. The continuations are static and single threaded (the lambda-bound variables in the continuations are dynamic). The rest of the abstractions and applications in the program are dynamic.

In order to get a good binding-time separation two eta-conversions must be inserted (one in the branch for abstractions and one in the branch for applications) [Bondorf, 1991]. These are not shown in the program below.

```
run(e,v)    = int(e,λm.v,λx.x)
int(e,r,c)  = if eq(hd e,'Const) then c (hd e)
                else if eq(hd e,'Var) then c (r (hd(tl e)))
                else if eq(hd e,'Cons) then
                  int(hd(tl e),r,λw1.int(hd(tl(tl e)),r,λw2.cons(w1,w2)))
                else if eq(hd e,'If) then
                  int(hd(tl e),
                      r,
                      λw1.if w1 then int(hd(tl(tl e)),r,c)
                                 else int(hd(tl(tl(tl e))),r,c))
                else if eq(hd e,'Abs) then
                  c (λw1.λc1.int(hd(tl(tl e)),upd(hd(tl e),w1,r),c))
                else
                  int(hd(tl e),r,λw1.int(hd(tl(tl e)),r,λw2.(w1 w2) c))
upd(n,v,r) = λm.if eq(n,m) then v else r m
```

Program 11. Ho.let. Interpreter for the language in "ho" extended with "let".

The binding times for the abstractions and applications in this program are the same as in "ho".

```
run(e,v)    = int(e,λm.v)
int(e,r)    = if eq(hd e,'Const) then hd(tl e)
```

```
                else if eq(hd e,'Var) then r (hd(tl e))
                else if eq(hd e,'Cons) then cons(int(hd(tl e),r),int(hd(tl(tl e)),r))
                else if eq(hd e,'If) then
                  if int(hd(tl e),r) then int(hd(tl(tl e)),r) else int(hd(tl(tl(tl e))),r)
                else if eq(hd e,'Abs) then λx.int(hd(tl(tl e),upd(hd(tl e),x,r))
                else if eq(hd e,'Let) then
                  int(hd(tl(tl(tl e))),upd(hd(tl e),int(hd(tl(tl e)),r),r))
                else (int(hd e,r)) (int(hd e,r))
upd(n,v,r) = λm.if eq(n,m) then v else r m
```

Program 12. Ho.func. Interpreter for a higher-order strict language with constants, variables, conditionals, let expressions, named functions, abstraction and application. The environment is represented as a name list and a value list.

Initial binding-time division: *e: S, p: S, v: D*. The abstraction and application at the end of the function *int* are dynamic.

```
run(e,p,v)          = int(e,cons('x,'nil),cons(v,'nil),p)
int(e,ns,vs,p)      = if eq(hd e,'Const) then hd(tl e)
                      else if eq(hd e,'Var) then lookupvar(hd(tl e),ns,vs)
                      else if eq(hd e,'Cons) then
                        cons(int(hd(tl e),ns,vs,p),int(hd(tl(tl e)),ns,vs,p))
                      else if eq(hd e,'If) then
                        if int(hd(tl e),ns,vs,p) then
                          int(hd(tl(tl e)),ns,vs,p)
                        else
                          int(hd(tl(tl(tl e))),ns,vs,p)
                      else if eq(hd e,'Let) then
                        int(hd(tl(tl(tl e))),
                             cons(hd(tl e),ns),
                             cons(int(hd(tl(tl e)),ns,vs,p),vs),
                             p)
                      else if eq(hd e,'Call) then
                        int(lookupbody(hd(tl e),p),
                             lookupnames(hd(tl e),p),
                             intlist(hd(tl(tl e)),ns,vs,p),
                             p)
                      else if eq(hd e,'Lam) then
                        λx.int(hd(tl(tl e)),cons(hd(tl e),ns),cons(x,vs),p)
                      else if eq(hd e,'App) then
                        (int(hd(tl e),ns,vs,p)) (int(hd(tl(tl e)),ns,vs,p))
                      else
                        'Error;
lookupvar(n,ns,vs)= if eq(n,hd ns) then hd vs
```

```
                      else lookupvar(n,tl ns,tl vs)
lookupbody(f,p)     = if eq(f,hd(hd p)) then hd(tl(tl(hd p)))
                      else lookupbody(f,tl p)
lookupnames(f,p)    = if eq(f,hd(hd p)) then hd(tl(hd p))
                      else lookupnames(f,tl p)
intlist(es,ns,vs,p) = if null es then 'nil
                      else cons(int(hd es,ns,vs,p),intlist(tl es,ns,vs,p))
```

Program 13. Ho.letrec. Interpreter for the language in “ho” extended with “letrec”.

Initial binding-time division: *e: S, v: D*. The abstractions and applications used to implement the environment are static. The abstraction and application used to interpret abstraction and application are dynamic. The abstractions and applications used to interpret letrec expressions are static. None of the lambdas are single threaded.

```
run(e,v)   = int(e,λm.v)
int(e,r)   = if eq(hd e,'Const) then hd(tl e)
             else if eq(hd e,'Var) then r (hd(tl e))
             else if eq(hd e,'Cons) then cons(int(hd(tl e),r),int(hd(tl(tl e)),r))
             else if eq(hd e,'If) then
               if int(hd(tl e),r) then int(hd(tl(tl e)),r) else int(hd(tl(tl(tl e))),r)
             else if eq(hd e,'Abs) then λx.int(hd(tl(tl e),upd(hd(tl e),x,r))
             else if eq(hd e,'Letrec) then
               int(hd(tl(tl(tl e))),fix (λr1.upd(hd(tl e),int(hd(tl(tl e)),r1),r)))
             else (int(hd e,r)) (int(hd e,r))
upd(n,v,r) = λm.if eq(n,m) then v else r m
fix(f)     = λx.f (fix f) x
```