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

Carsten Kehler Holst
Prolog Development Center
kehler@pdc.dk

Abstract. One of the remaining problems on the path towards fully automatic partial evaluation is ensuring termination of the specialization phase. In [10] 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 paper 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.

Keywords: partial evaluation, termination, abstract interpretation, size analysis.

1 Introduction

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. So termination must be ensured.

Almost all of today’s offline partial evaluators have unsafe termination properties (i.e., they are not guaranteed to terminate). The reason for this 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 [1, 5], Schism [6], and C-Mix [2, 3]). In this paper 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 for a higher order untyped strict functional language (e.g., Scheme) that is used to change 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 [10], 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, and whether they grow or get smaller along the possible evaluation paths. The gathered information is then used to generalize[1] variables for which upper bounds cannot be guaranteed. To adopt the approach from [10] 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 [10]. 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 [13]). 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 will not address in this paper, is *abnormal termination* of partial evaluation (errors occurring while executing static code). The problem has been fixed in Similix [1], which generates code to produce the error at run time when encountering an erroneous static expression.

Outline of the Paper. The paper presents an overview of the analysis. Non-essential technical details have been left out to give room for a report of several practical experiments. We give an account of how the analysis was developed

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

through trial and error in the hope that others may benefit from our experience. A more detailed description can be found in the technical report [4].

Section 2 defines the subject language used in this paper. Section 3 describes how quasi-termination forms the foundation of the termination analysis. Section 4 gives an overview of the termination analysis, and lists the component analyses on which it is based. Section 5 describes the development history of the analysis. Section 6 reports the results of applying the analysis to a number of different programs. Section 7 gives the central technical details. Section 8 concludes and describes related and future work.

Contributions of this Paper. The work presented here is an extension of [10], 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.

Prerequisites. This paper requires knowledge of partial evaluation corresponding to for example [11, Part II].

2 Language

We are essentially dealing with the untyped lambda-calculus augmented with named functions. Initially the program is lambda-lifted and the arguments are tupled, so each function has two arguments; a tuple with the free variables and one with the lambda bound. Thus, the lambda expression $\lambda x.y\ x$ becomes $\mathrm{Clo}(l, \langle y \rangle)$ where l is a freshly generated label (function) with the definition $l(y)(x) = y\ x$. Likewise, a function call $f(e_1, \ldots, e_n)$ becomes $\mathrm{Clo}(f, \langle\rangle)(e_1, \ldots, e_n)$.

When we use binding-time information we annotate the program by underlining the dynamic expressions. The termination analysis relies on the fact that the program is *well-annotated* [11].

3 Quasi-Termination

In this section 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 paper. The material presented in this section can also be found in Holst [10], 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 car 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 will 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 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:

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

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*[2] 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:

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

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 paper attempts to bring us in position to use the main result of Holst [10] 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*[3] *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.*

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

[3] 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 [8].

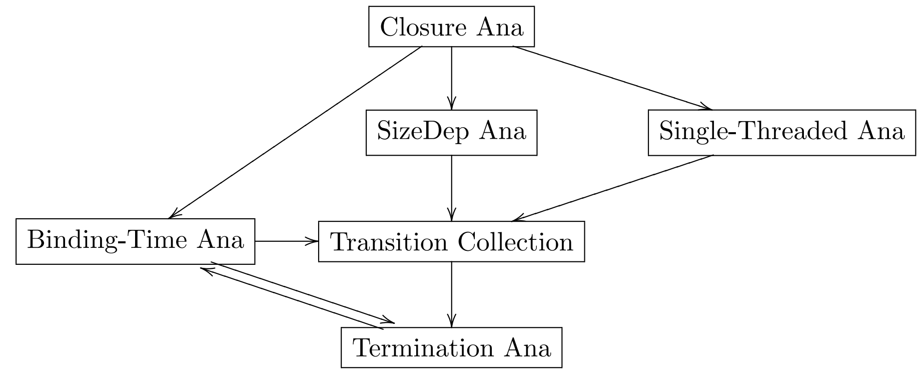
The Connection to Partial Evaluation. Let $trans_p$ be the set of all transitions during partial evaluation of a well-annotated program p. The dynamic variables do not occur in $trans_p$, as they do not take on any values during partial evaluation. If every endotransition in $trans_p$ with an in situ 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 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 $trans_p$.

4 Overview of the Termination Analysis

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



Closure Analysis. The net result of the closure analysis [14, 5], 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.

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

The analysis of Turner, Wadler, and Mossin [15] 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 standard 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[4]. 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. So 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.
3. Redo the transitions collection and termination analysis.

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

5 Development History

In this section we give an account of how we developed the analysis focusing 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 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

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

disjunction of the size values. The size value only describes data dependency and not control dependency. Consider the following functions:

$$\begin{aligned} f(x,y) &= g(x,x,y) \\ g(x,y,z) &= \text{if } z \text{ then } \mathit{car}\ x \text{ else } \mathit{car}\ y \end{aligned}$$

The return value of g becomes $\langle(\{D(1), D(2)\}, \{1,2,3\})\rangle$, meaning it is either smaller than x or smaller than y and it depends on x, y, and z. Why the either-or case is interesting should be clear when considering f whose return value becomes $\langle(D(1), \{1,2\})\rangle$.

Once the size and dependency information has been collected, an approximate 1-step transition is collected for each reachable call in the program. The size and dependency information is 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) and a tuple of (*size*, *dep*) pairs, which describes 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: $(\langle f,g\rangle, \langle E(1), E(1), E(2)\rangle)$ (the dependency information has been omitted for readability).

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 2. The functions

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

give rise to the following 1-step transitions:

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

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

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

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: $f(x) = h(\mathit{hd}\ (\mathit{cons}\ x\ x), \mathit{hd}\ x, x)$. 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 paper 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.

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-dep 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 size-dep 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.

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:

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

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.

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-dep value at each node describes the return value of functions more accurately than the simple size-dep values.

We present two different finite representations of the infinite trees, namely (1) cutting off the trees at depth k and (2) approximation using grammars [12]. 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:

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

A SizeDep value consists of a grammar and a set of triples $\langle \textit{label}, \textit{size}, \textit{dep} \rangle$ — one for each label the described function can return ($\mathcal{P}$ is the Hoare power domain). 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., two differently labeled lambdas). Note that a label also can denote a first-order value. For example, the abstract value (the size-dep information has been omitted for readability)

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

denotes a l_1-closure, where the first free variable of the closure is a first-order value, and the second is either an l_1- or an l_2-closure. l_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 $D(i\ l\ j)$ and $E(i\ l\ j)$, where i refers to the ith parameter, l is a label identifying an abstraction, and j refers to the jth free variable in the abstraction. For example, $D(1\ l_1\ 2)$ denotes a function whose return value is always less than the second free variable of an l_1-closure found 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:

```
int(e,ρ) = if car e = 'Var then ρ(cdr e)
           else if car e = 'Lam then λx.int(caddr e, update(cadr e, x, ρ))
           else int(cadr e, ρ) int(caddr 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-branchi n *int* is dynamic, 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.

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 Sect. 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 is performed directly instead of via an application later during evaluation. At the abstraction one 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 abstraction can at most be applied once (i.e., if it is single threaded), otherwise an infinite loop will be overlooked in case the closure is copied an unbounded number of times (consider the example above).

6 Experiments

In this section we shall report the result of applying the various versions of the analysis to a number of different programs, that contain non-trivial recursion and usages of higher-order features. 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 (using the obvious binding-time division of the initial parameters); T indicates termination and N indicates possible non-termination.

The S, G, and G+ columns contain the results of running three different versions of the analysis, namely the simple higher-order analysis (S), which uses the simple size-dep domains, 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
flatten	T	$\div$	$\div$	$\surd$	Flatten a list of lists, written in cps
kmp	T	$\surd$	$\surd$	$\surd$	naive pattern matcher[5]
closure	T	$\div$	$\surd$	$\surd$	extraction of static data from a closure
fo	T	$\surd$	$\surd$	$\surd$	int. for a first-order language with let
fo.func	T	$\surd$	$\surd$	$\surd$	"fo" extended with named functions
goto	T	$\surd$	$\surd$	$\surd$	int. for a goto-language
goto.while	T	$\div$	$\div$	$\div$	int. for a goto-language with while
ho	T	$\div$	$\div$	$\surd$	lambda interpreter
ho.cbn	T	$\div$	$\div$	$\surd$	call-by-name lambda interpreter
ho.cps	T	$\div$	$\div$	$\surd$	lambda interpreter written in cps
ho.let	T	$\div$	$\div$	$\surd$	"ho" extended with let
ho.func	T	$\div$	$\div$	$\surd$	"ho.let" extended with named functions
ho.letrec	N	$\surd$	$\surd$	$\surd$	"ho" extended with letrec

The simple higher-order analysis is able to detect that partial evaluation of fo terminates, which is encouraging since fo uses higher-order functions to represent the environment.

By inspection of the flatten example and most of the ho examples, it is obvious that use of binding-time and single-threaded information is crucial for getting accurate results. The analysis safely detects that ho.letrec may loop (e.g., if it is specialized with respect to the expression (*letrec x x*)).

[5] The result of specializing the naive pattern matcher is a Knuth-Morris-Pratt matcher (originally achieved by Consel and Danvy [7]).

The analysis is 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.

7 Central Technical Details

In this section we present selected technical details of the size analysis and the transition collection. A complete description of the analyses can be found in the technical report [4].

The Grammar-Based Size-Analysis. The central part of the size analysis is given below. A variable $x_{i,j}$ is described by the *identity* size-dep value for the values $x_{i,j}$ may evaluate to (i.e., *id* constructs a grammar that describes the structures of $x_{i,j}$'s possible values and inserts the relevant equal E(•) values at every node).

$$\begin{aligned}
&\mathcal{S} : \text{AnnExp} \to \text{Fenv} \to \text{SizeDep}\\
&\phi : \text{Fenv} = \text{Fnames} \to \text{SizeDep}
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\![x_{i,j}]\!]\phi &= \mathit{id}(x_{i,j})\\
\mathcal{S}[\![\mathit{Clo}(f_i, \langle e_1, \ldots, e_n\rangle)]\!]\phi &= \langle f_i, \mathrm{I}, [f_i \mapsto \langle \mathcal{S}[\![e_1]\!]\phi \downarrow_{\text{SizeDep}^-}, \ldots\rangle]\\
&\qquad\qquad \sqcup \textstyle\bigsqcup_j \mathcal{S}[\![e_j]\!]\phi \downarrow_{\text{Gram}}\rangle\\
\mathcal{S}[\![e\ (e_1, \ldots, e_n)]\!]\phi &= \textstyle\bigsqcup\{\, \phi(l) \circ^\sharp \langle x_{l1}, \ldots, x_{lm}, \mathcal{S}[\![e_1]\!]\phi, \ldots \mathcal{S}[\![e_n]\!]\phi\rangle \mid\\
&\qquad\quad \langle l, \_, \_\rangle \in \alpha,\ l \neq \mathit{FO}, \mathit{arity}(l) = n\}\\
&\qquad \text{where } \begin{aligned}[t] (\alpha, G) &= \mathcal{S}[\![e]\!]\phi\\ \langle \alpha_{l1}, \ldots, \alpha_{lm}\rangle &= G(l)\\ x_{li} &= \mathit{normalize}(\alpha_{li}, G)\end{aligned}
\end{aligned}$$

Recall that the program has been lambda lifted prior to running the size analysis and that an abstraction appears in the program as the creation of a closure $\mathit{Clo}(f_i, \langle e_1, \ldots, e_n\rangle)$. 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 an application $e\ (e_1, \ldots, e_n)$ we compose each closure of the right arity e can evaluate to with a tuple describing the arguments and take the least upper bound of the results. The description of the free variables are found by first looking up the relevant entry in the grammar: α_{li} (a SizeDep$^-$ value) and then selecting the entries from the grammar that are sufficient for describing α_{li} (the normalization step).

Abstract Transition Semantics. The abstraction transition semantics, which collects abstract 1-step transitions and compositions of these, is given below.

$$
\begin{array}{l}
\text{AbsTrans} = \text{Fnames} \times \text{Fnames} \mapsto \mathcal{P}(\text{SizeDep}^*) \\
\mathcal{CS}_* : \text{AnnExp} \to \text{Fenv} \to \text{AbsTrans}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{CS}_{f_i}[\![\mathit{Clo}(f_j, \langle e_1, \ldots, e_n\rangle)]\!]\phi & = \bigsqcup_k \mathcal{CS}_{f_i}[\![e_k]\!]\phi \\
\mathcal{CS}_{f_i}[\![\underline{\mathit{Clo}}(f_j, \langle e_1, \ldots, e_n\rangle)]\!]\phi & = \bigsqcup_k \mathcal{CS}_{f_i}[\![e_k]\!]\phi \sqcup \\
& \quad [\langle f_i, f_j\rangle \mapsto \langle \mathcal{S}[\![e_1]\!]\phi, \ldots, \mathcal{S}[\![e_n]\!]\phi, \bot, \ldots, \bot\rangle] \\
\mathcal{CS}_{f_i}[\![e\ (e_1, \ldots, e_n)]\!]\phi & = \mathcal{CS}_{f_i}[\![e]\!]\phi \ \sqcup \ \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi \sqcup \\
& \quad [\langle f_i, f_j\rangle \mapsto \langle \tau_1, \ldots, \tau_m, \mathcal{S}[\![e_1]\!]\phi, \ldots, \mathcal{S}[\![e_n]\!]\phi\rangle \mid \\
& \quad\ \mathit{Clo}(f_j, \langle \tau_1, \ldots, \tau_m\rangle) \in \mathcal{S}[\![e]\!]\phi\downarrow_{\text{SizeDep}^-}, \\
& \quad\ \text{arity}(f_j) = n] \\
\mathcal{CS}_{f_i}[\![e \ \underline{@}\ (e_1, \ldots, e_n)]\!]\phi & = \mathcal{CS}_{f_i}[\![e]\!]\phi \ \sqcup \ \bigsqcup_j \mathcal{CS}_{f_i}[\![e_j]\!]\phi
\end{array}
$$

$$
\begin{array}{ll}
\phi_p & = \mathit{fix}\ \lambda\phi.[f_i \mapsto \mathcal{S}[\![e_i]\!]\phi \mid f_i(\ldots)(\ldots) = e_i \in p] \\
\mathit{1\text{-}atrans}_p & = \bigsqcup_i \mathcal{CS}_{f_i}[\![e]\!]\phi_p, \ \text{where } f_i(\ldots)(\ldots) = e \in p, \text{ and } f_i \text{ is reachable} \\
\mathit{atrans}_p & = \mathit{fix}\ \lambda T.(\ [\langle f_i, f_k\rangle \mapsto t_2 \circ^\sharp t_1 \mid [\langle f_i, f_j\rangle \mapsto t_1], [\langle f_j, f_k\rangle \mapsto t_2] \in T] \\
& \qquad \sqcup\ \mathit{1\text{-}atrans}_p)
\end{array}
$$

The non-trivial entries in the abstract semantics are those for application and the creation of closures. 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, whereas residual closure creation contribute at creation time but not when applied.

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 an abstract transition from the calling function to f_j in which the lambda-bound variables are described by $\bot$ (because they are dynamic).

Static application generates 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.

Termination Analysis. We use the result of the abstract transition semantics to determine which variables may be in situ increasing and which are guaranteed to be in situ decreasing: Given an endotransition $t \in \mathit{atrans}_p(f_i, f_i)$ we classify the variable $x_{i,j}$ as in situ increasing if $\exists\langle \_, s, d\rangle \in \pi_j(t),$ where $s = \text{I}$ and $j \in d$, and as in situ decreasing if $\forall\langle \_, s, \_\rangle \in \pi_j(t) : s = \bot, \ s = \text{D}(j), \ s = \text{D}(j\ f\ k),$ or $s = \text{E}(j\ f\ k)$ for some $f \in \text{Fnames}, k \in \{1, \ldots, \mathit{fv}(f)\}$.

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

8 Related Work, Conclusion, and Future Work

Related Work. Jones, Gomard and Sestoft [11] present a termination analysis for a flowchart language, and Jones and Glenstrup [9] 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.

Conclusion. We have extended the first-order analysis of Holst [10] 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[6]. 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.

Future Work. The techniques presented in this paper 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 datatypes 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.

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

Acknowledgements. Our thanks to Neil D. Jones and Arne Glenstrup for much needed sparring and helpful comments.

References

1. The Similix system, version 5.1. 1995.
2. 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.
3. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
4. Peter Holst Andersen and Carsten Kehler Holst. *Termination Analysis for Offline Partial Evaluation of a Higher Order Functional Language.* Technical Report, DIKU, University of Copenhagen, Denmark, 1996. To appear.
5. 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.
6. 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.
7. Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.
8. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order. Cambridge Mathematical Textbooks*, Cambridge University Press, 1990.
9. 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.
10. Carsten Kehler Holst. Finiteness analysis. In John Hughes, editor, *Functional Programming Languages and Computer Architectures*, pages 473–495, ACM, Springer-Verlag, Cambridge, Massachusetts, USA, August 1991.
11. 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).
12. 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.
13. H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Transaction of the AMS*, 89:25–59, 1953.
14. Peter Sestoft. *Replacing Function Parameters by Global Variables.* Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988. 107 pages.
15. 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, ACM Press, La Jolla, California, June 1995.

This article was processed using the LaTeX macro package with LLNCS style