PEPM'92

ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation

Fairmont Hotel, San Francisco, CA, USA
June 19-20, 1992

YALEU/DCS/RR-909

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Foreword

This volume contains the papers presented at the ACM SIGPLAN'92 Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'92), held in June 19-20, 1992 at Fairmont Hotel, San Francisco, California.

The program committee received 47 submissions, from which it selected 16 for presentation and publication. Although the submissions were not formally refereed, they were reviewed by the program committee with selected outside assistance, each paper being read by at least 3 reviewers. Originality, relevance, and clarity were the primary criteria for selection.

I wish to sincerely thank the Program Committee members and all the other referees for their excellent assistance in the refereeing process. I also gratefully acknowledge the support of ACM SIGPLAN.

Charles Consel
PEPM'92 Program Chair

Program Committee

Charles Consel (Yale University)
Kent Dybvig (Indiana University)
Chris Hankin (Imperial College)
Timothy Hickey (Brandeis University)
Neil Jones (University of Copenhagen)
Arun Lakhotia (University of Southwestern Louisiana)
John Launchbury (University of Glasgow)
Daniel Weise (Stanford University)

Speedup Analysis in Partial Evaluation: Preliminary Results

Lars Ole Andersen and Carsten Krogh Gomard
DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagn Ø, Denmark
E-mail: lars@diku.dk, gomard@diku.dk

Abstract

A partial evaluator *mix* is an automatic program optimization tool, but no penecea: Sometimes partial evaluation yields large speedup, othertimes it does not. In this paper we study *speedup* in partial evaluation.

It has for long been suspected that partial evaluation can do no better than *linear speedup.* We show that if *mix* is based on the techniques: constant folding, program point/function specialization, function unfolding and transition compression, then *superlinear speedup* is impossible.

It is usually impossible to predict the outcome of a specialization. We have developed a simple but pragmatically rather succesful *speedup analysis* which, when given a binding time annotated program, computes a *relative speedup interval.* Specialization of the program will result in a speedup belonging to the predicted interval. We give some experimental results, and discuss shortcomings and possible improvements.

1 Introduction

During the last decade *partial evaluation* has proven its usefulness as an efficient and automatic program optimization tool. Given a general program and parts of its input, a partial evaluator *mix* produces a *specialized* version. The aim is efficiency: The specialized version often run an order of magnitude faster than the general program. Despite many impressive results, partial evaluation is no panacea: Sometimes specialization pay off in form of a fast(er) program, othertimes it does not. However, the user without detailed knowledge of partial evaluation is usually unable to predict how much can be gained by specialization. In practice, the question: "is specialization of this program worthwhile?" must often be answered by actually applying *mix* and running of specialized version.

An informative answer to the above question is particularly desirable when partial evaluation is applied to computationally heavy problems. Examples with long execution times include specialization of scientific computation [3], ray tracing [12], and training of neural nets [9]. It is clearly wasteful to have a programmer specialize a program and let it run for hours just to realize that partial evaluation gave no benefit.

An *estimate* of the obtainable speedup, available before the specialization is done, would be valuable information. Then partial evaluation could be applied where ever *feasible* and not where ever *possible*, as is often the case. On the basic of a speedup estimate, the user could decide to rewrite her program in order to improve the prospective speedup (or she could just forget about it!) It would be logical to combine the speedup analysis with a *binding time debugger* as supplied with *e.g.* the Similix [4] and the Schism [6] systems. Another perspective would be to let the partial evaluator automatically generalize static computations[1] which are found not to contribute to a good speedup, the objective being smaller residual programs.

Estimation of speedup is clearly undecidable. We compute an estimate before the values of the static variables are given and use only *binding time information*, that is, knowledge about *which* computations that will be done at *mix*-time.

In this paper we study *speedup* in partial evaluation. It have for long time been suspected that partial evaluation can at most yield *linear* speedup. We prove that if *mix*, as described in *e.g.* [7], is based on the techniques: constant folding, function unfolding, transition compression and program point/function specialization, *superlinear* speedup is impossible.

We have developed a *speedup analysis* which, when given a binding time annotated program, approximates the speedup to be gained by partial evaluation by a *relative speedup interval* $[l, h]$. The interpretation is: the specialized version of the program will run at least l but atmost h times faster than the original program. If h is ∞, an unbounded speedup is possible. This can happen if *e.g.* the program contains a completely static loop where the number of iterations is determined by the static input.

Outline

In Section 2 we define the notions of *speedup* and *linear speedup*, and prove that *mix* cannot accomplish *superlinear* speedup. Section 3 contains description of a preliminary *speedup analysis.* In the succeeding section we discuss shortcomings and improvements.

Section 5 compares our work with related work, Section 6 gives directions for future work in this field, and Section 7 concludes the paper.

[1] That is, to reclassify to dynamic

2 Measuring and defining speedup

Let the *flow chart* of a program p be the directed graph where the nodes n_i are control points and the edges are transitions (jumps, calls, returns, ...). Let $\overline{v}$ be the values of the program variables. If a program execution in state $(n_i, \overline{v}_i)$ causes control to be passed to n_{i+1} with variable values $\overline{v}_{i+1}$, write $(n_i, \overline{v}_i) \rightarrow (n_{i+1}, \overline{v}_{i+1})$. A *(finite) computation* is a (finite) sequence:

$$(n_0, \overline{v}_0) \rightarrow (n_1, \overline{v}_1) \rightarrow \ldots$$

where n_0 is the initial program point and $\overline{v}_0$ is the program input. If we assume that the cost in terms of time is the same for all transitions, it is reasonable to define the *execution time* for a finite computation to be the length of the sequence.

Application of a program to its argument(s) is by juxtaposition and $|\cdot|$ denotes the execution time of a program run. We assume the input $\overline{v}_0 = (s, d)$ to consist of a static part s and a dynamic part d. p specialized with respect to s is written p_s. Thus for given p, s, d the speedup gained by partial evaluation is:

$$\frac{|p(s,d)|}{|p_s d|}$$

2.1 Linear speedup

From Jones[2] [10,11]:

DEFINITION 2.1 (LINEAR SPEEDUP) *Partial evaluation accomplishes* linear speedup *on p if for all s there exists an a such that for all but finitely many d*

$$a \leq \frac{|p(s,d)|}{|p_s d|}$$

□

Let for each s, a_s be the least upper bound (we shall see later that it does exist) of the possible values for a. Call a_s the speedup on p for s. A *largest* a does not always exist. Consider a program with a dynamically controlled loop with equally time consuming static and dynamic computations, and assume that *outside* the loop there is one dynamic statement (no static). Any $a < 2$ will work, but not $a = 2$. Still $a_s = 2$ seems the right choice for *the* speedup, since the computations outside the loop contribute little to the total run-times when the loop is iterated many times.

Jones posed the following as an open question [10,11]: "If *mix* uses only the techniques [program point specialization, constant folding, transition compression/unfolding] do there exist programs p on which *mix* accomplishes superlinear speedups?" Equivalently: does there exist an s for which a_s is not defined?

When s varies, a_s can grow unboundedly. A good example: in Consel & Danvy's derivation of the Knuth, Morris & Pratt algorithm (complexity $\mathcal{O}(m+n)$, m = pattern length, n = string length) from a naive matching algorithm (complexity $\mathcal{O}(mn)$) by specializing with respect to the pattern, we have $a_s = km$, k constant and hence linear speedup.

[2] Jones requires $a > 1$, but that is not likely to hold for all s (for example, take s to be empty) even for programs where a speedup is intuitively obtained.

If *mix* uses, say, reductions such as `(car (cons e1 e2))` $\Longrightarrow$ `e1` to discard "unnecessary" computations[3], introduces memoization, or elimination of repeated subexpressions `(e + e)` $\Longrightarrow$ `(let v = e in (v + v))` then it is easy to conceive examples of superlinear speedup.

We will show that by restricting partial evaluation to program point specialization, constant folding, and transition compression/unfolding the assumption that partial evaluation terminates can be used to place a bound on a_s. We thus provide a negative answer to the question of Jones [10,11].

2.2 No superlinear speedup!

Consider a finite computation for program p on input $\overline{v}_0$:

$$(n_0, \overline{v}_0) \rightarrow (n_1, \overline{v}_1) \rightarrow \ldots \rightarrow (n_h, \overline{v}_h)$$

Assume that each $\overline{v}_i$ has form (s_i, d_i). Input to the program is thus $\overline{v}_0 = (s_0, d_0)$. Each step, $(n_i, \overline{v}_i) \rightarrow (n_{i+1}, \overline{v}_{i+1})$ in the computation involves computing variable values $\overline{v}_{i+1}$ and a new control point n_{i+1}. Variable values depending only on n_i and s_i can be computed at partial evaluation time (= constant folding), and so can the shift of control to n_{i+1} (= transition compression or unfolding) when it is uniquely determined by n_i and s_i. (Beware that there may be other reasons than pure dependency to refrain from performing a compuation of either kind at partial evaluation time.) We shall call those computations that *are* performed at partial evaluation time *static* and those that are postponed *dynamic*.

Time is measured in computation steps, and for now we take time to do a jump = time to do an assignment = time to do an arithmetic operation = one step. To compute the speedup gained by partial evaluation for p, s_0, and d_0, consider the computation $(n_0, \overline{v}_0) \rightarrow \ldots$ above and simply sum the time spent on static (t_s) respectively dynamic (t_d) computations. We stress that we consider a standard computation and imagine: *If* this program had been partially evaluated with respect to s, *then* this would have been static and this would have been dynamic.

The potential speedup is:

$$\frac{t_s + t_d}{t_d}$$

Assume that partial evaluation of p on s terminates in K steps. Then in the standard computation there can be at most K static steps *with no intervening dynamic computations*. This, in turn, means that the speedup for p, s_0, and d_0 is bounded[4] by $2(K+1)$. This bound is independent of d_0 which rules out superlinear speedup.

3 Speedup analysis

We study languages representable as flow chart programs, as defined in the previous section. Candidates include Pascal, C *etc*, but the techniques are also applicable to functional languages.

Input to the analysis is a binding time annotated program, *i.e.* a program representation where all statements have been marked as being either static or dynamic. As our analysis is not given the *value* of the static variables, an exact determination of the speedup is not possible. We shall

[3] The computations may have influence on termination properties.
[4] Usually this will be a *very* loose bound.

approximate speedup by a *speedup interval* $[u, v] \subseteq \{x \in \mathbb{R} \mid x \geq 1\} \cup \{\infty\}$. A speedup interval for p should capture possible speedups for all s and d in a sense to be made precise below.[5]

3.1 Safety of speedup intervals

A speedup interval $[u, v]$ is safe for p if the speedup "converges" to an element in the interval as (s, d) are chosen such that $|p(s, d)| \to \infty$. Consider again the scenario from Section 2.1 and assume that the speedup is independent of the choice of s. Then a safe (and precise) speedup interval is [2,2]. In general, the speedup will not converge to a fixed x as $|p(s, d)| \to \infty$, but we shall require that all programs that run "long enough" shall exhibit a speedup arbitrarily close to the interval.

Definition 3.1 (Safety of speedup interval)
A speedup interval $[u, v]$ is safe for p if for all sequences $((s_i, d_i))$: $|p(s_i, d_i)| \to \infty$ implies

$$\forall \varepsilon : \exists k : \forall j > k : \frac{|p(s_j, d_j)|}{|p_{s_j} d_j|} \in [u - \varepsilon, v + \varepsilon]$$

□

3.2 Simple loop and relative speedup

Let a flow chart program with nodes n_i be given. A *loop* is a sequence of nodes

$$n_1 \to n_2 \to \cdots \to n_k, \quad \text{for } k \in \mathbb{N}$$

where $n_1 = n_k$. A *simple loop* is a loop $n_1 \to \cdots \to n_k$ where $n_i = n_j$, $1 \leq i < j \leq k$ implies $i = 1$ and $j = k$.

Assume given a function assigning for each statement (assignment, jump *etc*) an execution time.[6] Define for a node n_i the *cost* $\mathcal{C}(n_i)$ as the sum of the execution times of the statements in n_i. For notational convenience, we write $\mathcal{C}_s(n_i)$ for the cost of the *static* statements in n_i, and $\mathcal{C}_d(n_i)$ for the dynamic statement.

Definition 3.2 (Relative Speedup in Loop)
Let $l = n_1 \to \cdots \to n_k$ be a loop. The relative speedup *$\mathcal{SU}(l)$ in l is then defined by:*

$$\mathcal{SU}(l) = \begin{cases} \frac{\mathcal{C}_s(l) + \mathcal{C}_d(l)}{\mathcal{C}_d(l)} & \text{if } \mathcal{C}_d(l) \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

where $\mathcal{C}_s(l) = \sum_{i=1}^{k-1} \mathcal{C}_s(n_i)$ and $\mathcal{C}_d(l) = \sum_{i=1}^{k-1} \mathcal{C}_d(n_i)$. □

The relative speedup of a loop is a number in $\{x \in \mathbb{R} \mid x \geq 1\} \cup \{\infty\}$.

Relative speedup can be defined for a single basic block in the obvious way.

3.3 Doing speedup analysis

Let a flow chart program p with simple loops $\mathcal{L}$ be given. The basic idea behind the analysis is the observation that the relative speedup of the whole program is determined by the speedups of the loops. If the program run for a sufficient long time, it will spend most of its time inside loops.

[5]For addition involving ∞ we use $x + \infty = \infty + x = \infty$ for all $x \in \mathbb{R} \cup \{\infty\}$

[6]The analysis can be refined by letting the execution time function take evaluation of expressions into account

Algorithm For all simple loops $l \in \mathcal{L}$ in p, compute the relative speedup $\mathcal{SU}(l)$. The *relative speedup interval* is then the smallest interval $[u, v] \subseteq \{x \in \mathbb{R} \mid x \geq 1\} \cup \{\infty\}$ such that $\forall l \in \mathcal{L} : \mathcal{SU}(l) \in [u, v]$. □

To see that the speedups for all non-simple loops are also in the interval, let us see that if $[u, v]$ is safe for loops l_1 and l_2 then it is also safe for a loop l composed from l_1 and l_2. Assume w.l.o.g. that $\mathcal{SU}(l_1) \neq \infty$ and $\mathcal{SU}(l_2) \neq \infty$.

$$\begin{aligned} \mathcal{SU}(l) &= \frac{\mathcal{C}_s(l) + \mathcal{C}_d(l)}{\mathcal{C}_d(l)} \\ &= \frac{\mathcal{C}_s(l_1) + \mathcal{C}_s(l_2) + \mathcal{C}_d(l_1) + \mathcal{C}_d(l_2)}{\mathcal{C}_d(l_1) + \mathcal{C}_d(l_2)} \\ &= \frac{\left(\frac{\mathcal{C}_s(l_1) + \mathcal{C}_d(l_1)}{\mathcal{C}_d(l_1)}\right)}{\left(\frac{\mathcal{C}_d(l_1) + \mathcal{C}_d(l_2)}{\mathcal{C}_d(l_1)}\right)} + \frac{\left(\frac{\mathcal{C}_s(l_2) + \mathcal{C}_d(l_2)}{\mathcal{C}_d(l_2)}\right)}{\left(\frac{\mathcal{C}_d(l_1) + \mathcal{C}_d(l_2)}{\mathcal{C}_d(l_2)}\right)} \\ &= \frac{\mathcal{SU}(l_1)}{\left(\frac{\mathcal{C}_d(l_1) + \mathcal{C}_d(l_2)}{\mathcal{C}_d(l_1)}\right)} + \frac{\mathcal{SU}(l_2)}{\left(\frac{\mathcal{C}_d(l_1) + \mathcal{C}_d(l_2)}{\mathcal{C}_d(l_2)}\right)} \end{aligned}$$

Assuming $\mathcal{SU}(l_1) \leq \mathcal{SU}(l_2)$ it is easy to prove

$$\mathcal{SU}(l_1) \leq \frac{\mathcal{SU}(l_1)}{\left(\frac{\mathcal{C}_d(l_1) + \mathcal{C}_d(l_2)}{\mathcal{C}_d(l_1)}\right)} + \frac{\mathcal{SU}(l_2)}{\left(\frac{\mathcal{C}_d(l_1) + \mathcal{C}_d(l_2)}{\mathcal{C}_d(l_2)}\right)} \leq \mathcal{SU}(l_2)$$

The speedup analysis does not take basic blocks outside loops into account. Clearly, the speedup of the loops will dominate the speedup of the whole program provided the execution time is large. However, the analysis can easily be modified to handle the remaining basic blocks by accumulating relative speedups for all path through the program without entering loops. Without the revision, the analysis will have nothing meaningfull to say about programs without loops.

Example 3.1 Consider the follwing trivial program which implements addition.

```
int add(int m, int n)
{
    int sum;
    1: sum = n;
    2: if (m) goto 3; else goto 6;
    3:     sum += 1;
    4:     m -= 1;
    5:     goto 2;
    6: return sum;
}
```

The basic blocks are: $\{1\}, \{2, 3, 4, 5\}, \{6\}$ where the second constitute a (simple) loop. Suppose that m is static but n dynamic. Then the statements 2, 4 and 5 are static and the rest dynamic. For simplicity, count 1 time-unit for each statement.

The relative speedup of the loop is 4. Hence, the approximated relative speedup of the whole program is $[4, 4]$.
End of Example

Theorem 3.1 (Safety of the Analysis)
Assume speedup analysis computes the speedup interval $[u, v]$ for program p. Then $[u, v]$ is safe for p.

Proof *An upper bound $v = \infty$ is trivially safe, so we assume $v \neq \infty$.*

Consider the sequence of nodes n_i visited during a computation c arisen from the application of program p to data (s,d).

$$n_1 \rightarrow n_2 \rightarrow \cdots \rightarrow n_k$$

To delete *a simple loop $n_i \rightarrow n_{i+1} \rightarrow \cdots \rightarrow n_{i+j}$ from c is to replace c by:*

$$n_1 \rightarrow \cdots \rightarrow n_{i-1} \rightarrow n_{i+j+1} \cdots \rightarrow n_k$$

Now delete as many simple loops as possible from c. Denote the multiset of deleted loops by $\mathcal{L}$. The nodes remaining in c now occur only once. The number of nodes in p provides a uniform bound on the number of remaining nodes, independent of the choice of (s,d). Denote the set of remaining nodes by NL *and define* nlstat $= \sum_{n \in \mathrm{NL}} C_s(n)$ *and* nldyn $= \sum_{n \in \mathrm{NL}} C_d(n)$*. Define the cost functions C_s and C_d on a* multiset *$\mathcal{L}$ of loops to be the sum of the costs for each loop $l \in \mathcal{L}$.*

We now calculate the speedup for p, s, d:

$$\frac{C_s(\mathcal{L}) + \text{nlstat} + C_d(\mathcal{L}) + \text{nldyn}}{C_d(\mathcal{L}) + \text{nldyn}}$$

This expression can be rewritten to the following (exactly as in Section 3.3):

$$SU = \frac{\left(\frac{C_s(\mathcal{L})+C_d(\mathcal{L})}{C_d(\mathcal{L})}\right)}{\left(\frac{C_d(\mathcal{L})+\text{nldyn}}{C_d(\mathcal{L})}\right)} + \frac{\left(\frac{\text{nlstat}+\text{nldyn}}{\text{nldyn}}\right)}{\left(\frac{C_d(\mathcal{L})+\text{nldyn}}{\text{nldyn}}\right)}$$

Now we will argue that for all $\varepsilon > 0$ there exists a K such that $SU \in [u-\varepsilon, v+\varepsilon]$ if only $|p(s,d)| > K$. Choose a sequence of (s_i, d_i) such that $|p(s_i,d_i)| \rightarrow \infty$ and examine the fractions.

To the right of the +, the numerator $\frac{\text{nlstat}+\text{nldyn}}{\text{nldyn}}$ is uniformly bounded and the denominator $\frac{C_d(\mathcal{L})+\text{nldyn}}{\text{nldyn}} \rightarrow \infty$ (recall that $C_d(\mathcal{L}) \rightarrow \infty$ since we assumed $v \neq \infty$).

To the left of the +, the denominator $\frac{C_d(\mathcal{L})+\text{nldyn}}{C_d(\mathcal{L})} \rightarrow 1$ so we conclude $SU \rightarrow \frac{C_s(\mathcal{L})+C_d(\mathcal{L})}{C_d(\mathcal{L})}$.

By the argument in Section 3.3, $\frac{C_s(\mathcal{L})+C_d(\mathcal{L})}{C_d(\mathcal{L})} \in [u,v]$ which concludes the proof. □

3.4 Experiments

We have implemented the speedup analysis as part of the C-Mix system, a partial evaluator for a subset of C [1,2]. The implemented version solely considers loops, and has a differentiated cost function for statements, but takes evaluation of expressions to take constant time. The analysis is fast; there is no significant execution time for the examples presented here.

The analysis has been applied to a number of programs. All experiments have been performed on a Sun SparcStation, and times measured via the UNIX `time` command (user seconds). Hence, we believe the reported figures provides a fair view of the reality. Note that the programs `Int` and `Scanner` are from [13] and thus *not* specially "cooked" for this paper.

Below the measured and estimated speedups for three different programs are shown. The `Add` program is given in Example 3.1 above. The `Int` program is an interpreter for a "polish-form" language stolen from [13]. In this example, the static input was a program computing the first n primes, and the dynamic input was $n = 500$. The program `Scanner` is a general lexical analysis taking as input a scanner table (static input) and a stream of characters (dynamic input). In the test run, it was given a specification of 8 different tokens which appeared 30000 times in the input stream.

Example	Run-time Src	Run-time Res	Speedup Measured	Speedup Estimated
Add	12.2	4.6	2.7	[2.7, 2.7]
Int	59.1	8.7	6.8	[5.1, ∞]
Scanner	1.5	0.9	1.7	[1.5, 4.1]

Assessment For the `Add` program the speedup factor is *independent* of the dynamic input and converges to 2.7 as the static input grows. Hence the very tight interval.

The upper bound for `Int` is corretly ∞ as the interpreter's code for handling unconditional jumps is completely static:

```
while (program[pp] != HALT)
   switch (program[pp])
      {
      case ...
      case JUMP:  pp = program[pp+1]; break;
      case ...
      }
```

Thus, an unboundedly high speedup can be obtained by specializing `Int` with respect to a program with "sufficiently" many unconditional jumps. We provide an example below. Note that the speedup, of course, is bounded by the length of the program, but as the analysis is performed *before* the static values are known, it must err in the most conservative way and report "unbounded speedup".

The interval for the `Scanner` is also satisfactory. Is a specification of unambiguous tokes given, very litte can be done at *mix*-time, and thus a low speedup. On the other hand, if the supplied table contains many "fail and backtrack" actions, the upper bound can be approached. □

To demonstrate that the seemingly non-tight speedup intervals computed by the analysis are indeed reasonable, we have applied the "polish-form" interpreter to three different programs, *i.e.* three different static inputs. Each program exploits different parts of the interpreter. Recall that the computed relative speedup interval for `Int` is $[5.1, \infty]$.

The `Primes` program is the program computing the first n primes. The `Addp` program is the equvivalent to the program `Add` in Example 3.1, but in "polish-form". The `Jump` program consists of a single loop with ten unconditional jumps (see below). The measured speedups are as follows.

Example	Run-time Src	Run-time Res	Speedup Measured
Primes	59.1	8.7	6.8
Addp	51.5	5.5	9.2
Jump	60.7	3.0	20.3

Assessment These experiments clearly demonstrate that the actual speedup *does* depend on the static input as previously claimed.

The `Primes` program contains mainly aritmetic operations, and hence, by specializing the interpreter to it, nearly nothing but the "pure" interpretation overhead can be removed. The lower bound 5.1 could be further approached by specializing the interpreter to a program containing nothing but operations depending solely on dynamic input (get, add *etc*).

In the case of the Add program, the static "book-keeping" in the interpreter makes up a greater part of the total execution time than for the Primes program. Hence the somewhat larger speedup.

The final example, Jump indicates that an arbitrarily high speedup is possible. The program is shown below (in a cooked, more readable syntax).

```
/* Jump */
read n;
loop
      exitif n = 0;
      jump 1;
   1: jump 2;
   2: jump 3;
   3: jump 4;
   4: jump 5;
   5: jump 6;
   6: jump 7;
   7: jump 8;
   8: jump 9;
   9: jump 10;
  10: n = n - 1;
end;
```

Since all the jumps $1 \rightarrow 2 \cdots \rightarrow 10$ can be done at specialization time ("transition compression", cf. the fragment of the interpreter shown above), the high speedup is obtained. Clearly, when the concrete program is unavailable to the speedup analysis, it cannot give an upper bound for the speedup. □.

4 Extensions and improvements

Even though the speedup analysis presented in the previous section has demonstrated some pragmatic success, it is also infested with severe drawbacks. We have found that the lower bound computed by the analysis usually provides a fairly good estimate, but it is easy to construct examples which "shake" the analysis. Consider for example the program fragments below.

```
n = N;                          n = N;
while (n)                       while (n)
   { S1; S2; n--; }                { S1; n--; }
                                n = N;
                                while (n)
                                   { S2; n--; }
```

Suppose that S1 (static) and S2 (dynamic) do not interfere meaning the two programs have the same effect. For the program to the left, the estimated speedup interval is $[4, 4]$ (counting 1 for all kinds of statements). The corresponding interval for the program to the right is $[3, \infty]$, where ∞ is due to the completely static loop. The latter result is still *safe* but less tight than the former.

The problem is that the analysis considers loops in isolation, and fails to recognize that the two loops iterate the same number of times. (Notice that even though the relative speedups and the residual programs are equal, the *specialization time* for the former is only half as large as that for the latter.)

4.1 Relating loops

As exemplified above, a major shortcoming in the analysis is that all loops are treated as being completely unrelated. If it was known that two loops iterate the same number of times, then their bodies could be treated as one. This would capture the above example but it is only too easy to construct new and worse "counterexamples". Another blemish in the method is that all loops contribute equally to the final approximation of the speedup. For example, in a program with two loops, the one iterated 2 times, speedup 2, the other iterated 1000 times, speedup 10, the actual speedup is clearly close to 10. The speedup analysis would report the safe but loose speedup interval $[2, 10]$.

A tempting idea is to relate the number of times a (simple) loop is iterated with its relative speedup. One can compute approximations in the form of an interval $[l, h]$ meaning: the loop iterates at least l times but at most h times. Here ∞ must be included with the meaning "unknown" number of iterations.

An advantage of this approach is that bounds on completely static loops might be detectable thus ruling out some sources of seemingly unbounded speedup. For instance, in the example program to the right, the static loop cannot give unbounded speedup as it is limited by N. This approach would not, however, capture *e.g.* the unbounded speedup in the "polish-form" interpreter, as it depends truly on the static input.

4.2 Making use of static values

As the obtainable speedup depends on the static input, it is a possibility to take the *values* of these into account to give a more precise speedup estimation. Either the *concrete* values or a *description* of the static input could be considered. We consider the latter in the next section, as this is the procedure in automatic complexity analysis.

Often the time for specialization of a program is far below the execution time of both the original and residual program. When this is the case, analysis of the residual program seems to be a reasonable way to proceed. An immediate advantage hereby is, that since the static loops have been unrolled, an upper bound for the speedup can always be computed, that is, no unbounded speedup. Further, *mix*-introduced optimizations such as arity arising, unfolding *etc*, can be dealt with in an easy way. Beware, a precise speedup still cannot be computed: The residual program may (dynamically) choose branches with different speedups.

A way to do speedup analysis of the residual program is to modify the partial evaluator to output "profiling" information into the residual program. More concretely, it could mark in the residual program where static computations had been performed. Then the speedup analysis described in Section 3 could be applied almost unmodified.

4.3 Analyzing speedup in functions

The presented analysis gives as result a speedup estimate for programs as a whole. In some cases it may also be useful to have a speedup estimate for each function. For example, "binding-time-crafting" could then be brought into action on functions with a low speedup, or a user could decide to specialize only functions with a high speedup estimate.

The analysis can easily be modified to support speedup estimation of functions. Consider a call statement as a basic unit with an associated cost. The flow-chart representation of a program then consists of a number of sub-graphs, one for each function. Analyze each sub-graph individually. A speedup estimate for the whole program can be given by accumulating the speedups for all (called) functions.

The speedup estimate is likely to be more coarse than before. The reason is that many small "very" static or "very" dynamic loops may turn up. On the other hand, a speedup estimation attached to each function may prevent a completely static "initialization" loop in a "main" function to distort the result of the global analysis.

4.4 Extension to other languages

In a straightforward manner, the analysis can be changed to analyze functional languages. For analyzing higher order languages, the trace of the control-flow becomes more complicated, but it can be handled by well-known techniques. We will not dwell on this here.

5 Related work

To the best of our knowledge, this is the first humble attempt at automatic estimation of speedup in partial evaluation.

5.1 Speedup in partial evaluation

Jones defined the notion of a partial evaluator accomplishing linear speedup [10,11], observing that the speedup may depend on the static input. He posed as an open problem whether it is impossible for *mix* to accomplish ***super-linear*** speedup, when it is based on the basic principles function specialization, constant folding, and unfolding of functions. We have given an affirmative answer to his problem in this paper.

As a curiosity, immediately before Jones stated the definition of linear speedup, Consel and Danvy made an interesting experiment [5]. They specialized a general pattern matching program and obtained the efficient Knuth, Morris & Pratt algorithm. It was found that the speedup was exactly the length of the static input string. Note that the same tendency is present in the "polish-form" interpreter.

Amtoft [8] proves in the setting of logic programming that fold/unfold transformations at most can give rise to linear speedup. The same restrictions as imposed upon *mix* are necessary. Note, however, that unification is daringly assumed to run in constant time.

5.2 Speedup analysis vs. complexity analysis

Automatic complexity analysis has received some attention during the last years. The aim is: given a program p and possibly some "size" descriptions of the input, to compute a *worst-case* complexity function $\mathcal{O}_p(\cdot)$ in terms of the input sizes n_i.

It is tempting to apply the techniques from automatic complexity analysis to speedup estimation. However, as the following example illustrate, the linear speedup in partial evaluation does not fit well into ordinary complexity analysis. Consider the program below.

```
while (n)
  {  S1; S2; ... Sj;  n = n - 1;  }
```

Assume the relative speedup obtained by specialization of the statement sequence `S1; ...Sj;` to be k. Then the relative speedup of the whole program will be approximately k regardless of the loop being static or dynamic. However, if the loop is static, complexity analysis of the residual program will produce the answer $\mathcal{O}(1)$, since the loop has been unrolled. If the loop is dynamic, the result will be $\mathcal{O}(n)$. Not much insight (about speedup, that is) is gained this way.

It is, however, most likely that the techniques from automatic complexity analysis can be adapted to aid the problem of speedup analysis.

6 Future work

There are several possible directions for continuation of the preliminary results presented in this paper. We believe that automatic speedup estimation is an emerging field, which will become more important as larger and more practically applicable partial evaluation systems are built.

Still further experiments may be helpful to figure out what kind of information is profitable for users of partial evaluators. An estimate of the loop-iterations appears to be a fruitful kind of information. It can provide the user with a tighter speedup estimate, but there are also other application areas.

Specialization of certain kind of programs have a bad habit of producing huge residual programs, cf. *e.g.* [9]. This may be inconvenient for several reasons, for example the underlying compiler may be exhausted. This is a general problem with machine produced programs. An *estimate* of the *size* of the residual program could give the programmer the opportunity to generalize a loop to prevent unrolling.

Since the size of the residual program to a great extent is determined by the static loops unrolled, loop-iteration estimates appear to be the needed tool. Function unfolding, splitting of data structures *etc*, may also have an impact, though.

Let p be a program and p_{res} a specialized version of p. Define the *code blowup* as the relation $\frac{\|p_{res}\|}{\|p\|}$ where $\| p \|$ denotes the "length" of p. There is no obvious relaxation between the speedup $\frac{|p(s,d)|}{|p_s d|}$ and the code blowup. Clearly, if is the speedup is large but the code blowup low, a lot is gained at a low price. A linear relation between the speedup and the code blowup appears to be acceptable: the speedup must be accepted to have a certain price. Can superlinear code blowup be allowed? No clear distinction seems to exist which supposedly makes automation of generalization with respect to code size impossible.

A general problem in this work is the lack of an underlying theoretical foundation, that is, a complexity theory for *linear* speedups. We have been able to answer the problem of superlinear speedups without such a theory, but a classification of for examples the speedup in interpreters would be helpful. This is still an open problem. Further, a closer connection between *safe* optimizations and superlinear speedup is still needed. As we saw, by discarding computations or introducing sharing, superlinear speedups *can* be achieved, but the last word about cause and effect is still to be said.

7 Conclusion

We have investigated *speedup* in partial evaluation. It was showed, that if *mix* is based on the techniques: function specialization, function unfolding/transition compression and constant folding, ***superlinear*** speedup is impossible.

A simple, but useful ***speedup analysis*** has been developed and implemented. The basic principle is that the speedup of the whole program is determined by the speedup in the loops. We gave some experiments which showed reasonable results, but we also pointed out that the analysis can fail miserably on nasty programs.

We believe speedup estimation is a valuable information for users of a partial evaluator systems, and hence, further investigation of this field should be undertaken.

References

[1] L.O. Andersen. C program specialization. Technical report, DIKU, University of Copenhagen, Denmark, 1992.

[2] L.O. Andersen. Self-applicable C program specialization. In *Proceeding of PEPM'92: Partial Evaluation and Semantics-Based Program Manipulation*, 1992.

[3] A.A. Berlin. Partial evaluation applied to numerical computation. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 139–150. ACM, 1990.

[4] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

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

[6] Charles Consel. *The Schism Manual, Version 1.0.* Yale University, New Haven, Connecticut, December 1990.

[7] C.K. Gomard and N.D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.

[8] T.A. Hansen. Properties of unfolding-based meta-level systems. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 243–254. ACM Press, 1991.

[9] H.F. Jacobsen. Speeding up the back-propagation algorithm by partial evaluation. DIKU Student Project 90-10-13, 32 pages. DIKU, University of Copenhagen. (In Danish), October 1990.

[10] N.D. Jones. Computer impelementation and application of kleene's s-m-n and recursion theorems. In Y.N. Moschovakis, editor, *Logic from Computer Science*, pages 243–263. Springer-Verlag, 1989.

[11] N.D. Jones. Partial evaluation, self-application and types. In M.S. Paterson, editor, *Automata, Languages and Programming. 17th International Colloquium, Warwick, England. (Lecture Notes in Computer Science, vol. 443)*, pages 639–659. Springer-Verlag, 1990.

[12] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[13] F.G. Pagan. *Partial Computation and the Construction of Language Processors.* Prentice-Hall, 1990.

Predicting Properties of Residual Programs

Karoline Malmkjær [*]
Department of Computing and Information Sciences
Kansas State University [†]

Abstract

The definition of a partial evaluator determines which input-output function will be computed by a residual program. For many applications, however, we would like a similar guarantee on *how* the residual programs compute this function. This paper presents the prototype of a tool for predicting the structure of residual programs based on the partial evaluator and (pieces of) the source program. The result is in the form of a grammar describing the possible residual programs. The technique is based on abstract interpretation.

1 Intensions and extensions of a partial evaluator

The standard definition of a partial evaluator is purely extensional, stating that the residual program will compute the "right" (in effect — the specialized) function on the dynamic data. This "correctness by construction" is the motivation behind most applications of partial evaluation. It is most explicit in the use of partial evaluation to generate correct compilers from definitional interpreters (as specified by the Futamura projections [Futamura 71]). But it is also essential to applications such as the specialization of string matchers by Consel and Danvy. By generating residual programs with the structure of, respectively, the Knuth-Morris-Pratt failure tables and the Boyer-Moore algorithm from the same source program with only simple modifications of the primitives, partial evaluation provides insight into the relation between these two otherwise distinct algorithms as well as an independent proof of their correctness [Consel & Danvy 89, Danvy 91].

In most practical applications, however, the partial evaluator is expected to have specific intensional properties as well — it should perform static reductions, so that only certain pieces of the source program are reproduced in the residual program [Jones *et al.* 89].

So when using a partial evaluator to compile, the partial evaluator is expected to perform the traditional "compile-time" actions. For example, a compiled program should not contain any references to a compile-time environment, but should contain direct accesses to the run-time store. This is expected to be a general property, that is, for a specific interpreter *all* the compiled programs should access the store directly.

[*]supported by DIKU, Computer Science Department, University of Copenhagen, Denmark
[†]Manhattan, Kansas 66506, USA. e-mail: karoline@cis.ksu.edu

In the string matching example, on the other hand, we do not have any preconceived notion of "how much" the partial evaluator should do. Clearly it should do "as much as possible", but there is no previously established reason why this should lead exactly to the structure of the KMP failure tables. So again, in order for the results about string matching to have validity as a theoretical tool for reasoning about hierarchies of algorithms, it must be established that the residual programs have the structure of the KMP failure tables, resp. the Boyer-Moore algorithm, for *all* legal static data.

Such results about families of residual programs might of course be proven for each individual source program by a proof over the entire partial evaluator. Such proofs would be considerably simpler when the partial evaluator is an off-line partial evaluator (that is, it has a separate binding time analysis), since the binding time annotated program could then be used as a starting point. Then a proof would only need to involve the actual core specializer. But it would still be a rather large and tedious task, and a large number of similar steps would have to be repeated for every source program one wished to prove properties from.

Thus it seems that there is a need for a generic tool, that could be proven once and for all and that produces this kind of information about residual programs.

2 Grammars describing residual programs

We have developed a technique for predicting properties of residual programs. From the partial evaluator and a source program we generate a context-free grammar describing the possible residual programs.

Together, a partial evaluator and a source program clearly define a set of residual programs (and possibly some error messages and infinite loops) corresponding to all possible static data. If we consider this set to be a kind of language, the choice of a grammar as an abstract representation comes quite naturally.

This is inspired by Jones [Jones 87], who proposes the use of grammars in flow analysis to represent potentially infinite structures, and Mogensen [Mogensen 88], who uses a grammar representation of compound structures to describe partially static structures.

With this approach, a non-terminal is generated for each procedure and possibly for each formal parameter in the program being analyzed. The non-terminals are intended to represent the result of procedure applications and identifier references. This way any recursive references in the program will appear as recursive productions in the grammar.

Since an essential part of this project is that it must be proven correct, we base ourselves on abstract interpretation.

In order to keep the abstraction — and thus the proofs — simple, we use the following observation: the generating extension of a source program (that is, the result of specializing the partial evaluator with respect to the source program) is a program that maps static data to residual programs [Ershov 78]. This implies that an abstract representation of all possible output of the generating extension is also an abstract representation of all residual programs corresponding to this source program.

So, we design an abstract interpretation of the target language of the partial evaluator, which is, of course, the language in which the generating extensions are written. This abstract interpretation produces an abstract representation of the output of a program in the form of a grammar. So when we use it on a generating extension, we get a grammar representation of the family of residual programs.

This approach implies that we only have to prove the abstraction of the standard semantics of the target language, instead of proving an abstraction of the entire partial evaluator. Also, our approach will in principle work for any partial evaluator with the same target language. On the other hand, it requires the partial evaluator to be self-applicable and each abstract result will correspond to a fixed division of the input into static and dynamic.

We have proven [Malmkjær 91] the safety of our analysis (all concrete results must be in the language generated by the grammar) using logical relations [Jones & Nielson 91].

As a practical experiment, we have implemented a prototype of the analysis for the Similix specializer [Bondorf & Danvy 91, Bondorf 91]. The prototype uses the abstract syntax trees generated by Similix before postprocessing. In order to represent the call structure of the residual programs, it generates non-terminals corresponding to residual procedures as well as procedures in the program being analyzed.

To facilitate a proof by logical relations, the grammars are generated by local fixed points instead of a global fixed point over the entire program. Once a nonterminal corresponding to a procedure has been generated, the corresponding production is computed as a fixed point over the right hand side of the production. At each step, the right hand side is replaced by the non-terminal and the grammar containing the production from the non-terminal to the right hand side. Since the non-terminals are never dereferenced, this ensures termination.

3 Example

As an example of the kind of results we can get with the current implementation, we consider the following piece of a source program for Similix:

```
(define (evalExpression E env)
   (cond
      [(isConstant? E) (constant-value E)]
      [(isVariable? E) (lookup-store (lookup-env (E->V E) env))]
      [(isPrim? E)
       (let ([op (E->operator E)])
          (cond
             [(is-cons? op) (cons (evalExpression (E->E1 E) env)
                                  (evalExpression (E->E2 E) env))]
             [(is-equal? op) (equal? (evalExpression (E->E1 E) env)
                                     (evalExpression (E->E2 E) env))]
             [(is-car? op) (car (evalExpression (E->E E) env))]
             [(is-cdr? op) (cdr (evalExpression (E->E E) env))]
             [(is-atom? op) (atom? (evalExpression (E->E E) env))]
             [else "Unknown operator"]))]
      [else "Unknown expression form"]))
```

The experienced reader will recognize this as a part of the MP interpreter that is distributed with Similix. This toy interpreter is used to demonstrate compiler generation by partial evaluation. We have chosen it as an example since it was written by someone else without any consideration for the present analysis.

The analysis determines that the specialized versions of this piece of code — that is, the compiled MP expressions — will be members of the language described by the following grammar:

```
sim-process-expr --> _|_
                  | (expr (const "Unknown expression form"))
                  | (expr (const "Unknown operator"))
                  | (expr (primop "atom?" sim-process-expr))
                  | (expr (primop "cdr" sim-process-expr))
                  | (expr (primop "car" sim-process-expr))
                  | (expr (primop "equal?" sim-process-expr sim-process-expr))
                  | (expr (primop "cons" sim-process-expr sim-process-expr))
                  | (expr (primop "lookup-store" (expr (const Val))))
                  | (expr (const Val))
```

Except for renaming the non-terminal to `sim-process-expr` for readability, the grammar productions appear exactly as produced by the analysis. $\perp$ represents possible errors during partial evaluation. `Val` represents any value, so the form `(const Val)` represents a constant in the residual program, the value of which cannot be determined by the analysis. Note that the grammar shows us that syntax errors will be reported at run-time rather than at compile-time. This is consistent with the semantics of the interpreter, which may terminate normally on input that does not cause the part with the syntax error to be executed.

The remaining productions correspond to the compilation of well-formed expressions. In particular we notice that the environment lookup has disappeared, whereas the store lookup remains.

We have run the analysis on the entire MP interpreter and the resulting grammar shows that, as intended, there will be no references to the environment in the compiled programs.

The results of this analysis could be combined with one or more postprocessing steps that could give more specific information automatically. A very simple example for Similix would be a step that could make a specialized "run-time package" of adt files, since the grammar can often tell which of the user-defined primitives are used by the residual programs.

4 Assessment

Although the analysis shows satisfactory results on several examples, there is still room for improvement in other cases.

At present higher-order values are handled in a trivial way, causing loss of information in programs that rely to any significant extent on higher-order programming. We plan to improve on this by adding a closure analysis.

We also plan to plan to improve the precision of parameter passing and perhaps to extend the analysis with some facility to trace back the unknown values in the output to the static data. Thus, for some of the unknown constants appearing in the residual program, the analysis may be able to determine that, although the constant is unknown, it corresponds to, *e.g.*, the primitive "length" applied to the second static argument.

It is likely that this analysis could be "pulled back" over the partial evaluator, so that we would analyze the binding time annotated source programs directly, instead of the generating extension. This should give quite similar results, but it would be much more complicated to prove that the analysis was consistent with the partial evaluator, since the proof would have to involve the behavior of the partial evaluator instead of just the standard semantics of the target language.

Alternatively, an analysis directly on binding time analyzed programs could be proven correct by generating it from the present analysis by partial evaluation. Since the present analysis works on the generating extension, it can be described as a function on the partial evaluator and the source program that first produces a generating extension and then analyzes it. If we denote the analyzer of generating extensions by A and the partial evaluator by PE and the functions they compute by $\mathcal{A}$ and by $\mathcal{PE}$, we can describe the composed function as a mapping from a partial evaluator and a binding time analyzed source program to a grammar:

$$(PE, p) \mapsto \mathcal{A}(\mathcal{PE}(PE, p))$$

Clearly this composed function can be specialized with respect to its first argument:

$$\mathcal{PE}(A \circ PE, PE)$$

The specialized composed function will be a program that takes a binding time analyzed program and produces a grammar representation of the possible residual programs. Note, however, that this is only an extensional achievement. To obtain non-trivial results from this specialization, we would need a more powerful partial evaluator than the current state of the art provides, since the generating extension is produced under dynamic control. Intensionally, the resulting analyzer would in fact operate by first producing the generating extension and then analyzing it.

It would also be possible to specialize the composed function by hand, though, in which case we have a much richer collection of meaning-preserving transformations at our disposal. Thus it may be possible to derive an analyzer operating directly on the binding time analyzed program in a non-trivial way by transforming the current analyzer. If we only use meaning-preserving transformations and if we have proven that the program PE is indeed a partial evaluator, the correctness of such an analysis will rely on the correctness of the current generating extension analyzer.

To the best of our knowledge, there has been no similar work in partial evaluation. It is still an open question how well the technique will adapt to other partial evaluators than Similix.

5 Conclusion

We have succeeded to develop a tool that determines some of the properties of residual programs in a generic way. We have proven that the analysis is safe and implemented a prototype, as reported here. The prototype is still relatively simple but appears to be an appropriate basis for several extensions that will produce considerably more precise results.

Currently the partial evaluation programmer writes a source program with a particular intention of which parts should be reduced — an intention that corresponds to the programmers experience with how the partial evaluator works. As the partial evaluation technology develops, however, it becomes increasingly more powerful but also increasingly more complex. This implies that the connections between the source program and the possible residual programs become less and less transparent. Correspondingly, the need for automatic and safe tools for determining these connections is likely to grow. The present approach has the advantage that the analysis will not grow more complex with the partial evaluator, as it relies only on the target language.

References

[Bondorf & Danvy 91] Bondorf, Anders & Olivier Danvy: "Automatic autoprojection of recursive equations with global variables and abstract data types", *Science of Computer Programming*, vol 16, pp 151-195 (1991)

[Bondorf 91] Bondorf, Anders: "Automatic autoprojection of higher order recursive equations", *Science of Computer Programming*, vol 17, pp 3-34 (December 1991)

[Consel & Danvy 89] Consel, Charles & Olivier Danvy: "Partial Evaluation of Pattern Matching in Strings", *Information Processing Letters*, vol 30, pp 79-86 (January 1989)

[Danvy 91] Danvy, Olivier: "Semantics-Directed Compilation of Nonlinear Patterns", *Information Processing Letters*, vol 37, no 6, pp 315-322 (March 1991)

[Ershov 78] Ershov, Andrei P.: "On the Essence of Compilation", in *Formal Description of Programming Concepts*, E.J. Neuhold (ed.), pp 391-420, North-Holland (1978)

[Futamura 71] Futamura, Yoshihito: "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler", *Systems, Computers, Controls*, vol 2, no 5, pp 45-50 (1971)

[Jones 87] Jones, Neil D.: "Flow analysis of lazy higher-order functional programs", in *Abstract Interpretation of Declarative Languages*, S. Abramsky & C. Hankin (eds), pp 103-122, Ellis Horwood, Chichester (1987)

[Jones *et al.* 89] Neil D. Jones, Peter Sestoft & Harald Søndergaard: "Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation", *Lisp and Symbolic Computation*, vol 2, no 1, pp 9-50 (1989)

[Jones & Nielson 91] Neil D. Jones & Flemming Nielson: "Abstract Interpretation: a Semantics-Based Tool for Program Analysis", invited paper (in preparation) for *The Handbook of Logic in Computer Science*, North-Holland (1991)

[Malmkjær 91] Karoline Malmkjær: "Predicting Properties of Specialized Programs", PhD Proposal, Kansas State University (November 1991)

[Mogensen 88] Mogensen, Torben Æ.: "Partially Static Structures in a Self-Applicable Partial Evaluator", in *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov & N. D. Jones (eds.), pp 325-347, North-Holland (1988)

Semantical Interprocedural Analysis by Partial Symbolic Evaluation

Babak Dehbonei [1]
Pierre Jouvelot [2,3]

[1]Bull Corporate Research Center, France
[2]Ecole des Mines de Paris, France
[3]MIT Laboratory for Computer Science, USA
{dehbonei,jouvelot}@ensmp.fr

Abstract

Partial symbolic evaluation is the basis of a new general formalism for the semantical interprocedural analysis of imperative programs. This paper introduces this abstract interpretation framework and presents two of its applications for (1) interprocedural constant propagation and (2) interprocedural side effect analysis. Our approach thus allows a clean unification of otherwise quite different data-flow analysis methods. Besides its theoretical appeal, this technique also permits multiple analysis to be efficiently performed in one single pass; this was not achievable with previously published techniques.

Partial symbolic evaluation is based upon a symbolic evaluator of programs ; its partial aspect comes from the fact that the choice of the simplification routine on symbolic expressions is dependent upon the application at hand. Using an abstract interpretation framework, we give a complete description of our symbolic evaluator for a generic intermediate imperative language, state its correctness theorem with respect to the standard semantics and study the computational complexity of our method. By looking at specific examples, we show that our technique is thus not only more general and formally sounder than the ad-hoc algorithms presented in the literature but also gives better results in some cases (e.g., side effect analysis).

Partial symbolic evaluation and the two applications discussed in the paper have been implemented inside the PMACS parallel programming environment at Bull. We give performance figures comparing our solution to previous techniques, on programs given in the literature.

Keywords: symbolic evaluation, data flow analysis, abstract interpretation, interprocedural analysis, constant propagation, side-effect analysis.

1 Introduction

Many optimizing compilers perform intraprocedural data-flow analysis to gather information that is used within the optimization phase. The most common kinds of analysis are: binding-time analysis used in constant folding, available expressions for common subexpression elimination and use-kill bit vectors for dead code elimination [ASU86]. Vectorizing and parallelizing tools also rely upon the results of optimization analysis for building more accurate dependence relations among statements of programs. Independently, intraprocedural analysis have integrated more sophisticated semantics-based techniques [CH79, RT81, J87, JD89] that greatly enhance the quality of information provided to subsequent optimization phases.

While all these techniques take into account the data-flow behavior inside subroutines, only few have been adapted to go across procedure boundaries, thus impairing their precision when programs are written (as they should be) in a modular fashion. This lack of accuracy is predominantly damaging to parallelizing compilers, for which the quality of the dependence analysis is crucial for an optimal detection of implicit parallelism within programs. Although [RG89a, RG89b] argue about the usefulness of these interprocedural extensions for sequential compilers, several measurements [C88,LSY89] have shown that interprocedural data-flow analysis drastically reduces the number of dependencies.

This paper describes a new framework, based on partial symbolic evaluation, for *semantical interprocedural analysis* of programs that not only unifies different analysis schemes but also outperforms current techniques [C88]. As a practical application, we show how constant propagation and kill-use computation can be merged together. On a more theoretical note, by viewing our analysis as a non-standard interpretation of programs, we can formally prove its soundness [D90], which is seldom done in the literature.

Our approach is to apply a powerful, albeit efficient, symbolic evaluator to each procedure; its result is a function of the formals that is kept associated to the corresponding procedure. To propagate the symbolic information interprocedurally, we link this function to the intraprocedural data-flow information of the actuals at each call site. Then, depending on the data-flow analysis framework we want to deal with, a dedicated interpretation of the resulting symbolic store is performed, yielding the desired information. By limiting the amount of simplification performed on symbolic stores (i.e., by using a *partial* symbolic evaluator), different analysis such as constant propagation or side-effect can be done with the same information.

Section 2 presents the related work in symbolic interpretation and interprocedural analysis. In section 3, we define a simple language whose symbolic evaluator is discussed in section 4. Section 5 discusses the issue of partial symbolic evaluation for the interpretation of stores in the context of two interprocedural applications: constant propagation and determination of side effects of procedures. We conclude in section 6.

2 Related Work

Symbolic evaluation can be seen as an abstract interpretation [CC77] of programs. This technique is usually very time consuming and many attempts have been made to try to find a trade-off between the efficiency and the precision of the evaluation. [RL76] suggests to compute, for each variable, a text expression that represents the value of the variable at a given program point for all executions of the program. Another version of this algorithm, less precise but more efficient, has been described by [RT81]. These two techniques only deal with those variables that denote the same symbolic value for all paths of the flow graph of the program. [F84] introduces the notion of symbolic stores and investigates the use of symbolic evaluation for detecting recurrences in numerical analysis applications. [J86] presents a technique for the symbolic evaluation of structured programs with conditional branches by introducing the notion of *guarded expressions* that represent the symbolic value for a particular path in the flow graph; this technique subsumes the *conditional constants* of [WZ85]. A modified version of this technique has been used in [JD89] for improving the detection of generalized reductions, i.e. loop invariants, induction variables and reduction operations like dot products. All these techniques are intraprocedural; in this paper, we extend symbolic evaluation to general procedure calls. A similar approach has been used in the context of software re-use [CPGM91] but it addresses Ada and also resorts to a simplistic loop-unrolling scheme to deal with DO loops.

Interprocedural data-flow analysis has been mainly studied in the framework of parallelization of scientific programs. [CCKT86] introduces a series of techniques for extending the intraprocedural constant propagation methods in order to deal with procedure calls. [R79] presents an algorithm for detecting whether a procedure invocation modifies or uses a variable or preserves its value. This information is encoded either by a boolean or a symbolic expression in terms of the program variables. This algorithm is very powerful but very expensive. [CK88] introduces the *bindings graph* as a means to performing an efficient interprocedural flow-insensitive side-effect analysis. Using this graph, the formal propagation subproblem becomes linear in the length of the program. However, this method can only detect those side-effects that happen at least for one path through the flow graph of the program. [C88] proposes an interprocedural flow-sensitive side-effect determination algorithm. The flow-sensitivity comes from the intraprocedural analysis performed by the algorithm, namely *reaching definitions.* The algorithm is polynomial in the size of the program. Since no semantical information are computed, this technique cannot determine if a flow-sensitive effect occurs for all paths of the flow graph. Therefore, the result of the analysis is related to only one path. Our technique will overcome this drawback by performing a semantical analysis via symbolic evaluation. Another benefit of our approach is that only one single pass is required to perform many different analysis. It is not clear how the previously published techniques could be properly combined (e.g., constant propagation and side-effect analysis) to reach some optimum solution.

[TIF86] and [BC86] address the issue of interprocedural side-effect analysis of arrays. Our approach only deals with scalar variables but offers a much cleaner, more powerful and sounder solution than previous scalar-oriented techniques.

There is some argument about the practical impact of interprocedural analysis. [RG89a, RG89bb] consider that, for most of the optimization phases used in sequential compilers (e.g., common subexpression elimination), procedure inlining is more effective than current interprocedural algorithms (but see [HS89] for another point of view, in the domain of software testing). This is not true, as advocated in [C88], for parallelizing compilers that can use interprocedural analysis to drastically reduce the size of dependence graphs.

3 Language Definition

In order to give a formal specification of our method, we use a kernel language that includes several features of widely-used imperative programming languages such as Fortran77; it can be viewed as a generic intermediate language. We assume that branch instructions have been eliminated by some control flow structuring techniques [B77, AKPW83]. Since Fortran does not support recursive calls, they are prohibited here, although they could be taken into account by an extension of our method. Explicit declarations of global variables are not allowed. However, these variables can be seen as extra arguments of all procedures and, therefore, can be treated by our symbolic scheme. Furthermore, subroutines cannot be passed as arguments to procedure calls.

The syntax below defines the syntactic domains of Programs, Declarations, and Commands (I denote the domain of Identifiers):

```
P = D;C
D = Var I
    Proc I(I₁,..,Iₙ) D C
    D₁;D₂
C = Call I(I₁,..,Iₙ)
    If I C₁ C₂
    C₁;C₂
    Do I₁ I₂ C
```

We assume that all arguments are passed by reference. The succinctness of our syntax deserves some explanations. First, a program that uses functions can be trivially rewritten into one that only uses procedure calls and some temporaries. Consequently, a domain for expressions is not necessary any more; all operators are converted to functions, constants being nullary functions. As a final simplification and without loss of generality, we disallow nested procedure declarations; they can be eliminated by lambda-lifting [Joh86].

We give below an example of the kind of transformations that are required to map a usual Fortran-like program to our abstract syntax:

Example 1

```
SUBROUTINE TEST (X,Y)
INTEGER X, Y
X = Y+2
RETURN
END
```

is written as

```
PROC TEST(X,Y)
```

```
Var TWO;Var T;
Call Two(TWO);
Call Add(T,Y,TWO);
Call Assign(X,T);
```

where `Two` is the procedure that assigns 2 to its argument, `Add` stores in `T` the sum of `Y` and `TWO` and `Assign` stores in `X` the value of `T`.

The dynamic semantics is given denotationally. We give below the definitions of domains for Basic values, Locations, Procedure values, Values, States and Environments:

$$\begin{array}{rcl} \mathit{Basic} & = & \mathit{Bool} + \mathit{Int} + \ldots \\ l, \alpha \in \mathit{Loc} & = & \mathit{Int} \\ p \in \mathit{Proc} & = & \mathit{Value}^* \rightarrow \mathit{State} \rightarrow \mathit{State} \\ \mathit{Arg} & = & \mathit{Loc} \\ \mathit{Local} & = & \mathit{Loc} \\ \mathit{Value} & = & \mathit{Arg} + \mathit{Local} + \mathit{Proc} \\ s \in \mathit{State} & = & \mathit{Loc} \rightarrow \mathit{Basic} \\ r \in \mathit{Env} & = & \mathtt{I} \rightarrow \mathit{Value} \end{array}$$

Note that the *Value* domain contains two *Loc* summands in order to give a more precise definition of bindings by distinguishing arguments (in the first summand) from locals. Denotable values are either locations for formals and locals or procedures.

The semantical denotation of a declaration, defined by $\mathcal{D}$, is an environment transformer of type $\mathtt{D} \rightarrow \mathit{Env} \rightarrow \mathit{Env}$. Its structural definition is given below. We use the following standard notations: x, y denotes the pair of x and y, $[x_1; \ldots; x_n]$ the list constructor, $[x.y]$ the list with x as head and y as tail, @ the concatenation of lists. $[y/x]f$ is the function that maps z to y if z is equal to x and to $f\ z$ otherwise. $e \rightarrow f, g$ denotes the function that calls f if e is true and calls g otherwise. We omit injection functions in_{dom} in the domain *dom* when they can easily be inferred by the reader (likewise for projection functions out_{dom}):

$$\mathcal{D}[\![\ \mathtt{Var\ I}\ ]\!]\ r = [\mathit{NewLocal}/\mathtt{I}]r$$
$$\mathcal{D}[\![\ \mathtt{Proc\ I(I_1,\ldots,I_n)\ D\ C}\ ]\!]\ r = \lambda\mathtt{I}'.\mathtt{I} = \mathtt{I}' \rightarrow p, r\mathtt{I}'$$
$$\text{with } p = \lambda[\alpha_1; \ldots; \alpha_n].\mathcal{C}[\![\ \mathtt{C}\ ]\!]\ (\mathcal{D}[\![\ \mathtt{D}\ ]\!]\ r')$$
$$\text{and } r' = \lambda\mathtt{J}.\mathtt{J} = \mathtt{I}_i \rightarrow out_{Arg}(\alpha_i), \bot_{Loc}$$

where *NewLocal* creates a new location in the *Local* domain. A procedure declaration binds its identifier to a function, abstracted over the list of formal parameters locations, that evaluates the procedure body, using the command evaluation function $\mathcal{C}$, in an extended environment that binds each formal to its location.

The semantical denotation of a command (recall that we do not have any expression), defined by $\mathcal{C}$, is a state transformer of type $\mathtt{C} \rightarrow \mathit{Env} \rightarrow \mathit{State} \rightarrow \mathit{State}$. The structural definition of a call statement is given below:

$$\mathcal{C}[\![\ \mathtt{Call\ I(I_1,\ldots,I_n)}\ ]\!]\ rs =$$
$$\lambda l.(l \in \{r\mathtt{I}_i\}) \rightarrow (r\mathtt{I})[(r\mathtt{I}_1); \ldots; (r\mathtt{I}_n)]sl, (sl)$$

The resulting state binds the location of each actual to its value, computed in the state transformed by the called procedure; other locations' values are left unchanged.

4 Partial Symbolic Evaluation

Symbolic evaluation is an approximation of the dynamic semantics of programs. We introduce here the notion of *partial symbolic evaluation.* In a partial symbolic evaluator, some algebraic simplifications on terms are *not* performed during symbolic evaluation, but are delayed to enable more sophisticated analysis (see section 5).

4.1 Definitions

The syntax of symbolic terms is given below:

$$\begin{array}{rcl} \mathtt{E} & = & \mathtt{I} \\ & & \mathtt{I(E_1, \ldots, E_n)} \end{array}$$

The symbolic evaluator manipulates expressions in `E`, even though the program does not contain any. Builtin functions, such as the addition, which are seen as general procedures in the program, are encoded here as strict function calls; since their semantics is known, simplification will generally be possible on them.

$$\begin{array}{rcl} e \in \mathit{Sexp} & = & \mathtt{E} + \{\top, \bot\} \\ \kappa \in \mathit{Sbasic} & = & (\mathit{Sexp} \times \mathit{Sexp})^* \end{array}$$

The domain *Sexp* of symbolic expressions contains $\top$ that denotes the unknown symbolic value and $\bot$ that represents an error such as referencing an uninitialized variable. A symbolic value in *Sbasic* is a list of pairs of symbolic expressions (see section 4.4 for a simple example). The first element of each pair is called a *guard*; it denotes the symbolic logical expression of a path in the control flow graph, from the beginning of the program to the point where the symbolic evaluation is performed. The second expression denotes the value of the symbolic expression when the corresponding guard is true. This representation permits symbolic expressions to be propagated through the different branches of conditional statements.

$$\begin{array}{rcl} p \in \mathit{Sproc} & = & \mathit{Svalue}^* \rightarrow \mathit{Sstate} \rightarrow \mathit{Sstate} \\ \mathit{Svalue} & = & \mathit{Arg} + \mathit{Local} + \mathit{Sproc} \\ \sigma \in \mathit{Sstate} & = & \mathit{Loc} \rightarrow \mathit{Sbasic} \\ \rho \in \mathit{Senv} & = & \mathtt{I} \rightarrow \mathit{Svalue} \end{array}$$

Variable bindings (in *Svalue*) are looked up in symbolic environments (in *Senv*). To be able to deal with interprocedurality, the symbolic evaluator manages two separate symbolic states in *Sstate* : the *intraprocedural* and the *interprocedural* state. As a general rule, values are extracted from the intraprocedural state, except when the variable is a formal argument with a yet intraprocedurally unknown symbolic value (i.e., $[(\mathtt{true}, \top)]$ where `true` is the builtin truth constant function), in which case its value is looked for in the interprocedural state.

It will be convenient to define the following function $\mathcal{I}_s()$ that extracts the value of a location in either the intraprocedural state σ_a or the interprocedural state σ_e:

$$\mathcal{I}_s(l)\sigma_e\sigma_a = (\sigma_a\ l = [(\mathtt{true}, \top)]) \rightarrow \sigma_e\ l, \sigma_a\ l$$

Declarations

We now give the symbolic evaluation of declarations $\mathcal{D}_s$ of type $\mathtt{D} \rightarrow \mathit{Senv} \rightarrow \mathit{Senv}$ that transforms symbolic environments. The interesting part concerns procedures:

$\mathcal{D}_s[\![$ Proc I(I$_1$,..,I$_n$) D C $]\!]\, \rho =$
$\lambda I'.I = I' \rightarrow p, \rho I'$
with $p = \lambda[\alpha_1; ..; \alpha_n].\lambda\sigma_e.$
$\mathcal{C}_s[\![\, C \,]\!]\, (\mathcal{D}_s[\![\, D \,]\!]\, \rho')\kappa\sigma_e\sigma_a$
and $\rho' = \lambda J.J = I_i \rightarrow out_{Arg}(\alpha_i), \bot_{Loc}$
and $\kappa = [(\mathbf{true}, \mathbf{true})]$
and $\sigma_a = \lambda l.[(\mathbf{true}, (l \in \{out_{Arg}(\alpha_i)\}) \rightarrow \top, \bot)]$

We will see below that the symbolic evaluation $\mathcal{C}_s$ of commands requires, besides the environment ρ, a *context* κ, an interprocedural symbolic state σ_e and an intraprocedural state σ_a. A context is a symbolic value (i.e. member of *Sbasic*) that abstracts the current path in the local control flow graph of the program; it is the current guard. The top-level context is [(**true**,**true**)] since no conditionals have been locally encountered yet. Similarly, the intraprocedural state binds variables to symbolic values that have a **true** guard and $\top$ for formals and $\bot$ for locals. Therefore, formals have unknown but valid intraprocedural values for any path of the flow graph up to the moment they are assigned. Unlike formals, if a local variable is used in any path of the flow graph before being assigned, an error occurs.

Commands

The symbolic evaluation of commands $\mathcal{C}_s$ of type C $\rightarrow$ *Senv* $\rightarrow$ *Sbasic* $\rightarrow$ *Sstate* $\rightarrow$ *Sstate* $\rightarrow$ *Sstate* transforms symbolic states in a given environment and context.

The case for a procedure call is reminiscent of the one used in Section 3.

$\mathcal{C}_s[\![$ Call I(I$_1$,..,I$_n$) $]\!]\, \rho\kappa\sigma_e\sigma_a =$
$\lambda l.(l \in \{\rho I_i\}) \rightarrow \mathcal{I}_s(l)\sigma_e\sigma', \sigma_a l$
with $\sigma' = (\rho I)[\rho I_1; ...; \rho I_n]\sigma_a$

In the new state, all references to locations that are not in the argument list go to σ_a, while the ones that are passed as actuals go through the transformer associated to the body of the callee. Symbolic expressions are built whenever a builtin procedure is called. A program is symbolically evaluated in an initial environment ρ_i and state σ_i that bind them to appropriate values (see below for an example).

The problem with the test command comes from the variables that are only assigned in one of the two exclusive conditional branches [JD89]; we have to represent both the modifications incurred in one branch *and* the non-modifications in the other one. To preserve the invariant that all the guards in a symbolic value are mutually exclusive, we have to compute the symbolic states σ_i at the entry of each branch in order to take into account the corresponding value of the test I. We then separately evaluate each branch with respect to these initial states, before merging them together.

$\mathcal{C}_s[\![$ If I C$_1$ C$_2$ $]\!]\, \rho\kappa\sigma_e\sigma_a = \lambda l.\sigma_1{}'l @ \sigma_2{}'l$
with $\sigma_i{}' = \mathcal{C}_s[\![\, C_i \,]\!]\, \rho\kappa_i\sigma_e\sigma_i$
and $\sigma_i = \lambda l.\mathit{shuffle}\ \kappa_i\ (\mathcal{I}_s(l)\sigma_e\sigma_a)\ (\lambda e_1 e_2.e_2)$
and $\kappa_1 = \mathit{shuffle}\ v\ \kappa\ (\lambda xy.\mathrm{and}(x, y))$
and $\kappa_2 =$
$\mathit{shuffle}\ [(e_1, \mathrm{not}(e'_1)); ...; (e_n, \mathrm{not}(e'_n))]$
κ
$(\lambda xy.\mathrm{and}(x, y))$
and $v = [(e_1, e'_1); ...; (e_n, e'_n)] = \mathcal{I}_s(\rho I)\sigma_e\sigma_a$

where *shuffle* merges two symbolic values by combining them component-wise, performing an and of their guards and applying its third argument to the associated expressions. More formally:

$\mathit{shuffle}[x_1, y_1; ...; x_n, y_n][x'_1, y'_1; ...; x'_m, y'_m]f =$
$@_{i,j}[(\mathrm{and}(x_i, x'_j), f\ y_i\ y'_j)]$

Loops

We did not mention how our interprocedural technique applies to Do loops. The computation of the fixed point that corresponds to a loop command is independent of the issue of interprocedurality that concerns us here. It is generally impossible to compute the fixed point corresponding to a loop. Nonetheless, it can sometimes be obtained by using the technique of [JD89]. The basic idea is to match the state resulting from the symbolic evaluation of a loop body with a database of idioms that represent standard behaviors such as loop invariants or induction variables. If a match is found, the variable is bound to the corresponding generalized reduction, else a freshly defined symbolic value is assigned to the modified variable. In this latter case and conservatorily, all information about the variable are lost until another value is assigned to it (if ever). We expect this approach to be adequate for typical scientific programs; most of the loop-carried dependencies correspond to generalized reductions.

4.2 Correctness

Symbolic evaluation can be seen as an abstract interpretation [CC77] of programs written in our kernel language. This formal framework has been used in order to prove the correctness of the symbolic evaluation (see [J86] for the intraprocedural loop-free problem and [D90] for the interprocedural version of the algorithm that deals with the approximation of fixed points).

Definition 1 *For every environment r and context κ, the concretization $\gamma(r, \kappa)e$ of a symbolic expression e is defined by :*

$$\gamma(r, \kappa)e = \{v/[_i(g_i, e_i)] = \mathit{shuffle}\ \kappa\ e\ (\lambda e_1 e_2.e_2) \wedge (\exists s, \exists i/\mathcal{E}[\![\, g_i \,]\!]\, rs = \mathit{true} \wedge v = \mathcal{E}[\![\, e_i \,]\!]\, rs)\}$$

while the concretization $\Gamma(r, \kappa)(\sigma_e, \sigma_a)$ of the symbolic states (σ_e, σ_a) is defined by:

$$\Gamma(r, \kappa)(\sigma_e, \sigma_a) = \{s/\forall l, \exists v \in \gamma(r, \kappa)(\mathcal{I}_s(l)\sigma_e\sigma_a) \text{ s.t. } sl = v\}$$

The denotation of $\mathcal{E}$ is straightforward and its definition is omitted. Concretization functions map symbolic information to the set of values they approximate. The correctness theorem [D90] expresses that symbolic evaluation is a conservative approximation (via concretization functions) of the dynamic semantics.

Theorem 1 (Correctness) *For every command* C*, compatible environments r and ρ, context κ, interprocedural state σ_e and intraprocedural state σ_a*

$$\{\mathcal{C}[\![\, C \,]\!]\, rs/s \in \Gamma(r, \kappa)(\sigma_e, \sigma_a)\} \subseteq \Gamma(r, \kappa)(\sigma_e, \mathcal{C}_s[\![\, C \,]\!]\, \rho\kappa\sigma_e\sigma_a)$$

4.3 Complexity Evaluation

The symbolic evaluation is linear in the size of the program but exponential in the number of data dependencies going from one conditional to another one. This is not admissible since this parameter can grow rapidly in many existing programs. The exponential behavior comes from the use of standard abstract syntax trees (namely E) as symbolic expressions when the function *shuffle* is applied within the evaluation of conditionals.

To alleviate this problem, we developed a new representation of conditional expressions called a *value graph*, inspired by Bryant's Binary Decision Diagrams [B86]. The nodes of such graphs are either a *conditional* node, an *operation* node or an identifier. The *conditional* node represents the *shuffle* function (see section 5.3). Such a node contains three branches: the *test* branch (annotated by t) that points to the value graph associated with the test expression of the conditional, the *true* branch (annotated by 1) and the false branch (annotated by 0). The *operation* node represents the function calls in E. Consequently, since we avoid to perform the costly *shuffle* function, the complexity of the symbolic evaluation is reduced to the one of building the value graph. Obviously, this latter complexity is equal to $O(l)$ where l is the number of instructions of the program translated in our kernel language assuming that creating a node and its links costs a unit time.

All the techniques described in this paper have been implemented in the PMACS [DLTKK91] programming environment using value graphs. The first experimentations show that our symbolic evaluator runs just slightly slower than standard graph-based techniques such as [CK88].

4.4 Example

To show how our symbolic evaluator performs, we give below the result of applying $\mathcal{D}_s$ on the declaration of TEST given above (Example 1). The initial environment ρ_i defines the symbolic values of Two, Add and Assign:

$$\begin{array}{lcl} \texttt{Two} & = & \lambda[x].\lambda\sigma.\lambda l.l = x \rightarrow [(\text{true}, 2)], \sigma l \\ \texttt{Add} & = & \lambda[x,y,z].\lambda\sigma.\lambda l. \\ & & \quad l = x \rightarrow \textit{shuffle}\,(\sigma y)\,(\sigma z) +, \sigma l \\ \texttt{Assign} & = & \lambda[x,y].\lambda\sigma.\lambda l.l = x \rightarrow (\sigma y), \sigma l \end{array}$$

The result of evaluating the definition of TEST in ρ_i is an augmented environment where TEST is bound to:

$$\begin{array}{l} \lambda[x,y].\lambda\sigma.\lambda l.(l = x) \rightarrow \\ \qquad \textit{shuffle}\,(\sigma y)\,[(\text{true}, 2)] +, \\ \qquad \sigma l \end{array}$$

Note that some calls to *shuffle* have to be delayed until the actual parameters are known; in any case, they are completely performed on local expressions.

5 Applications

Our formalism can be applied to the computation of a wealth of interesting program properties, since symbolic evaluation strives to mimic the original program semantics. As an illustration, we give below two examples where we discuss some interprocedural analysis: (1) constant expression propagation and (2) determination of side-effects.

Up to now, we voluntarily omitted the issue of symbolic expressions simplification. It turns out that different uses of the symbolic state require different simplification rules for both the symbolic guards and the values. This is the crux behind the notion of *partial* symbolic evaluation.

5.1 Interprocedural Constant Propagation

The determination of constant expressions at a given call site is straightforward once the complete symbolic state has been computed. The simplification routine for this problem is the usual one found, for instance, in optimizing compilers that perform constant folding [ASU86]. We only add to these standard rules the following one, informally described as:

$$\begin{array}{l}[\ldots;(\text{and}(e,f),e');\ldots;(\text{and}(\text{not}(e),f),e');\ldots] = \\ \qquad [\ldots;(f,e');\ldots]\end{array}$$

It is mainly used in the test command to simplify unmodified expressions when merging the symbolic states of the two branch analysis.

Definition 2 (Constant) *A variable* I *is* constant *across a procedure call if its simplified symbolic value is of the form* $[(e_1,c);\ldots;(e_n,c)]$ *where* c *is some builtin constant function.*

5.2 Interprocedural Side Effect Analysis

Since symbolic evaluation gives a static representation of the dynamic behavior of programs of which side effects are an approximation, we show that there exists a simplification routine that enables the determination of interprocedural side effect information, also called *kill-use* or *mod-use* analysis.

Definition 3 (Local Expressions) *A symbolic expression is* local *in a symbolic environment* ρ *if all its terms are either a constant, a local variable* I *(such that* $is\_in_{Loc}(\rho \mathrm{I})$ *is true) or its subexpressions are local.*

For side effect analysis, any simplification technique can be used, as long as they are only performed on local expressions. For such simplification to be effective, normalization of symbolic expressions (such as reordering summands or multiplicands) can be required to put together local subexpressions. The reason why non local expressions cannot be simplified can be seen by looking at the following example:

```
Proc LOSE(X)
Var ZERO ;
Call Subtract(ZERO,X,X) ;
Call Assign(X,ZERO) ;
```

where if the expression X-X were simplified to 0, then the read access on X would be lost. Note that this restriction could be alleviated if expressions evaluation were atomic in the dynamic semantics.

Two kinds of side effect analysis (we discuss here the case of *mod* analysis, the case for *use* being similar) can be performed by a simple treatment of the symbolic stores. The first ones are independent of the call site and are obtained by applying the symbolic state transformer of the callee Q to a simple interprocedural state σ_0 that binds each formal to itself: $\sigma_0(\rho \mathrm{I}_i) = [(\texttt{true}, \mathrm{I}_{i0})]$. Consider the call site defined by Call $\mathrm{Q}(\mathrm{I}_1, \ldots, \mathrm{I}_n)$.

- Flow-insensitive [CK88] (may) *mod* analysis consists in finding at least one path where a given argument is modified. If the resulting symbolic state binds the address of the formal to [(true, ⊤)], the formal is not modified on any path. More formally,

$$I_i \in Mod\,(Q) \Leftrightarrow (\sigma'(\rho I_i)) \neq [(\mathrm{true}, \top)]$$

where $\sigma' = (\rho\ Q)[\rho I_1; ...; \rho I_n]\sigma_0$

- Flow-sensitive (must) *mod* analysis consists in finding all the paths where a given argument is modified. If the resulting symbolic state binds the address of the formal to a symbolic value in which all terms are equal to the formal, then the formal is never modified. More formally, if σ_a denotes the symbolic state after the procedure call is performed:

$$I_i \in Mod\,(Q) \Leftrightarrow \forall i \in \{1, \cdots, n\}\ e'_i \neq I_{i0}$$

where $\sigma'(\rho I_i) = [(e_1, e'_1); \cdots; (e_n, e'_n)]$ and $\sigma' = (\rho\ Q)[\rho I_1; ...; \rho I_n]\sigma_0$.

The second class of side effect analysis is dependent of a particular call site and is obtained by applying the symbolic state transformer of the callee to the current intraprocedural state of the caller. If the resulting symbolic state binds the address of the formal to a symbolic value in which all the terms are equal to the ones they had before the call site, then the formal is never modified. For more details, see [D90].

5.3 Example

The behavior of our interprocedural scheme is illustrated in the following example written in Fortran 77 that is inspired by the example given in [C88].

```
subroutine main(x,y)          subroutine suba(x,y,z)
call suba(x,y,0) (1)          call subb(x,y,z)
call suba(5,y,x) (2)          call subc(x,y,z)
z=x-y                         return
end                           end

subroutine subb(u,v,w)        subroutine subc(u,v,w)
if (u.eq.5) w=v               if (u.neq.5) w=v+1
return                        return
end                           end
```

If x_0, y_0 and z_0 are the initial values of x, y and z, the symbolic resulting store of the procedure suba is:

$$\begin{aligned} x &\leftarrow [\mathrm{true}, x_0] \\ y &\leftarrow [\mathrm{true}, y_0] \\ z &\leftarrow [(=(x_0,5), y_0); (\mathrm{not}(=(x_0,5)), y_0+1)] \end{aligned}$$

Similarly, the figure 1 shows the value graph associated to the symbolic values of the variables x and z in main.

Let us look at what can be detected in main by our two analysis. With side-effect analysis, x is seen to be modified for all the paths through the control flow graph; this a *must* modify situation and both [CK88] and [C88] fail to find this result. On the contrary, y is not modified. For constant propagation, after the call site (2), since the value of x is y_0, z can be shown to be equal to 0.

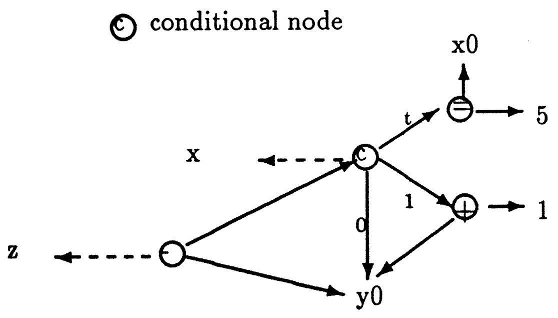


Figure 1: Example of value graph

Program	Nb. of Lines	Method	Time (s)
Example (above)	35	[CK88]	0.20
		SIA	0.20
[CCKT86]	25	[CK88]	0.16
		SIA	0.18
Long1	14	[CK88]	0.70
		SIA	1.00
Long2	410	[CK88]	5.30
		SIA	7.50

Figure 2: Performance comparison between [CK88] and our semantical interprocedural analyzer (SIA) for side-effect analysis on a Sun Sparc 1 workstation.

6 Conclusion

We described a new scheme for the interprocedural semantical analysis of programs. Based on techniques of symbolic evaluation and abstract interpretation, this general framework can be applied to different specific problems. We have shown how interprocedural constant propagation and side effect analysis can be derived by using two different simplification routines on the symbolic store computed by the partial symbolic evaluator. Our technique can be proved correct by standard proof methods and is able to perform as well as or better than the already known ad-hoc algorithms.

As presented in this paper, our method does not deal with recursive calls. They can be, in many cases, transformed into iterative loop constructs using techniques standard in functional languages rewriting systems. Then, techniques such as [JD89] can be applied to the resulting iterative constructs. Another area of future research is the extension of this approach to arrays.

This powerful interprocedural symbolic evaluator has been implemented in less than 2000 lines of C code within the PMACS parallel programming environment used at Bull [DLTKK91]. By using value graphs to avoid generating symbolic expressions of exponential size, we found that its performance on literature programs ([R79],[CCKT86],[D90]) is satisfactory compared to our implementation of [CK88] (cf. Figure 2). Long1 and Long2 are toy programs that exhibit an artificially large number of data dependencies; this explains the degradation of performance of our approach (see section 4.3). We do not expect this effect to be a factor in real-life programs.

References

[ASU86] Aho, A. V., Sethi, R., and Ullman J. D. *Compilers.* Addison Wesley, 1986.

[AKPW83] Allen, R., Kennedy, K., Porterfield, C., and Warren, J. Conversion of Control Dependencies to Data Dependencies. In *Proceedings of ACM POPL 83*, Jan. 1983

[B77] Baker, B. An algorithm for Structuring Flow Graphs. *Journal of the ACM*, vol. 24, no. 1, pp. 32, Jan. 1977

[B86] Bryant, R. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, vol. c-35, no. 8, Aug. 1986

[BC86] Burke, M., and Cytron, R. Interprocedural Dependence Analysis and Parallelization. In *Proceedings of the ACM Sigplan'86 Symposium on Compiler Construction*, Jun. 1986

[C88] Callahan, D. The Program Summary Graph and Flow-sensitive Interprocedural Data Flow Analysis. In *Proceedings of ACM Sigplan PLDI'88*, Atlanta, Jun. 1988

[CCKT86] Callahan, D., Cooper, K., Kennedy, K., and Torczon, L. Interprocedural Constant Propagation. In *Proceedings of the ACM Sigplan'86 Symposium on Compiler Construction*, pp. 152-161, Jun. 1986

[CPGM91] Coen-Porisini, A., Paoli, F., Ghezzi, C., and Mandrioli, D. Software Specialization via Symbolic Execution. *IEEE Transactions on Software Engineering*, vol. 17, np. 9, Sep. 1991

[CK88] Cooper, K., and Kennedy, K. Interprocedural Side-Effect Analysis in Linear Time. In *Proceedings of Sigplan PLDI'88*, Atlanta, Jun. 1988

[CC77] Cousot, P., and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximations of Fixed points. In *Proceedings of ACM POPL 77*, Jan. 1977

[CH79] Cousot, P., and Halbwachs, N. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of ACM POPL 79*, Jan. 1979

[D90] Dehbonei, B. *Génération de code et analyse interprocédurale au sein d'un environnement de programmation parallèle.* Ph.D. Thesis, University of Paris 6, Dec. 1990

[DLTKK91] Dehbonei, B., Laurent, C., Tawbi, N., Kulkarni, R., and Kulkarni, S. PMACS: An Environment for Parallel Programming. In *Proceedings of the Fifth IEEE International Parallel Processing Symposium*, Anaheim, Apr. 1991

[F84] Feautrier, P. Outil de calcul symbolique. In *Proceedings of the International Symposium on Programming*, Toulouse, LNCS 167, Springer-Verlag, Apr. 1984

[HS89] Harrold, M.J., and Lou Soffa, M. Interprocedural Data Flow Testing. In *Proceedings of the Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments*, 1989

[Joh86] Johnsson, Th. *Lambda-Lifting: Transforming Programs to Recursive Equations.* Chalmers Institute of Technology, 1986

[J86] Jouvelot, P. Parallelization by Symbolic Detection of Reductions. In *ESOP86*, LNCS 213, Springer Verlag, Mar. 1986

[J87] Jouvelot, P. Semantic Parallelization: A Practical Exercise in Abstract Interpretation. In *Proceedings of ACM POPL 87*, Jan. 1987

[JD89] Jouvelot, P., and Dehbonei B. A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions. In *Proceedings of ACM International Conference on Supercomputing 89*, Crete, Greece, Jun. 1989

[LSY89] Li, Z., Shen, Z., and Yew, P. An Empirical Study on Array Subscripts and Data Dependencies. In *Proceedings of the International Conference on Parallel Processing*, vol 2, pp. 145-152, 1989

[RT81] Reif, J., and Tarjan, R. Symbolic Program Analysis in Almost Linear Time. *SIAM Journal of Computing*, vol. 11, pp. 81-93, Feb. 81

[RL76] Reif, J., and Lewis, H. Symbolic Evaluation and the Global Value Graph. In *Proceedings of ACM POPL 76*, Jan. 1976

[RG89a] Richardson, S., and Ganapathi, M. Interprocedural Analysis vs. Procedure Integration. *Information Processing Letters* 32, pp. 137-142, 1989

[RG89b] Richardson, S., Ganapathi, M. Interprocedural Optimization: Experimental Results. *Software, Practice and Experience* 19, pp 149-169, 1989

[R79] Rosen, B. Data Flow Analysis for Procedural Languages. *Journal of the ACM*, vol. 26, no. 2, pp 322-344, Apr. 1979

[TIF86] Triolet, R., Irigoin, F., and Feautrier, P. Direct Parallelization of Call Statements. In *Proceedings of the ACM Sigplan'86 Symposium on Compiler Construction*, Jun. 1986

[WZ85] Wegman, M., and Zadeck, K. Constant Propagation with Conditional Branches. In *Proceedings of ACM POPL 85*, Jan. 1985

A Polyvariant Binding Time Analysis [*]

Bernhard Rytz, Marc Gengler

Ecole polytechnique fédérale de Lausanne
Swiss Federal Institute of Technology - Lausanne
Laboratoire d'Informatique Théorique
EPFL-DI-LITH, Ecublens(IN), CH-1015 Lausanne, Switzerland
`{rytz,gengler}@lithsun1.epfl.ch`

Abstract

A *binding time analysis* (BTA) is a crucial part of a self-applicable partial evaluator. Given the source program and a description of which inputs will be known, the BTA annotates every expression of the source program as either known or unknown.

For improved accuracy, a *polyvariant* BTA may create more than one annotation of an expression, leading to more efficient specialization. This paper presents a polyvariant BTA, duplicating procedures and lambda-abstractions, for a higher-order functional language.

Keywords : binding time analysis, polyvariance, higher-order, functional language, partial evaluation, self-application.

1 Introduction

Partial evaluation transforms programs with incomplete input data. The partial evaluator *specializes* a given source program together with a part of its input to obtain a *residual* program. Applying the residual program to the remaining input gives the same result as applying the source program to all of the input.

Specializing the partial evaluator itself is called *self-application* and may be used to generate compilers from interpreters and even a compiler generator [Fut71].

It has been shown that *preprocessing* is an essential stage of a self-applicable partial evaluator. Preceding the actual program specialization, preprocessing adds *annotations* to the source program. These annotations will guide the specializer generating the residual program. The crucial phase of preprocessing is the *Binding Time Analysis* (BTA) : Given the source program and a description of the inputs that will be known during specialization, the BTA statically determines which expressions of the source program solely depend on these known inputs. Known expressions will be evaluated (reduced) by the specializer, whereas unknown expressions must be reconstructed in the residual program. We will say that fully known expressions have the binding time value *static*, unknown expressions are called *dynamic*.

[*]This research was funded by the Swiss National Science Foundation, Grant No. 20-27951.89.

2 Overview

The remainder of this paper is organized as follows : In section 3 we present very briefly the partial evaluator Similix-2 [Bon90, Bon91b] on which our system is built. Section 4 shows how a monovariant BTA as used in Similix can lead to inefficient residual programs. Section 5 explains how polyvariance is achieved through expression duplication. In section 6 we show that more fine-grained binding time values allow better polyvariance. Section 7 discusses efficiency issues. Section 8 briefly presents how partially known values may be treated in the same framework. Section 9 provides examples. Section 10 concludes.

3 Similix

Since our work is an extension of the partial evaluator Similix-2 (Similix for short), we give a short description of Similix' functionality and structure. Details may be found in [Bon90]. Similix is a self-applicable partial evaluator for a higher-order subset of the weakly typed functional language Scheme [RC91]. The Similix system consists of several preprocessing phases, a specializer and a postprocessor. Similix is "automatic" in the sense it doesn't require any annotations by the programmer. The use of dynamic conditionals as "specialization points" avoids infinite call unfolding. The problem of infinite specialization however is not addressed.

3.1 The language treated by Similix

A Similix program is a collection of user-defined procedures and user-defined operators. While the partial evaluator knows the code of procedures, operators are treated as primitives: they are either completely evaluated or left untouched. The syntax of Similix programs [Bon90] :

```
Pr ∈ Program, PD ∈ Definition, F ∈ FileName,
E ∈ Expression, C ∈ Constant, V ∈ Variable,
O ∈ OperatorName, P ∈ ProcedureName
Pr ::= (loadt F)* (load F)* PD+
PD::= P (V*) = E
E  ::= C | V | (if E E E) | (let ((V E)) E) |
       (begin E+) | (O E*) | (P E*) |
       (lambda (V*) E) | (E E)
```

We are using a usual semantics for such a language, assuming call-by-value and lexical scoping.

3.2 The preprocessing phases of Similix

All the preprocessing phases work on the source code and attach annotations (attributes) to some or all expressions.

- The first phase of preprocessing is the *closure analysis* (CLA). Its purpose is to collect, for every application point, the set of lambda-abstractions that might be applied at that point. Given only the source program it annotates every expression with the set of lambda-abstractions that might be a result of the expression.

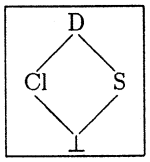
- The second phase of preprocessing is the binding time analysis (BTA). As mentioned before, its purpose is to determine the expressions that solely depend on known inputs and which may thus be evaluated by the specializer. The BTA uses the annotations provided by the CLA to determine at every application point, the binding times of the implicated lambda-abstractions' parameters and the binding time value of the result of the application.

- The other phases are not relevant to our work. They deal (among other things) with computation duplication and automatic specialization.

The CLA and the BTA are based on *abstract interpretation* of the source program: for each analysis a finite lattice of abstract values and a set of monotonic abstract rules are defined. To do the analysis, the program annotations are first initialized with the least abstract value. Then the source program is traversed, applying the abstract rules to every expression and updating the corresponding annotation. This traversal is repeated until a fixed point is reached.

Similix' CLA indexes all the lambda-abstractions of a source program to identify them unambiguously. The lattice of abstract closure values is the powerset of the set of all lambda-abstraction indices, ordered by set inclusion.

The BTA uses a lattice BT that includes the standard bottom ($\perp$), static (S), dynamic (D) values as well as a *closure* (Cl) value denoting statically known lambda-abstractions. Note that known expressions that can return ordinary and closure values are classified as dynamic. The bottom value is only used by the BTA (and not the specializer) to denote a "yet unknown" binding time value. The completely binding time analyzed program does not contain any bottom annotations unless there is some programming error.



The abstract rules for the CLA and the BTA may be found in [Bon90]. An expression e which as a binding time value X will be denoted as X:e.

4 Monovariance of the BTA and its implications

A BTA must satisfy the following (conflicting) constraints :

- To be *safe*, the BTA must never declare incorrectly that an expression is static. Otherwise the partial evaluator might try to evaluate an unknown expression.

- To be *useful*, the BTA should declare as many expressions as possible as being static. Only expressions declared static will be reduced by the partial evaluator.

Since the BTA is based on abstract interpretation over a very restricted (finite) domain, its results can only be approximate. According to the safety criterion above, however, it must only err "on the dynamic side".

Some expressions may be the result of several other expressions which don't necessarily have the same binding time values. When the BTA encounters conflicting binding time values for an expression, a *widening* ensures the unique annotation required by the specializer. Widening typically takes the "most dynamic" or, more precisely, the least upper bound (as defined by the lattice) of the conflicting binding time values to ensure that only the intersection of static values is treated as static. Widening implies that certain static expressions will be annotated as being dynamic and thus will not be reduced by the specializer. This may lead to inefficient residual programs. A BTA which systematically widens conflicting binding time values is called *monovariant*.

Widening may happen when an expression has two or more "sources". This is the case for conditional expressions (which select one of two expressions) or parameters of procedures or lambda-abstractions called at more than one point. Using simple examples, we now present the situations in which widening occurs.

4.1 Widening for if-expressions

The result of a conditional is either of its branches. These branches don't necessarily have the same binding time.

```
main (a b) = (if (zero? a) b '17)    with S:a, D:b
```

Since we don't know the actual value of the parameter a during the BTA, we don't know which of the branches of the conditional will be selected. One of the branches being unknown, the BTA must assume, that the result of the conditional will be unknown. Should a be different from 0 during specialization we will treat the static value 17 as being dynamic and never operate any reduction on it.

4.2 Widening in Residual Code Contexts

In the so-called *residual code contexts* (RCC) [Bon90], a lambda-abstraction must be treated as being completely dynamic. This is the case if the lambda-abstraction may be the result of a dynamic expression.

```
main (a b) = (let ((f (lambda (x) (* x x))))    with S:a, D:b
               (list (f a) ((if (zero? b) f (lambda (x) 1)) a)))
```

In this example, the lambda-abstraction f may be returned by the dynamic if-expression. Therefore f will figure textually in the residual program and all occurences of f must be dynamic. Specializing for a=3 yields the program main-mono instead of the more efficient main-poly :

```
main-mono (b) = (let ((f (lambda (x) (* x x))))
                  (list (f 3) ((if (zero? b) f (lambda (x) 1)) 3)))
main-poly (b)  = (let ((f (lambda (x) (* x x))))
                  (list 9 ((if (zero? b) f (lambda (x) 1)) 3)))
```

4.3 Widening for the parameters of a procedure

If the same procedure is used with different binding times for the arguments, we must widen the different tuples of abstract arguments to obtain a unique safe description of the procedure parameters for all calls to the procedure.

```
main (a b) = (list (test a b) (test b a))   with S:a, D:b
test (x y)  = (+ (* x x) (* y y))
```

The procedure `test` is called with the abstract tuples (S×D) and (D×S). Widening these tuples gives (D×D), disallowing any reduction of `test` during partial evaluation. Specializing for a=3 we get **`main-mono`** instead of **`main-poly`**:

```
main-mono (b) = (list (+ (* 3 3) (* b b)) (+ (* b b) (* 3 3)))
main-poly (b)  = (list (+ 9 (* b b)) (+ (* b b) 9))
```

4.4 Widening for the parameters of a lambda-abstraction

Like procedures, lambda-abstractions may be applied to different tuples of abstract arguments.

```
main (a b) = (let ((f (lambda (x y) (+ (* x x) (* y y)))))
                  (list (f a b) (f b a)))   with S:a, D:b
```

The parameters of the lambda-abstraction `f` are widened to (D×D) for the same reasons as described in 4.3.

4.5 Widening due to functional parameters of a procedure

Sometimes a procedure call or lambda-abstraction application may cause widening even when there is only one tuple of abstract arguments.

```
main (a b) = (let ((f (lambda (x) (+ a x)))    with S:a, D:b
                   (g (lambda (x) (+ b x))))
                  (list (test f a) (test g a)))
test (h i)   = (let ((r (h i))) (* r r))
```

Even though the binding times of the arguments are the same (Cl×S) in both calls to `test`, there occurs a widening as the residual program (for a=3) shows:

```
main-mono (b) = (list (* 6 6) (* (+ b 3) (+ b 3)))
```

In Similix, the BTA uses only one binding time value (Cl) to denote lambda-abstractions. Different lambda-abstractions may however, when applied to the same abstract arguments, yield different abstract results. This may lead to widening at the point of application of such lambda-abstractions. In our example the lambda-abstraction `f` has an abstract signature (S→S) while g has the signature (S→D). Since the parameter h in `test` may evaluate to `f` or g, the result of the application `(h i)` becomes dynamic.

4.6 Widening due to functional parameters of a lambda-abstraction

By replacing in 4.5 the procedure `test` by an equivalent lambda-abstraction, we see that lambda-abstractions can cause the same kind of widening as procedures, when functional arguments of different signatures are passed to them.

5 A Polyvariant Binding Time Analysis

In section 4 we have shown when widening occurs and that it may lead to inefficient residual programs. In this section and the next we show how widening can be avoided. We must satisfy the two following goals :

- To obtain self-applicability of the specializer, the BTA must *completely* and *unambiguously* annotate every expression of the source program.
- To avoid widening, the BTA should provide *flexible* annotations of certain expressions.

We achieve this through *duplicating* some expressions of the source program. Duplicated expressions may be annotated independently and thus widening can be avoided in many cases. Duplication happens before program specialization and its effects are mostly invisible to the specializer. The specializer thus can be kept simple, which is important for efficient self-application.

The duplication of expression happens during the BTA. The BTA repeatedly traverses the abstract syntax tree of the source program. At each expression the binding time annotation is updated according to the abstract rule for the current expression. If a widening of the expression's binding time value is about to happen we will try to avoid it by duplicating the expression concerned. Note that, strictly speaking, we don't need to duplicate the offending expression itself but just its annotations. For simplicity we will continue talking about "expression duplication".

We will now introduce step by step a polyvariant analysis for the different widening situations described in 4.1-4.6. For each situation the following questions must be examined. How can the BTA recognize an imminent widening? What kind of duplication must be done to avoid widening? What impact does this expression duplication have on the BTA or even the whole preprocessing stage. Will the BTA still remain finite?

5.1 Polyvariance for if-expressions

The solution to this problem lies in the equivalence of

```
(E1 (if E2 E3 E4)) = (let ((V E1)) (if E2 (V E3) (V E4)))
```

This problem obviously cannot be resolved by the BTA through mere duplication of some annotations and we don't address it any further.

5.2 Polyvariance for Residual Code Contexts

Some lambda-abstractions in residual code contexts may be protected against widening through *eta-conversion*[Mos91]. Details of our method based on automatic eta-conversion of certain variables bound to lambda-abstractions are described in [Ryt92].

5.3 Polyvariance for the parameters of a procedure

The obvious solution (which has been widely used manually as well as for first-order languages) is to duplicate the procedure for each tuple of abstract arguments. We call all duplicates of a given procedure its *variants*. All variants of one procedure are operationally equivalent but may have different binding time annotations.

5.3.1 Detecting widening & creating duplicates

The part of the BTA dealing with procedure calls may be described by the following function `bt`. `bt` is common to a monovariant and a polyvariant analysis but the function `choose-procedure` it calls differs in both cases.

```
bt : Expression → BT
bt[Expr] =
  case [Expr] of
     ...
     [(P E_1 ... E_n)]:
         let param* == <param_1 ... param_n>
                    == fetch-procedure-parameters(P),
             bt-par* == <bt[param_1] ... bt[param_n]>,
             bt-arg* == <bt[E_1] ... bt[E_n]>,
             P' == choose-procedure(P, param*, bt-par*, bt-arg*)
         in  replace-caller(Expr, P')
             annotate(Expr,fetch-procedure-bt(P'))

annotate : Expression × BT → BT
annotate(Expr, Bt) =
  set-bt(Expr, Bt)
  return(Bt)
```

In a monovariant BTA `choose-procedure` is defined as :

```
choose-procedure : ProcedureName × Parameter* × BT* × BT*
                   → ProcedureName
choose-procedure(P, param*, bt-par*, bt-arg*) =
  for i:[1..n] do annotate(param_i, bt-par_i ⊔ bt-arg_i)
  return(P)
```

where ⊔ is the smallest upper bound to its two arguments according to the lattice BT. A polyvariant BTA might define `choose-procedure` as :

```
choose-procedure : ProcedureName × Parameter* × BT* × BT*
                   → ProcedureName
choose-procedure(P, param*, bt-par*, bt-arg*) =
  if bt-par* = bt-arg* then
    return(P)
  else if variant-exists?(P, bt-arg*) then
       return(fetch-variant(P, bt-arg*))
     else
       return(duplicate-procedure(P, bt-arg*))
```

where `duplicate-procedure` creates a new variant of a procedure for a given tuple of abstract arguments. This new variant, which initially has "bottom" annotations, is added to the source program and remembered, such that it can be reused. Different variants are naturally indexed by the binding times of their parameters and receive a unique procedure name. Since the BTA works by continually updating binding time annotations, some variants that have been created, may be of no use in the completely binding time analyzed program. A call graph analysis may be done when the annotation process is finished in order to eliminate them.

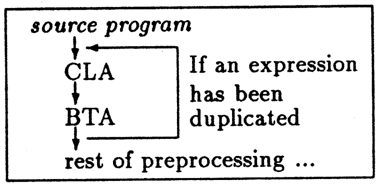
5.3.2 Iterating the CLA & BTA

Duplication of procedures seems to be straightforward at first glance. There is a catch however : Remember that the closure analysis, run before the BTA, annotates each expression with the set of lambda-abstractions the expression might evaluate to. Now, when we duplicate procedures, what happens with their CLA-annotations? We might simply preserve those annotations and thus share lambda-abstractions among variants of the same procedure. Unfortunately this leads to yet another instance of undesired widening. Consider the following example :

```
main (a b) = (list (test a) (test b))   with S:a, D:b
test (x)   = (let ((f (lambda () x)))
                (* (f) (f)))
```

Two variants of the procedure `test` will be generated, one for S:x (say `test.S`), another for D:x (`test.D`). If the lambda-abstraction bound to `f` is shared by the two variants, its free variable `x` has two different binding time values and must be widened to dynamic. Thus the variant `test.S` will have the same, dynamic, annotation as the variant `test.D` which won't result in an efficient residual program.

To prevent this kind of widening of their free variables, lambda-abstractions should not be shared between variants. Each time a new variant is created, the lambda-abstractions occurring in that variant, must be distinguishable from existing lambda-abstractions. Introducing a new variant of a procedure changes the flow of control and may globally invalidate the CLA-annotations. Therefore we choose to iterate CLA & BTA when a new variant is created.


source program
CLA
BTA
If an expression has been duplicated
rest of preprocessing ...

Of course it would be very costly to restart the CLA & BTA each time a new variant is found. In section 7 we explain, how several variants may safely be created before an iteration is needed.

Iterating CLA & BTA may also avoid generating useless variants for lambda-abstractions or procedures.

```
main (a b)   = (let ((f (lambda (x y) (* x y)))   with S:a, D:b
                     (g (lambda (x y) (+ x y))))
                 (list (test f a b) (test g b a)))
test (u v w) = (u v w)
```

The CLA detects that the variable `u` at the application point `(u v w)` in `test` may be bound to the lambda-abstractions `f` and `g`. The BTA generates two variants for `test` with the binding time values (Cl×S×D) and (Cl×D×S). Hence for both `f` and `g` we must generate the two variants (S×D) and (D×S).

If we restart the CLA after BTA has generated the variants for `test` we get more precise results. The second CLA will be applied to the following code.

```
main (a b)   = (let ((f (lambda (x y) (* x y)))   with S:a, D:b
                     (g (lambda (x y) (+ x y))))
                 (list (test.Cl.S.D f a b) (test.Cl.D.S g b a)))
test.Cl.S.D (u v w) = (u v w)
test.Cl.D.S (u v w) = (u v w)
```

Now, the variable `u` of `(u v w)` in `test.Cl.S.D` (resp. `test.Cl.D.S`) may only point to `f` (resp. `g`). Hence, we need only the variants (S×D):`f` and (D×S):`g`.

5.4 Polyvariance for the parameters of a lambda-abstraction

Like procedures, lambda-abstractions may be called with different abstract arguments. It seems obvious to duplicate a lambda-abstraction for each tuple of abstract arguments. We call these duplicates *versions*. Since lambda-abstractions are anonymous and we don't want to overly change the source programs by explicitly passing around all versions, we simply wrap them in a new syntactic construction that replaces the original lambda-abstraction (see example). Operationally and from a CLA point of view this construction behaves like any of the versions it contains. At the point of application of the lambda-abstraction the BTA chooses the appropriate version, based on the arguments' binding time values. We also extend the specializer in this way to handle the versions. The choice of the appropriate version is thus entirely determined by the binding time annotation, which is crucial for efficient self-application.

5.4.1 Detecting widening & creating duplicates

Thanks to the CLA annotations we know at each point of application the set of applicable lambda-abstractions. For each of those lambda-abstractions a version corresponding to the binding time values of the arguments must exist.

The next example shows a versions-construct and the way it is extended to contain multiple versions for the different binding time values possible. We give the (approximate) annotations only for the abstractions involved. In the following we will not explicitly write the `versions` operator.

```
main (a b)     = (let ((f (versions (lambda (x) x))))   with S:a, D:b
                   (list (f a) (f b)))
main_ann (a b) = (let ((f (versions Cl:(lambda (S:x) S:x)
                                    Cl:(lambda (D:x) D:x))))
                   (list (f a) (f b)))
```

5.5 Finiteness of the CLA & BTA

For both procedures and lambda-abstractions, only a finite number of variants or versions will be generated. At most, we create as many variants of a procedure as there are tuples of abstract arguments. If r is the number of possible binding time values (r=4 in Similix) and d the arity of the procedure, then there are at most r^d possible variants per procedure. The same applies to lambda-abstractions and their versions.

The CLA is exactly the same as in [Bon90] and has been proved to be finite. The BTA also is the same as [Bon90], unless it detects a widening situation, in which case it stops (after duplicating an expression). Thus it is obviously finite. The number of iterations of CLA & BTA is limited by the number of new variants/versions that may be created and hence is finite.

6 Extending the binding time lattice

6.1 Motivation

So far we based the detection of widening on the simple binding time lattice BT of Similix which uses one value (Cl) to denote all statically known lambda-abstractions. Certain cases of widening, as described in 4.5 and 4.6, involving procedures and lambda-abstractions, will escape our polyvariant BTA, unless we use a more fine-grained binding time lattice taking into account binding time signatures of lambda-abstractions. Let's first treat a typical problem in an informal way and consider again the example of 4.5.

```
main (a b) = (let ((f (lambda (x) (+ a x)))   with S:a, D:b
                   (g (lambda (x) (+ b x))))
               (list (test f a) (test g a)))
test (h i)  = (let ((r (h i))) (* r r))
```

As pointed out previously, `h` may be bound to the abstractions `f` and `g` which have the different signatures (S→S) and (S→D). The type of `(h i)` is thus S for the call `(test f a)` and D for `(test g a)`. Note that Similix must inspect the annotation of the abstraction bodies of `f` and `g` in order to get the types of the results for `(h i)`. If, as it is the case here, the result types are different then a widening takes place.

Our approach to this problem consists in "indexing" the binding time value Cl of the formal parameter `h` with the possible signatures of the abstractions `h` is bounded to. As both `f` and `g` may be bound to the formal parameter `h`, the two abstractions undergo the same treatment in the procedure `test`. In particular they will be applied at the same points. Hence we may avoid duplicating the procedure `test` as long as the versions common to `f` and `g`, i.e. applied to the same arguments, are compatible, that is, yield the same type of result. In the opposite case, we duplicate the procedure.

Let's point out that the correctness is trivial. We might not duplicate the procedures, systematically duplicate all procedures or do anything in between. The correctness is in fact guaranteed by the widenings taking place where necessary. The finiteness is less trivial and will be addressed later.

In our example, the computation proceeds as follows : In a first traversal, there are no versions for `f` and `g` and `test` will not be duplicated. Analyzing `test` will generate a (S→S) version for `f` and a (S→D) version for `g`.

In the second pass, the BTA will notice that `f` and `g` are incompatible, i.e. may yield results of different binding time values when applied to a static argument. The procedure `test` is therefore duplicated. The overall result will be as expected :

```
main (a b) = (let ((f (lambda (x) (+ a x))) with S:a, D:b
                   (g (lambda (x) (+ b x))))
               (list (test-f f a) (test-g g a)))
test-f (h i) = (let ((r (h i))) (* r r))
test-g (h i) = (let ((r (h i))) (* r r))
```

6.2 Considering more fine-grained binding time values

When treating a procedure call, the BTA must determine whether an appropriate variant already exists or not. Thus, we need a compatibility predicate on the call-arguments to decide whether two different calls may share the same variant without widening.

Consider two unary calls `(P E₁)`, `(P E₂)` (for n-ary procedures, arguments are handled pair-wise). So far (5.3) we compared only the binding times of the arguments :

$$\text{compatible?}(E_1, E_2) = (\text{bt}[E_1] = \text{bt}[E_2])$$

To avoid widening for arguments that have `bt[Eᵢ]=Cl` we must distinguish closures of different binding time signatures. As mentioned in 3.2 the CLA annotates each expression with the set of closures it may evaluate to. Consequently we now use the closure annotations as well as the binding time annotations when comparing call arguments. The following formulation of `compatible?'`, which simply compares the closure sets for equality, is not satisfying because many occasions for sharing are neglected. Indeed, if different abstractions have the same binding time signatures they may be passed to the same variant without causing widening.

$$\text{compatible?'}(E_1, E_2) = (\text{bt}[E_1] = \text{bt}[E_2]) \wedge (\text{cl-set}[E_1] = \text{cl-set}[E_2])$$

Let's repeat that a lambda-abstraction may exist in several versions. Two different lambda-abstractions passed to a procedure may share that procedure without widening, if all the versions they have in common have the same binding time signature. The following formulation of `compatible?''` expresses this idea.

```
compatible?"(E_1, E_2) =
  (bt[E_1] = bt[E_2]) ∧
  (bt[E_1] = Cl ⇒
        let $X$ == all-versions(cl[E_1]), $Y$ == all-versions(cl[E_2])
        in $\forall d. \forall x == (x_1 \times \ldots \times x_d \rightarrow xr) \in X.$
              $\forall y == (y_1 \times \ldots \times y_d \rightarrow yr) \in Y.$
                $(x_1 \times \ldots \times x_d) = (y_1 \times \ldots \times y_d) \Rightarrow xr = yr$
```

Now we are left to define the function `all-versions` which, given a set of closures, is supposed to return a set of binding time signatures, describing the complete (binding time) behavior of the abstractions in the closure set.

As described in 4.6 widening may not only occur when different lambda-abstractions are passed to procedures but also when different lambda-abstractions are passed to lambda-abstractions. Therefore abstract signatures may actually be nested. Example :

```
main (a b) = (let ((f (lambda (h x y) (h (+ x y))))   with S:a, D:b
                   (g (lambda (x) x)))
               (list (f g a b) (f g b a)))

all-versions(cls(f)) = { ((D→D)xSxD→D), ((D→D)xDxS→D) }
```

More disconcertingly, binding time signatures may even be infinitely nested :

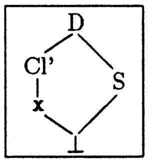
```
main (n)   = (wrap (lambda (x) x) n)  with S:n
wrap (g n) = (if (zero? n) (g 1) (wrap (lambda (x) g) (- n 1)))

all-versions(cls(g)) = { (S→S), (S→(S→S)), (S→(S→(S→S))),..}
```

The function `all-versions` must of course only return a finite number of abstract signatures and we need therefore a way to detect, represent and handle infinite signatures.

We introduce a new binding time lattice BT' which is the smallest set containing {⊥, S, D, Cl'} and closed under the following operation

$$x_1^1,\ldots,x_{j_1}^1, xr^1, \cdots, x_1^i,\ldots,x_{j_i}^i, xr^i \in \mathrm{BT}' \Rightarrow$$
$$\mathrm{Cl}_{\{}x_1^1 \times \ldots \times x_{j_1}^1 \rightarrow xr^1, \cdots, x_1^i \times \ldots \times x_{j_i}^i \rightarrow xr^i\} \in \mathrm{BT}'$$

The ordering in the lattice is given by the following figure where **x** is any element of BT' not in {⊥, S, D, Cl'}. The new symbol Cl' represents the upper bound of all closures.



Two elements X and Y of BT' different from ⊥, S, Cl', D are ordered as defined below. Y is superior to X if it continuously extends all signatures of X. The least upper bound of X and Y will be written X ⊔ Y.

$$X, Y \in \mathrm{BT}' - \{\perp, S, D, Cl'\}$$
$$X \leq Y \Leftrightarrow \forall d \geq 0. \forall x == (x_1 \times \ldots \times x_d \rightarrow xr) \in X.$$
$$\exists y == (y_1 \times \ldots \times y_d \rightarrow yr) \in Y.$$
$$(x_1 \times \ldots \times x_d) = (y_1 \times \ldots \times y_d) \wedge xr \leq yr$$

The function ϕ below takes a simple binding time value and the closure set of an expression and returns a more fine-grained, yet finite binding time value in BT'. It allows us to define `all-versions`.

```
all-versions : Closure-Set → BT'
all-versions(cls) = $\phi$(Cl, cls)

$\phi$ : BT × Closure-Set → BT'
$\phi$(⊥, _)    = {⊥}
$\phi$(S, _)    = {S}
$\phi$(D, _)    = {D}
$\phi$(Cl, {i}) =
  if infinite-binding-time?($i$) then
    {Cl'}
  else
    let v* == fetch-versions(i) in
      $\bigcup_{v \in v*}$ let $b_1 \times \ldots \times b_n$ == param-bt(v),
                 $br$                == result-bt(v),
                 $c_1 \times \ldots \times c_n$ == param-closure-set(v),
                 $cr$                == result-closure-set(v)
             in let $P_1 == \phi(b_1, c_1), \ldots, P_n == \phi(b_n, c_n)$,
                    $Pr == \phi(br, cr)$
                in $\{(p_1 \times \ldots \times p_n \rightarrow Pr) \mid p_i = P_i$ if $P_i \in \{\perp, S, D, Cl'\}$
                                              $p_i \in P_i$ otherwise }
$\phi$(Cl, I∪J) = $\phi$(Cl,I) ⊔ $\phi$(Cl,J)
```

The explicit test `infinite-binding-time?` is the key to the finiteness of the result of ϕ. Whenever a binding time is known to be infinite, it is approximated by Cl'. Thanks to the results of the closure analysis, finiteness of binding times of abstractions can be easily verified : it is sufficient to check the graph of closure annotations of a lambda-abstraction for cycles. The previous example would have been annotated in the following way by the CLA :

```
main (n)           = (wrap {0}:(lambda_0 (x) x) n)
wrap ({0,1}:g n) = (if (zero? n)
                         ({0,1}:g 1)
                         (wrap {1}:(lambda_1 (x) {0,1}:g) (- n 1)))
```

We observe that the body of the lambda-abstraction **1** returns the closure set **{0,1}** i.e. it may return itself and will therefore receive the binding time value Cl'.

We now have all the tools to decide, at the point of a procedure call, whether a variant of a procedure may safely be shared or if a widening might occur.

6.3 Example : Polyvariance for the functional parameters of a procedure

In 5.4 a new variant was created, when a new abstract tuple of arguments for a procedure was encountered. Now we use the extended binding time lattice BT' to compare abstract tuples. Let's consider an example :

```
main (a b) = (let* ((f (lambda (x) (* x x)))    with S:a,D:b
                    (g (if (odd? a) (lambda (x y) (+ x y)) f))
                    (h (lambda (x) (* x b))))
               (list (f b) (test f a b #f) (test g a b (odd? a))
                     (test h a b #f)))
test (u a b c?) = (if c? (u a b) (u a))

f : { (S→S),(D→D) },  g : { (S→S),(D→D), (SxD→D) },
h : { (S→D) }     ⇒ f,g are not compatible with h

main_ann (a b) = ...(list (f b) (test_1 f a b #f) (test_1 g a b (odd? a))
                           (test_2 h a b #f))
```

Note that the procedure `test` has been duplicated once, but that it is still shared by the first two calls because the versions of `f` and `g` are compatible.

Functional arguments passed to lambda-abstractions are treated analogously.

6.4 Finiteness

The argumentation given in 5.5 is still valid if the number r of possible binding time signatures for any function is bound. This is the case as the result of the function ϕ (6.2), on which the duplication of expressions is based, is finite.

7 Complexity & Efficiency

The complexity of the CLA and the BTA depend on the number of iterations needed to reach a fixed point : both do repeated traversals of the source program, the complexity of the traversal being proportional to the length of the program. Repeated traversals are needed when the annotation of some expression depends on annotations not yet established. In the worst case, there is a long chain of dependencies treated in the inverse order by the BTA. The length of such a chain determines the number of iterations needed before a fixed point is reached. The following program needs n iterations of the BTA since the bodies of the lambda-abstractions are analysed before they are applied.

```
t1   (a) = (let ((f (lambda (x) x))) (t2 (f a)))
...
tn-1 (a) = (let ((f (lambda (x) x))) (tn (f a)))
tn   (a) = a
```

Our polyvariant analysis increases the complexity because it may

- duplicate expressions, increasing the size of the source program,
- need several CLA & BTA passes.

We stated in 5.5 that at most r^d variants/versions may be generated for a procedure/lambda-abstraction of arity d, r being the number of binding time values. The following program generates $3^4 = 81$ variants for procedure **test** :

```
main (a b)     = (test a b (lambda () a) a)    with S:a, D:b
test (a b c d) = (if #t 'ok
                    (list (test a a a a) (test a a a b) (test a a b b)
                          (test a b c c) (test b c d a) (test a b d c)))
```

In general, it is not possible to reduce the number of variants/versions generated without sacrificing some efficiency of the residual program.

As explained in 5.3.2 an expression duplication may invalidate the results of the CLA and thus necessitate redoing the CLA & BTA. It would however be extremely costly to stop the BTA, reset all annotations and restart a new CLA & BTA cycle after each duplication of an expression. Instead we adopt the following scheme :

- The BTA always continues until reaching a fix-point. If new variants/versions have been created or if the flow of control has changed during the BTA, we restart a CLA & BTA cycle.
- Even when the the results of CLA have been invalidated, our BTA never over-estimates any binding time values. Thus, binding time annotations need not be reset between iterations, leading to faster convergence of the BTA in subsequent runs.

To achieve this, we keep track of the abstractions for which the CLA-annotations have become invalid and we don't apply these abstractions any more, thus effectively underestimating some binding time values. When no duplications are created, all abstractions remain valid and the desired fix-point is reached.

A polyvariant BTA has two sides : on one hand it is more precise, leading to more efficient residual programs, on the other hand it may considerably (or catastrophically) increase the time and space requirements of the BTA. Our BTA is based on detecting *potential* loss of information, without any analysis of effectiveness. Moreover, a more precise analysis also increases the risks of infinite specialization, since there is more static information with respect to which procedures may be specialized.

These negative aspects suggest the introduction of a means of manual annotation by the programmer to guide the BTA. Currently this is possible using the **generalize** operator [Bon90] which acts as an identity operator but forces its argument to become dynamic. *Filters* as used in Schism [Con89] would ultimately be more versatile and are conceptually simpler. A polyvariant BTA might also warn the user if a given number of variants is reached and indicate which parameters have different binding times.

8 Extending the BTA to partially static values

If partially static values, like a cons of a static and dynamic argument, are considered being entirely dynamic (as does Similix) much information may be lost and the programmer has to use tricks (like splitting an a-list or using continuations for multiple return values) to obtain efficient residual programs. We have developed an extension of Similix' BTA, that handles partially static conses. Other constructed data-types (e.g. vectors) could be easily included.

The extension is based on a *Cons-Point Analysis* [Con89] and is done together with the CLA : After indexing unambiguously all points of construction (i.e. cons-operators), every expression is annotated with the set of construction-points that may be returned by this expression.

This annotation is used by the new BTA to maintain structured binding time information. Recursively defined data-structures are easily detected and are handled correctly to keep the polyvariant BTA finite.

9 Examples

9.1 Self-Application

Self-application permits not only a realistic test with a fairly complex program, but also verifies that the generation of compilers is still feasible. Since the original specializer had been written with a monovariant binding time analysis in mind, the residual programs produced by our polyvariant system are not significantly improved. Here are some results showing the number of CLA & BTA iterations, the passes CLA and BTA needed to reach the fixed point and the sizes of the program.

Pass	1	2	3	4	5	6	7	Final
Prog. Size (K)	19	41	110	155	220	280	280	130
CLA Passes	4	5	5	5	5	5	5	-
BTA Passes	4	4	6	4	2	2	1	-

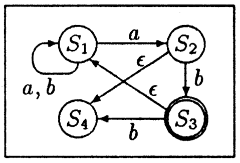
With respect to a the monovariant analysis, the preprocessing time for the polyvariant analysis has increased by a factor 75 for the CLA and by a factor 25 for the BTA.

9.2 A toy interpreter

FUNBACK is an interpreter for a strict functional language with back-tracking (and the related `fail` and `cut`), implemented in continuation passing style [Sch86]. The language, close to `ML` and `Prolog`, defines procedures as clauses which are called by pattern matching.

The interpreter may be specialized with respect to a program, but without knowing the main function (goal). In this case, Similix treats all goals (including the ones defined by the source program) as unknown, while the polyvariant BTA distinguishes between known and unknown goals.

If the goal is known, both Similix and our polyvariant partial evaluator produce similar results, with a slight advantage for the polyvariant evaluator because of minor binding time improvements. The following automaton



may be described in FUNBACK as :

```
accept (S '())          = (final S)
accept (S (cons X R)) = (let ((N (trans S X))) (accept N R))
accept (S R)            = (let ((N (silent S))) (accept N R))
final ('S3)             = 'ok
trans ('S1 'a)          = 'S1
...
silent ('S2)            = 'S4
...
```

Comparing FUNBACK to Similix and our evaluator, we get the following result for an acceptable string of length 20 for which there are 128 ways to be recognized :

	Time (s)	Size (cells)	Speedup over FUNBACK	Increase factor in size
FUNBACK	19.2	162	-	-
Goal Known				
Similix	1.14	1057	16.8	6.5
Polyvariant	0.95	930	20.2	5.7
Goal Unknown				
Similix	2.10	1799	9.1	11.1
Polyvariant	1.10	2100	17.5	13.0

Both partial evaluators reduce a significant part of the interpretation. The polyvariant system yields better results in execution time especially for specialization with unknown goals.

10 Conclusion

In this paper, we have presented a polyvariant BTA for higher-order programs. We have described the situations in which undesired widening may occur. Duplicating expressions during the BTA allows for multiple annotations and can help avoid widening.

We have shown that a fine-grained binding time lattice is required to avoid certain cases of widening. Recursive binding times must be detected in order to avoid infinite duplications. Since our approach concentrates on the preprocessing phases, with only minor modifications to the specializer itself, self-application still yields efficient compilers. We have shown bounds for time & space requirements of our method and that it behaves reasonably in practice.

Our polyvariant BTA is another step towards making partial evaluation practically useful as a bigger class of programs now can be specialized to more efficient residual programs. This frees the programmer from knowing the intricacies of the partial evaluation tool.

10.1 Comparison with other work

Our work is based on the partial evaluator Similix-2 [Bon90]. Similix contains a so-called "binding time debugger" [Mos91], that can signal undesired widening, but leaves the duplicating and adjusting to the programmer.

[Con89] discusses a polyvariant BTA. The method identifies fully connected sub-graphs in the call graph and only creates new instances of procedures for calls between different sub-graphs. This ensures a finite number of procedure variants, but still may lead to widening. It does not address higher-order programs.

[NBV91] use a form of expression duplication (called bifurcation) to allow partially static values in a (polymorphically) typed functional language.

10.2 Future work

To make the preprocessing phase more efficient, the CLA & BTA should be incremental i.e. more annotations should be conserved between iterations. We need also a method to help making trade-offs between the complexity of the preprocessing and the quality of the residual programs. These could be made by a heuristic estimating the usefulness of a duplication or through annotations by the programmer.

References

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation*. Ph.D. Thesis, University of Copenhagen, Denmark, December 1990.

[Bon91b] A. Bondorf. Automatic autoprojection of higher order recursive equations. In *Science of Computer Programming*, Vol. 17, December 1991.

[Con89] Ch. Consel. *Analyse de Programmes, Evaluation Partielle et Générateur de Compilateurs.*, Ph.D. Thesis, LITP, Université Paris 6, 1989.

[Con90] Ch. Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Languages*, June 1990.

[Fut71] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. In *Systems, Computers, Controls*, 2(5), pp. 45-50, 1971.

[Mos91] Ch. Mossin. *Similix Binding Time Debugger Manual, system version 4.0*. Included in Similix distribution, August 1991.

[NBV91] A. De Niel, E. Bevers, K. De Vlaminck. Program Bifurcation for a Polymorphically Typed Functional Language. In *Proceedings of the 1991 ACM Symposium on Partial Evaluation and Semantics-Bases Program Manipulation*, June 1991.

[RC91] J. Rees, W. Clinger (editors). *Revised^4 Report on the Algorithmic Language Scheme*. CS Department Technical Report, MIT, 1991.

[Ryt92] Bernhard Rytz. *Polyvariant Binding Time Analysis for Partial Evaluation*. Technical Report EPFL LITH (to appear).

[Sch86] David A. Schmidt. *Denotational Semantics. A Methodology for Language Development*. Allyn and Bacon, 1986.

Towards a New Perspective on Partial Evaluation

Morry Katz
Computer Systems Laboratory
MJH 416A
Stanford University
Stanford, CA 94305
(katz@cs.stanford.edu)

Daniel Weise*
Computer Systems Laboratory
MJH 426
Stanford University
Stanford, CA 94305
(weise@cs.stanford.edu)

1 Introduction

We have designed a new method for performing partial evaluation. This method divides specialization into two phases: a polyvariant analysis phase and a code generation phase. The analysis phase does not differentiate between unfolding and specializing. *Symbolic execution* is the sole operation performed during the analysis phase. Symbolic execution of an expression yields a characterization of the value that would be returned by the expression if it were executed at runtime, a description of the residual operation(s) that must be performed to generate the runtime value, and a record of the information utilized in performing symbolic execution. (The third result of symbolic execution will be explained in some detail later.) Only delta-reductions are performed during symbolic execution. No beta-substitution of user functions occurs during the analysis phase and no values are in-lined. Consequently, symbolic execution yields extremely polymorphic, and highly reusable, *specialized function bodies*. The code generation phase constructs the residual program from the specialized function bodies. It is responsible for all beta-substitution. Delaying the beta-substitution decisions until the code generation phase allows us to construct a partial evaluator that both produces highly specialized code and makes intelligent decisions regarding code size versus lower function call overhead.

Termination decisions are made in the analysis phase based on *lazy use-analysis*, an extension of eager use-analysis [5]. Lazy use-analysis considers both the information used by symbolic execution in performing delta-reductions and how information about the values returned by delta reductions is used. Information is only used in a fundamental sense if there is a causal chain between its use in performing some delta reduction and the production of the final runtime answer returned by a program. The less information used in creating a specialized function body that contributes to the return value of the function, the greater the number of contexts in which the specialization can be reused. Furthermore, the less information used about the value returned by a function call, the less restrictions that are placed on the characteristics of the specialization to be called. A partial evaluator based on lazy use-analysis promises to produce a better combination of quality residual code and termination than previous alternatives. For example, a partial evaluator constructed using lazy use-analysis safely handles the standard "counter" problem[1] and properly unfolds Mogensen's (unpublished) regular expression accepter. No automatic partial evaluator to date has been able to do both.

The remainder of this paper is divided into three sections. The first presents a classification scheme for the types of divergences that partial evaluators experience and a discussion of different forms of termination (reuse) mechanisms. Next, use-analysis and its application to termination and reuse mechanisms are presented. The paper closes with our plans for future research and some conclusions.

2 Termination

2.1 Types of Divergence

We divide the types of divergence experienced by partial evaluators into three categories: *true divergences*, *hidden divergences*, and *induced divergences*. A program is considered divergent if normal execution of the program diverges for some valid program inputs. A partial evaluator is said to experience *true divergence* when it diverges processing a divergent input program and an input specification that includes values on which the input program diverges. Some convergent programs contain divergent segments that can never be reached for any valid set of inputs. If control ever reached these divergent pieces of code, the programs would diverge. We say such programs contain a *hidden divergence*. A partial evaluator may diverge because it stumbles across a hidden divergence during symbolic execution due to the partial nature of an input specification.

All other divergences of a partial evaluator are called *induced divergences*. The divergence does not exist in the input program, but is induced by the partial evaluation strategy being utilized. Induced divergences result from a partial evaluator pursuing a control path that could never be followed by a runtime evaluator. Some induced divergences result from bad termination strategies; others, from an inability of the partial evaluator to prove that a control path could never be taken.

2.2 Types of Termination Mechanisms

We classify termination schemes based on induced divergences. *Termination priority (TP)* partial evaluators utilize

*Supported in part by NSF Contract No. MIP-8902764, and in part by Defense Advanced Research Projects Agency Contract No. N0039-91-K-0138.

[1]Many partial evaluators fail to terminate on programs containing a recursion in which a static argument is incremented between each recursive call (e.g., counting up factorial) when the recursion cannot be completely unfolded during partial evaluation.

termination strategies that ensure that no induced divergences will take place. *Residual code priority (RCP)* partial evaluators allow some induced divergences in an attempt to yield better residual code. In principle, a TP partial evaluator should be able to produce 'optimal' residual code in some cases. In practice, existing RCP partial evaluators produce better code than their TP cousins. We propose a partial evaluator based on a new form of analysis that is aggressive, yet attempts to minimize the classes of programs on which induced divergence will take place. Our reasons for taking the RCP approach are discussed in more detail later.

2.3 Essence of Termination

Termination is only an issue for recursive programs, as symbolic execution of non-recursive code always terminates. When a partial evaluator encounters a recursive function call,[2] the question to be answered is whether continued symbolic execution of the recursion will terminate. If symbolic execution will terminate, then it is safe for the partial evaluator to completely symbolically evaluate the recursion, and this will yield the best possible residual code. If symbolic execution will diverge, then it is desirable for a partial evaluator to discontinue symbolic execution and produce a residual loop. Unfortunately, it is not decidable whether symbolic execution will terminate.

All termination mechanisms use some concept of equivalence in deciding whether to continue symbolic execution or produce residual code. When a recursive call is reached, the current iteration of the loop is compared with previous iterations. If two iterations are deemed equivalent, then symbolic execution of the loop is discontinued and a residual loop is produced. If all iterations are deemed distinct, then symbolic execution is continued. TP partial evaluators use equivalence metrics that place all iterations of a loop into different equivalence classes only when symbolic execution of the loop terminates or the loop is truly divergent. RCP partial evaluators use equivalence metrics that only place two iterations of a loop in the same equivalence class if no improvement in residual code quality would result from continuing symbolic execution.

It is enlightening to think about termination of symbolic execution of recursions in terms of fixed points. Deciding whether to continue symbolic execution of a recursion is basically the same as determining whether symbolic execution of the recursion has reached a fixed point with respect to the creation of new, unique specialized function bodies. Once a fixed point has been reached, continued symbolic execution of the recursion would lead to divergence since the data available for making the termination decision is by definition not changing and the algorithm has previously decided based on this data to continue symbolic execution. The central issue, therefore, is how to define an equivalence metric that captures the concept of symbolic execution having reached, or not reached, a fixed point.

Two specializations are completely equivalent if one specialization can be replaced by the other (in any context) without changing a program's behavior. Complete equivalence is overly restrictive. A more liberal equivalence that gets to the essence of partial evaluation is: two specializations are equivalent with respect to a given call site in a residual program if a call to one specialization can be replaced by a call to the other without changing a program's behavior. Note that an original function and any nontrivial specialization of it are not equivalent in the first sense, but can be equivalent in the second.

2.4 Enlarging Equivalence Classes

Virtually all partial evaluators characterize specializations based upon the information in the arguments that the specializer was *allowed to use* when constructing the specialization.[3] For example, in offline specializers such as Similix [1] and MIX [4], specializations are characterized by the information in the static arguments; in online specializers such as Fuse [8], all of the information in the arguments characterizes the specialization. These characterizations form equivalence classes of arguments. A specialization can be safely reused at any call site where the information allowed to be used during specialization is present at the call site (i.e., wherever the arguments are in the equivalence class of the specialization). Regardless of how they are created or the context from which they are invoked, two specializations of the same function are equivalent when the equivalence classes of their arguments are equal.

Recently, Ruf showed how to enlarge the equivalence class (reusability) of a specialization by characterizing it by the information in the arguments that was *used to construct* the specialization [5]. He invented the term *domain of specialization* (DOS) to name the equivalence class a specialization can be safely reused on, and showed how to compute a safe approximation to the DOS (called the MGI) by tracking the information used to construct a specialization. Ruf showed that the larger equivalence classes that his methods constructed for a specialization significantly increased the reuse and sharing in residual code without degrading the quality of residual code.

The DOS defined by Ruf is not as large as possible because it assumes all the information contained in the value returned by a specialized function at runtime will be used; however, this is often not the case. The less information that is used about the return value of a specialization by some context, the larger the equivalence class of specializations that can be called from that call site. We will refer to the DOS defined in [5] as the *context free domain of specialization* (CF-DOS) to differentiate it from the *context sensitive domain of specialization* (CS-DOS) that accounts for both the values to which the specialization is applicable and how its results can be used. When a function is called from a context that doesn't use all the information available about its return value, it is important to know if a specialization already exists (even one returning a different value) that makes it reusable from this call site. The CF-DOS is not as useful for this purpose since it is a characterization based on information that may not be used in this context. That is, the CF-DOS may be too small to be of use, in particular, too small to ensure termination.

A direct consequence of the definition of the CS-DOS is that only information that *causally contributes to generating the result of a program* appears in the definition of any equivalence class. This follows from each specialization being characterized by the context in which it is called, which is in turn characterized by the context in which it is called, etc. We argue that the CS-DOS is the largest possible equiv-

[2]Not all partial evaluators distinguish recursive calls from non-recursive calls. This observation doesn't affect our argument.

[3]The significant exception to this rule is systems using manual annotations such as Schism [2].

alence class that can be used by a partial evaluator without degrading the quality of residual code. (i.e., The CS-DOS characterizes specializations with the greatest potential for reuse that does not impact the ability to create useful specializations.)

To effectively approximate the equivalence classes of the CS-DOS, a partial evaluator should place two specializations of a function that depend on identical information from two separate argument vectors in the same equivalence class. Two otherwise equivalent specializations lose their equivalence if beta substitutions of values into the specializations yields residual code with differing equivalence classes. This means specialization must not perform unnecessary beta-substitution.

A partial evaluator that bases its control decisions on an accurate estimate of information usage can produce a better combination of residual code quality and termination properties than one that does not. For example, a TP partial evaluator could use knowledge of the information that would be used in forming a specialization to prove that symbolic execution of a function call would result in a new, unique specialized function body. This would enable a TP partial evaluator to be more polyvariant (i.e., create more useful specializations and perform more unfolding) while still maintaining its termination properties.

Conversely, an RCP partial evaluator that exploits the property that two specializations only fail to be interchangeable when different information about the available values was utilized in creating them can better decide whether symbolic execution of a function body will result in a new specialization. For example, whereas Similix assumes that progress is being made towards producing a new specialization as long as no introduced function is called with identical static arguments [1], an information based partial evaluator would only distinguish between argument sets that differed in some piece of information to be used in forming the specialization. This enables the RCP partial evaluator to correctly terminate more often.

2.5 Equivalence Classes and Termination

We propose using equivalence classes based upon use information as our sole termination method, unlike previous methods that appeal to *dynamic conditionals*, such as Fuse and Similix, use induction methods, such as Mix [6], use finiteness criteria, such as [3], or use manual annotations, such as Schism [2]. It is vital that we construct the largest classes possible without degrading residual code quality. In particular, to ensure termination, it is important that the equivalence class of a specialization not be restricted simply because the value of some variable has been inlined in the specialization, if inlining has not allowed for further optimization. Gratuitous inlining decreases the size of equivalences classes, and induces unnecessary distinctions between specializations.

For example, consider a procedure, called `check-numeric-type`, that signals an error if its sole argument is not numeric, and is otherwise the identity function (Figure 1). The application of `check-numeric-type` to the value 1 can yield two different specializations. Performing all possible beta reductions yields a specialization that just returns the value 1 (Figure 2). A more polymorphic specialization is the identity procedure that remains parameterized over the argument value, which must be numeric (Figure 3). A partial evaluator that generates the less polymorphic form of specializations could not reuse the specialization generated for `(check-numeric-type 1)` for a call to `(check-numeric-type 2)` in any context that uses the return value of the call.

```
(define check-numeric-type
  (lambda (val)
    (if (number? val)
        val
        (error val))))
```

Figure 1: A function for checking that an argument is numeric

```
(define check-numeric-type
  (lambda (val) 1))
```

Figure 2: A maximally specialized version of `(check-numeric-type 1)`

To detect the equivalences required for effective termination, we propose dividing the specialization process into two phases. During the first (analysis and symbolic execution) phase of partial evaluation, the specializer only incorporates into specializations that information about values that is necessary to perform ***further optimizations*** during symbolic execution. In other words, it would produce the more polymorphic specialization in Figure 3 for `(check-numeric-type 1)`. Only type information about the value 1 is used in evaluating the predicate of the conditional in `check-numeric-type`, and no information is used in the consequent. The specializer would not inline the value since doing so enables no further optimizations, and would restrict the applicability of the specialization (i.e., reduce the size of its equivalence class). Additional inlining decisions are delayed until the code generation phase. During code generation all of the call sites of a specialization are known, which determines the maximum amount of code sharing that is possible after the analysis phase has completed. At that time the traditional inlining trade-off between code size and execution efficiency can most effectively be made.

An added benefit of our two phase approach based on use-analysis is that the partial evaluator can choose to produce code for more or fewer specializations than it generated during the analysis phase. When a single specialization is only applied to a small number of elements of the equivalence class to which it is applicable, it may choose to generate a separate specialization for each element of the equivalence class that appears in the code. For example, a specialization might be applicable to all numbers, but only called with the arguments 0 and 1. Instead of producing a single specialization, it might be better to produce two specializations by inlining one of the possible argument values in each. This would yield two specializations: one applicable to 0 and the other applicable to 1. Conversely, the partial evaluator may occasionally combine several specializations produced during the analysis phase into a single, more general specialization. An important property of use-analysis is that it produces code that is as polyvariant as desirable

```
(define check-numeric-type
  (lambda (val) val))
```

Figure 3: A maximally polymorphic specialization of (check-numeric-type 1)

without forcing the code generation phase to produce more code than is necessary.

3 Use-Analysis

3.1 The Use-Analysis Framework

Use-analysis maintains an approximation to the information used by a partial evaluator during symbolic execution. Use information can be represented by points in an information lattice. Figure 4 shows the value domains for a pure subset of Scheme. A richer set of value domains is required for representing use information. They are needed because information about a value other than its identity is often used in forming a specialization. We add values to the Scheme domains that represent unspecified members of existing domains, similar to the *symbolic values* used by Fuse[8]. For example, use of only the integer property of the number 3 in a specialization might be represented by the abstract value $\bot_{Int}$. A complete set of value domains for a partial evaluator based on this concept is presented in Figure 5.

An information lattice based on the domains in Figure 5 and the binary relation $\prec$, meaning *has less information than*, is presented in Figure 6.[4] $\bot_{PEval}$ represents having no information about a value. The first clause in the lattice description states that there is information contained in the type of a value. The next three clauses express that there is more information in the identity of a value than in its type. Pairs are organized in an information hierarchy based on the information known about the car and cdr of the pair. The more information known about the two components of the pair, the more information that is known about the pair, itself. Finally, the last clause states that $\top$ represents more information than any other lattice element.

There are many types of information about values that cannot be precisely represented by nodes in the information lattice of Figure 6. For example, the best representation of the information that an integer is less than 5 is either the integer's identity or $\bot_{Int}$. The integer's identity is an overspecification, excluding other integers that are less than 5. $\bot_{Int}$ is an underspecification, including integers greater than or equal to 5. The effects on partial evaluation algorithms of different choices of information lattices and means of imprecisely representing information usage utilizing them will be discussed later.

Our presentation of use-analysis will proceed in three stages. First, an eager use-analysis that is fairly simple to explain and understand will be developed. A fully lazy use-analysis will then be discussed. Finally, lazy use-analysis that eliminates some of the shortcomings of eager use-analysis will be presented.

[4]The orientation of an information lattice is arbitrary in that one can either have more information represented by higher or lower points in the lattice. The orientation we have selected is opposite of the one used by Ruf in [5]

Eager Use-Analysis

A partial evaluator uses information about data values when performing computations during symbolic execution. Eager use-analysis, first presented in [5], records usage information as soon as information about a value is used in performing a computation. The information about a value that is utilized is represented by an element of an information lattice. Values in a partial evaluator are replaced by annotated values that are pairs composed of a representation of the information known about an object's value and a use annotation for that value. When the partial evaluator symbolically executes a primitive function on a set of annotated values, a new value is produced and the use information of the annotated argument values is updated. Use-analysis seeks to record the maximum amount of information about a value that is utilized, so primitives only modify the use information field of an annotated value when they use more information about that value than has previously been used.

A table of Scheme function applications and use profiles for their arguments based on the information lattice presented earlier (Figure 5) is shown in Figure 7. Less trivial use profiles result when multiple functions are composed. For example, (number? (car (car '((1 2) (3 4))))) results in a use profile that might be represented as $<< \bot_{Int}, \bot_{PEval} >, \bot_{PEval} >$. The innermost car uses the information that the argument is a pair. The next car uses the information that the first component of the pair is also a pair. number? uses the information that the first component of that pair is an integer.

Eager use-analysis often produces an overestimation of the information required to produce a given specialization. For example, in the expression (number? (+ 1 2)), + uses the identity of both of its arguments to produce the result 3 and modifies the use annotations of the arguments to reflect this usage. The function number? only uses that 3 is an integer, not its value. This function would return the same result for any two integer arguments to which + were applied. The specialization only depends upon the types of 1 and 2, not their identities (values). Embedding the example expression in a larger expression leads to yet greater overestimation of information usage. In the expression ((lambda (a b) a) #t (number? (+ 1 2))), no information about 1 or 2 is used since the result of number? is thrown away; however, the use annotations still show that the identities of these integers are used. While the code examples shown may seem unrealistic, similar expressions do appear in real programs because of macros and other abstraction mechanisms.

Fully Lazy Use-Analysis

Eager use-analysis has the flaw of recording information usage of intermediate computations that do not contribute to the result of a program. Ideally, only information utilized in generating the final result should be recorded. Information contributes to the result in one of two ways, either by affecting the data or the control flow of a program. How information about a value percolates through the data flow of a program to affect the final result is fairly obvious. Values also affect the result of a program when they are used to make a control flow decision (e.g., the predicate of a conditional). If a control flow decision affects the final result of a program, then the information used to make the control flow decision has been used in generating the program's result.

Int	$=$	$0, \pm 1, \pm 2, \cdots$	integers
$Bool$	$=$	$\mathtt{true} + \mathtt{false}$	booleans
Nil	$=$	$\mathtt{nil}$	empty list
$Pair$	$=$	$Sval \times Sval$	pairs
$Func$	$=$	$Sval^* \rightarrow Sval$	function values
$Sval$	$=$	$Int + Bool + Nil + Pair + Func$	scheme values

Figure 4: Value domains for a subset of Scheme

Int	$=$	$0, \pm 1, \pm 2, \cdots$	integers
		$\bot_{Int}$	unspecified integer
$Bool$	$=$	$\mathtt{true} + \mathtt{false}$	booleans
		$\bot_{Bool}$	unspecified boolean
Nil	$=$	$\mathtt{nil}$	empty list
$Pair$	$=$	$PEval \times PEval$	pairs
		$\bot_{Pair} = \bot_{PEval} \times \bot_{PEval}$	unspecified pair
$Func$	$=$	$PEval^* \rightarrow PEval$	function values
		$\bot_{Func}$	unspecified function
$Kval$	$=$	$Int + Bool + Nil + Pair + Func$	known values
$Bots$	$=$	$\bot_{Int} + \bot_{Bool} + \bot_{Pair} + \bot_{Func}$	bottom values
$PEval$	$=$	$Kval + Bots + \bot_{PEval}$	partial evaluation values
		$\bot_{PEval}$	unspecified value

Figure 5: Value domains for information

$$\begin{array}{l}
\forall x \in (Bots \cup Nil).(\bot_{PEval} \prec x) \\
\forall i \in Int.(\bot_{Int} \prec i) \\
\forall b \in Bool.(\bot_{Bool} \prec b) \\
\forall f \in Func.(\bot_{Func} \prec f) \\
\forall <x, y> \in Pair.(\forall x' \prec x.(<x', y> \prec <x, y>)) \\
\forall <x, y> \in Pair.(\forall y' \prec y.(<x, y'> \prec <x, y>)) \\
\forall <x, y> \in Pair.(\forall x' \prec x.(\forall y' \prec y.(<x', y'> \prec <x, y>))) \\
\forall k \in (Kval \cup Bots).(k \prec \top)
\end{array}$$

Figure 6: Information lattice description

Expression	Argument Use Profiles
`(+ 1 2)`	1,2
`(number? 3)`	$\bot_{Int}$
`(car '(1 . 2))`	$\bot_{Pair}$
`(boolean? #t)`	$\bot_{Bool}$

Figure 7: Eager use annotations

In fully lazy use-analysis a complete model of the execution of the entire program must be built before a use-analysis can be performed. Only then can it be determined if there is a causal chain from the use of some information at one point in a program's execution all the way to the final result. While fully lazy use-analysis would be ideal, it is unfortunately unknown at this time how to implement it, or even if it is theoretically computable.

Lazy Use-Analysis

Lazy use-analysis is an approximation to fully lazy use-analysis that solves many of the problems associated with eager use-analysis. Lazy use-analysis delays deciding whether information about values has been used until the information usage contributes to making a control flow decision. It differs from its fully lazy cousin in that all control flow decisions immediately assert the usage of information, even if they eventually turn out not to contribute to the final result. Lazy use-analysis works backwards from a point where information is used to make a control flow decision back through the data flow to the original sources of the information used in making that decision. Lazy use-analysis maintains a record relating the information usage of one value to the information usage of another value. For example, execution of the expression (number? 3) creates a link between using the value of the boolean result and using the type of 3. Execution of (+ 1 2) creates links that express two properties: use of the value of the result implies use of the values of both 1 and 2, and use of the type of the result implies use of the types of both arguments (i.e.,, that they are integers).

When a value affects a control flow decision, lazy use-analysis records usage information about that value using an annotation from an information lattice. If the value utilized in the control flow decision is linked to other values, then the values to which it is linked also have information usage recorded about them. They, in turn, may pass on usage information to other values to which they are linked. This process is continued until all necessary values have had their usage profiles updated. Since the use links follow the data flow of the program, they must be acyclic and the process is guaranteed to terminate.

3.2 Termination Based on Use-Analysis

Our termination mechanism defines its equivalence classes in terms of information usage profiles. When the first recursive call to a function is detected, an information based partial evaluator has already acquired information about the first iteration of the loop. However, the partial evaluator must continue symbolic execution until a second recursive call is made, because two equivalent calls (iterations) are required to detect a fixed point. Once the second recursive call is reached, the question is whether the two iterations are equivalent with respect to information usage. If the iterations are equivalent, then the analysis process has reached a fixed point, and a residual loop will be produced. If the iterations are not equivalent, then symbolic execution should continue until either the recursion terminates (i.e., is completely unfolded) or a new iteration which is equivalent to some previous iteration is detected.

We divide each recursive procedure into two segments. The *head* is the portion of the procedure performed before the recursive call; the *tail*, the portion performed after. The termination algorithm described so far has only utilized use information about the head of a recursion in deciding whether two iterations are equivalent. In attempting to produce a specialized function body, it may be determined that two iterations are in actuality not equivalent because their tails are not equivalent. In this case, symbolic execution of the recursion is resumed and proceeds as outlined above. A final residual, recursive loop is only produced when both the head and the tail of two iterations of a recursion are found to be equivalent.

An ideal partial evaluator maintaining perfect use information would suffer from no induced divergences and could produce as good residual code as is possible from any physically realizable system. Perfect use information would require a fully lazy analysis based on a set of value domains that could capture precisely the information used by all primitives in the language being specialized. This type of optimality arises because fully lazy use-analysis captures perfectly the applicability of all specializations based on the available data. Therefore, when symbolic execution of a recursion reaches a fixed point (finds two equivalent iterations) based on this perfect form of use-analysis, it is guaranteed that two iterations are fundamentally equivalent and that no alternative means of specialization could cause them to be nonequivalent.

Imperfect use profiles result in either TP or RCP partial evaluators. Information profiles that always over-record information usage yield RCP partial evaluators. Over-recording information usage can cause two equivalent iterations of a loop to appear distinct. This means the partial evaluator may not recognize that the symbolic evaluation process has reached a fixed point, and may diverge trying to completely unroll the recursion. Under-recording information usage yields a TP partial evaluator. Iterations of a recursion that are not in actuality equivalent may appear to be equivalent to such a partial evaluator. This means that an under-recording partial evaluator may give up unrolling a loop before the symbolic execution process has actually reached a fixed point, which results is an overly general equivalence class and a suboptimal specialization. However, under-recording can never lead to more symbolic execution than would result with perfect information profiles, so it must terminate whenever the ideal information based partial evaluator does. Since an ideal partial evaluator does not suffer from induced divergence, an under-recording partial evaluator will not, either.

4 Two Termination Examples

Lazy use-analysis enables a partial evaluator to produce higher quality code than many existing partial evaluators while incurring fewer induced divergences than other RCP partial evaluators. This is demonstrated through two example programs. No other automatic partial evaluator both terminates on the Iota program and produces good residual code for the Regular Expression Acceptor.

```
(define iota
  (lambda (n)
    (define loop
      (lambda (i)
        (if (= i n)
            '()
            (cons
             i
             (loop (1+ i))))))
    (loop 0)))
```

Figure 8: Iota function

4.1 Effective Termination

Partial evaluation of the iota function in Figure 8 with respect to an unknown value for n causes an induced divergence for Similix[1]. This system defines equivalence in terms of the identity of the values to which parameters that have been labeled as *static* by a *binding time analysis*[4] are bound. The salient issue is that it parameterizes its specialization points by the variable i, which it labels as being static. Since the value of i is incremented on every recursive call to loop, no two iterations of the recursion ever have identical values for i so Similix unfolds and symbolically executes loop forever.

Partial evaluation of iota based on lazy use-analysis terminates. During the first symbolic execution of loop, no use annotation is created for (= i n) since n is unknown; and, no use annotation is created for evaluating i to place it in the car of a cons cell since this involves no computation. The expression (1+ i) causes the creation of a use link between the value returned by the function application and the value of i. The second iteration creates a similar annotation, but no actual use of i. At the point when symbolic execution reaches the second recursive call to loop, it is recognized that the two previous iterations of the loop have used identical information about the free variables (i.e., no information). In other words, the two iterations of loop are equivalent so a residual loop is produced. Eventually, partial evaluation terminates yielding a residual program that is identical to the input program.

If iota had been applied to a known value, (= i n) would create a use dependence between the value of i and the resultant value of applying = to its arguments. The value of the predicate of the conditional would be used in making a control flow decision causing the value of the = expression, and therefore the value of i, to be used. Since each iteration of loop has a different value of i, no two iterations are ever found to be equivalent, and the recursion continues to be unfolded until the execution of iota is completed. The residual program that is eventually produced is just straight line code that conses together the list to be returned by iota.

4.2 High Quality Residual Code

Optimal residual code for partial evaluation of the regular expression matcher in Figure 9 with respect to a known regular expression, a^*, and an unknown input string is a completely inlined decision tree with a single loop for kleene star matching as shown in Figure 10.[5,6] Mix cannot produce this code because it only continues symbolic evaluation of a recursion as long as the loop is self recursive and all of the static arguments either have identical values or are bound to proper substructures of previous values. Leaving aside the self recursive limitation, the difficulty is that as a regular expression is processed, the expression is on average shrinking; however, kleene star processing temporarily increases the size of the regular expression. Also, the regular expression is often stored in two pieces, one of which is shrinking, and the other is growing.

Fuse uses a slightly more aggressive unfolding strategy. Unfolding and symbolic execution of a recursion is always continued unless two iterations of a recursion are separated by a *dynamic conditional*, one whose predicate is not decidable during partial evaluation. When a recursion spans a dynamic conditional, symbolic execution is still continued as long as all the arguments are either identical or shrinking in size. Although Fuse fails to produce optimal residual code for reasons similar to Mix, we include a more detailed outline of the steps performed by Fuse because it illuminates the underlying problem. Symbolic execution of the regular expression matcher by Fuse would proceed as follows:

```
(match? (make-kleene-star (make-term 'a)) ?)
(match? [kleene-star [term 'a]] ?)
(match-pattern? [kleene-star [term 'a]]
                [null-pattern] ?)
(match-star? [kleene-star [term 'a]]
             [null-pattern] ?)
(if (match? [null-pattern] ?)
    #t
    (match-pattern? . . .))
(match? [null-pattern] ?)
(match-pattern? [null-pattern] [null-pattern] ?)
(match-null? [null-pattern] ?)
(null? ?)
```

Since (null? ?) can not be evaluated during partial evaluation, this expression would be left residual. As a result, the conditional in **match-star?** would be dynamic. Symbolic execution of the alternative of this dynamic conditional would yield a call of the form (match-pattern? [term 'a] [concat [kleene-star [term 'a]] [null-pattern]] ?). Since this is a recursive call to match-pattern? that spans a dynamic conditional and one argument grows in size, Fuse would suspend symbolic execution at this point and use generalization to produce a specialized version of the loop. The result is residual code that still includes the dispatcher in **match-pattern?** and many of the data-structure abstractions of the pattern matcher.

Partial evaluation of the regular expression matcher using lazy use-analysis is not inherently complicated, but it is easy to get lost in the details. This presentation will therefore proceed through a simulation of the partial evaluation process virtually function call by function call. The procedure to which a recursive call will be detected and which will

[5]The regular expression matcher presented does not include disjunctions or epsilon rules. These have been omitted for simplicity and do not effect the way in which any of the systems discussed would perform partial evaluation.

[6]Brackets are used in code segments to denote objects to which functions are to be applied. Question marks are used to represent unspecified values (i.e., $\perp_{PEval}$).

```
(define match?
  (lambda (pattern input)
    (match-pattern? pattern null-pattern input)))

(define match-pattern?
  (lambda (pattern rest-pattern input)
    (cond ((null-pattern? pattern)
           (match-null? rest-pattern input))
          ((term? pattern)
           (match-term? pattern rest-pattern input))
          ((kleene-star? pattern)
           (match-star? pattern rest-pattern input))
          ((concat? pattern)
           (match-concat? pattern rest-pattern input)))))

(define null-pattern?
  (lambda (pattern) (eq? pattern null-pattern)))

(define match-null?
  (lambda (rest-pattern input)
    (if (null-pattern? rest-pattern)
        (null? input)
        (match? rest-pattern input))))

(define match-term?
  (lambda (term-pattern rest-pattern input)
    (if (and (pair? input)
             (equal? (term-symbol term-pattern)
                     (car input)))
        (match? rest-pattern (cdr input))
        #f)))

(define match-star?
  (lambda (star-pattern rest-pattern input)
    (if (match? rest-pattern input)
        #t
        (match-pattern?
         (kleene-star-expr star-pattern)
         (concat star-pattern rest-pattern)
         input))))

(define concat
  (lambda (pattern1 pattern2)
    (cond ((null-pattern? pattern1) pattern2)
          ((null-pattern? pattern2) pattern1)
          (#t (make-concat pattern1 pattern2)))))

(define match-concat?
  (lambda (concat-pattern rest-pattern input)
    (match-pattern?
     (concat-head concat-pattern)
     (concat (concat-tail concat-pattern) rest-pattern)
     input)))
```

Figure 9: Regular expression matcher example

```
(define match?
  (lambda (pattern input)
    (if (null? input)
        #t
        (if (and (pair? input)
                 (equal? 'a (car input)))
            (match? [kleene-star [term 'a]]
                    (cdr input))
            #f))))
```

Figure 10: Optimal residual code for (match? (make-kleene-star (make-term 'a)) ?)

lead to termination is match?. This is the very first function that is called after the regular expression that is supplied as input has been generated.

(match? (make-kleene-star (make-term 'a)) ?) Initial call to begin partial evaluation.

(match? [kleene-star [term 'a]] ?) Regular expression description created.

(match-pattern? [kleene-star [term 'a]] [null-pattern] ?) A use dependence is created between the pattern variable in match-pattern? and the pattern variable in match?.

(null-pattern? [kleene-star [term 'a]]) A use dependence is created between pattern in null-pattern? and match-pattern?.

(eq? [kleene-star [term 'a]] [null-pattern]]) A use dependence is created between the #f returned by eq? and pattern in null-pattern?. The value of #f is used to make a control flow decision in the cond in match-pattern?. This use propagates along the use dependence links eventually asserting use of the identity of pattern in both match-pattern? and match?.

(term? [kleene-star [term 'a]]) A use dependence is created between the #f returned by term? and pattern in match-pattern?. When this #f is used by the cond the same use assertions created by the previous clause of cond are made.

(kleene-star? [kleene-star [term 'a]]) Same as the previous step.

(match-star? [kleene-star [term 'a]] [null-pattern] ?) A use dependence is created between the corresponding variables of match-star? and match-pattern?.

(match? [null-pattern] ?) This expression returns an unknown value. Since it is the predicate of a conditional, both arms of the conditional will be investigated by the partial evaluator. None of the use-analysis done for this expression is of interest, so the details have been omitted.

(kleene-star-expr [kleene-star [term 'a]]) A use dependence is created between the value [term 'a] returned by the function application and the corresponding subcomponent of star-pattern.

(concat [kleene-star [term 'a]] [null-pattern]) Use dependencies are created between the values pattern1 and pattern2 in concat and the values star-pattern and rest-pattern in match-star?. The values of both pattern1 and pattern2 are then used causing use assertions to be propagated back to pattern and rest-pattern in match-pattern?.

(match-pattern? [term 'a] [kleene-star [term 'a]]) Use dependencies are created between the formal parameters in match-pattern? and the sources of the actuals. null-pattern? is then handled similarly to the previous call to this function.

(term? [term 'a]) A use dependence is created between the #t that is returned and pattern. The #t is then used by the cond causing a use assertion to propagate back all the way to the [term 'a] subcomponent of the very first call to match?.

(match-term? [term 'a] [kleene-star [term 'a]] ?) Use dependencies are created between the corresponding arguments. The predicate in match-term? does not yield a value during partial evaluation so both arms of the conditional must be investigated. The use assertions made during the evaluation of the predicate are again uninteresting.

(match? [kleene-star [term 'a]] ?) This is the first recursive call to match?. At this point a use profile for the first iteration of the recursion is available. It shows that the value of pattern was used, but no information about input was required.

The second call to match? would proceed exactly like the first, yielding a second recursive call to this function. At that point, the use profiles of the two iterations would be compared. In both iterations, the value of pattern was used; however, since the values were identical, the two iterations are equivalent. As a result, a residual loop would be created for match. The residual code produced is precisely the optimal code for (match? [kleene-star [term 'a]] ?) shown in figure 10.

5 Future Research

We are in the process of building an RCP partial evaluator based on lazy use-analysis and an information lattice similar to that in Figure 6. The input language that the partial evaluator will accept is a subset of higher-order, pure (functional) Scheme including only the integer, boolean, pair, and closure data types. This language includes all of the fundamental complexities inherent in any higher-order, applicative order, functional language.

Our goals are to investigate the classes of programs on which the proposed mechanism terminates, how efficiently the mechanism can be implemented; and, depending on these results, in what ways the information lattice or information retention mechanisms might be modified to produce a better partial evaluator. We also hope to gain insight into the use of use information as a guide to partial evaluation. This may enable us to investigate TP use information based partial evaluators in the future.

6 Conclusions

A partial evaluator based on lazy use-analysis will terminate on programs including counters and will properly unfold Mogensen's (unpublished) regular expression accepter. No automatic partial evaluator to date has been able to do both. Use-analysis offers a potential solution to other open challenges in partial evaluation such as selecting which specializations to include in a residual program, specializing imperative programs, and performing driving. Selecting which specializations to produce in a residual program was the first application to which use-analysis was applied [5]. This work used an eager form of use-analysis and demonstrated the viability and effectiveness of use-analysis. We believe that lazy use-analysis will only improve the solution to this problem.

Partial evaluation of imperative programming languages is not conceptually complicated. The problem is how to determine when two iterations of a recursion are equivalent. Previous partial evaluators could not handle arbitrary side effects because the only known method for comparing two iterations was comparison of the complete stores at two points in the execution sequence. This is simply infeasible. In a use-analysis based partial evaluator, only those portions of the store that are used must be compared to decide equivalence. Since the portion of a store used by a program segment will typically be a small fraction of the entire store, use-analysis makes partial evaluation of imperative languages both computationally and conceptually feasible.

Whether use information can be utilized to perform driving [7] is an open question. Preliminary investigation indicates that maintaining use information not only about values, but also about the residual code produced, may enable a use based partial evaluator to perform the same optimizations achieved through driving. The viability of achieving driving through use-analysis should become clearer through further investigation of use information.

In conclusion, use-analysis is a promising new technology for solving many of the open problems in partial evaluation. We are in the process of building an RCP partial evaluator based on lazy use-analysis that will produce a better combination of residual code quality and effective termination than any existing system. We believe this system will demonstrate the efficacy of basing termination on use-analysis and act as a stepping stone to the solution of several other problems currently impeding the transition of partial evaluation from a laboratory experiment into a widely used tool.

References

[1] Anders Bondorf and Olivier Danvy. Automatic autoprojection for recursive equations with global variables and abstract data types. DIKU Report 90/04, University of Copenhagen, Copenhagen, Denmark, 1990.

[2] Charles Consel. New insights into partial evaluation: the SCHISM experiment. In *Proceedings of the 2nd European Symposium on Programming*, pages 236–246. Springer-Verlag, LNCS 300, 1988.

[3] Carsten Kehler Holst. Finiteness analysis. In *Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 473–495. Springer-Verlag Lecture Notes in Computer Science, August 1991.

[4] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 1(3/4):9–50, 1988.

[5] Erik Ruf and Daniel Weise. Using types to avoid redundant specialization. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, CN, June 1991.

[6] Peter Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.

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

[8] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online program specialization. In *Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 165–191. Springer-Verlag Lecture Notes in Computer Science, August 1991.

Fully Lazy Higher-Order Removal

Wei-Ngan CHIN
Dept of Information Systems & Computer Science
National University of Singapore

Abstract

Higher-order functions have become quite indispensable to functional programming. They increase the expressive power of the language and provide an additional level of software abstraction from which to write concise and reusable programs. However, higher-order functions are more expensive to execute and are also more difficult to analyse for optimisations.

In [CD92], we proposed a transformation method which could automatically remove *most* higher-order expressions from functional programs. However, the proposed method does not preserve full laziness and may actually result in (lower-order) programs which are less efficient. Previous full laziness techniques have depended on the higher-order facility and are therefore not compatible with the higher-order removal method. In this paper, we propose a new full laziness transformation method that does not rely on the higher-order facility. This new method extracts out *ground-type* maximal free expressions and can be formalised as a transformation algorithm. We present the algorithm, use examples to illustrate it, and provide an outline of its termination proof. Integration of this full laziness method into higher-order removal will also be presented. Lastly, we review the proposed full laziness method.

1 Introduction

Higher-order functional programs can contain functions as results and/or pass functions as arguments. This facility enables more concise and reusable programs to be constructed. However, higher-order facility is expensive to support (relies heavily on heap space for closures) and complicates program optimisation (more difficult to handle than first-order language).

Recently, an automatic transformation method, called higher-order removal [CD92], has been proposed which can remove *most* higher-order expressions from purely functional programs by transforming them to equivalent first-order or lower-order expressions. This method uses two techniques which can be formalised as fold/unfold transformation algorithms. The first technique, called *lump uncurrying*, is used to convert curried applications to equivalent uncurried function calls. This technique helps to eliminate function-type results. The second technique involves a procedure which can *specialise* higher-order function calls with instantiated function-type arguments. This technique helps to eliminate function-type arguments. Together, the two techniques can remove most higher-order expressions from functional programs.

However, a current problem of this method is that it is not *fully lazy*. In particular, during the elimination of *non-linear*[1] function-type arguments, our method may result in less efficient programs. For example, the following Hope program[2] contains a function call, *dup*, with higher-order argument, *lambda x ⇒ sqr(y)+x end.*

```
---p(y)    ⇐ dup(lambda x ⇒ sqr(y)+x end);
---dup(v) ⇐ v(3)+v(4);
```

The original higher-order removal method is able to eliminate the function-type argument by directly unfolding the *dup* call. However, this step produces the following less efficient first-order program:

```
---p(y)    ⇐ (sqr(y)+3)+(sqr(y)+4);
```

The main cause of the above loss of efficiency is that a *free expression*[3], *sqr(y)*, has been duplicated (during unfolding) via the non-linear function-type parameter of *dup*. This causes some codes to be duplicated.

The problem posed by this duplication of free expressions is closely related to the full laziness concern which was addressed by Wadsworth's fully lazy graph reduction technique [Wad77], and Hughes's fully lazy lambda lifting technique [Hug82]. Wadsworth's technique is a run-time technique, while Hughes's technique is a compile-time one. Both techniques try to identify *maximal free expressions* which ought to be shared, rather than duplicated. (A *maximal free expression* (MFE) is a free expression which is **not** contained within another free expression). For example, the lambda lifting technique, which converts all lambda abstractions to supercombinators, can be made fully lazy by the direct extraction of MFEs during lambda lifting itself. Lately, Peyton-Jones and Lister [PJL91] have shown that the full laziness technique can be divorced from lambda lifting. They use the *let* construct to capture MFEs which are then floated outwards, as much as possible. This helps to facilitate sharing of the MFEs.

However, all the previous methods of preserving full laziness (by extracting MFEs) have relied on the higher-order

[0] Author's Address: Dept of IS & CS, National University of Singapore, Kent Ridge, Singapore 0511, e-mail: chinwn@iscs.nus.sg

[1] A parameter of a function is *non-linear* if it occurs more than once in each evaluation branch of its function's RHS term.

[2] In the Hope language, equations are of the form ---*LHS* ⇐ *RHS;*

[3] A *free expression* (of a lambda abstraction) is an expression (other than a variable or a constant) which does not contain any bound variables (of the lambda abstraction).

facility. They are therefore not compatible with higher-order removal. This is because the extracted MFEs may be function-type. Their extractions through new intermediate functions (or via *let* constructs) actually cause the introduction of function-type arguments, rather than their removal. Apart from this incompatibility with the higher-order removal method, there are several other shortcomings present in the previous full laziness techniques. These shortcoming include:

- Maximum laziness is often dependent on the *positional order* of parameters. Different orderings of parameters can result in different degrees of laziness [PJ87].
- Extraction of partially applied expressions which are *unshared* can result in unnecessary intermediate functions [HG85].
- Space leak may occur when partially applied recursive functions are lambda lifted in a fully lazy way [PJ87].

In this paper, we propose a new transformation approach to full laziness which is able to overcome the above shortcomings. This new full laziness method is also compatible with the higher-order removal method. In Section 2, we describe four main techniques which can be used to preserve full laziness, without depending on (the introduction of) higher-order expressions. In Section 3, we formalise these full laziness preservation techniques as a transformation algorithm. This algorithm is terminating. An outline of the termination proof will be given. In Section 4, we show how the full laziness method is integrated with the higher-order removal method. Pros and cons of the new full laziness method, together with measures to minimise loss of laziness, are discussed in Section 5. Section 6 concludes.

2 New Techniques for Full Laziness

Instead of extracting out all MFEs, our new full laziness method will extract out *only* ground-type MFEs. A *ground-type MFE* (GT-MFE) is an MFE whose result type is that of a first-order object (e.g. num, char). In contrast, a *function-type MFE* is an MFE whose result type is that of a function (e.g. *num* → *num* → *num*).

We avoid the extraction of function-type MFEs. This is because they are incompatible with the higher-order removal method. Function-type MFEs result in function-type arguments. They introduce new higher-order expressions, rather than their removal.

To maximise the laziness of the transformed programs, we shall use four main extraction techniques, namely:

T1: Extraction of GT-MFEs from lambda abstractions

T2: Conversion of grounded GT-MFEs to constant functions.

T3: Direct unfolds of non-recursive function calls. This is to help reveal *implicit* GT-MFEs, where possible. *Implicit* MFEs are MFEs which are hidden underneath function calls (inside the calls' function bodies).

T4: Linearise the parameters of certain non-recursive function calls. This is to facilitate direct unfolds to help reveal further implicit GT-MFEs.

These four techniques are referred to as **T1**, **T2**, **T3** and **T4**, respectively. The full laziness transformation method consists of these four safe techniques to help float out both *explicit* and *implicit* GT-MFEs. During higher-order removal, these techniques can be repeatedly applied to each (non-linear) higher-order argument until no more GT-MFEs can be found. This is to ensure that the non-linear higher-order argument can be eliminated without loss of laziness.

In the next few sub-sections, we describe (with the help of examples) the above four techniques for extracting GT-MFEs.

2.1 T1: Extraction of GT-MFEs from lambda abstractions

We assume the use of an *uncurried* language where partial applications are written using lambda abstractions. These are also the places where free expressions may exist and can be extracted to preserve full laziness. The first technique **T1**, of our full laziness method, is to search and extract out all explicit GT-MFEs from lambda asbtractions.

As an example, consider the function *F*:

```
dec F : list(num) # num # num → list(num);
dec map : list(A) # (A → B) → list(B);
---F(xs,A,B)    ⇐ map(xs,lambda a ⇒ a+A*B end);
---map([],f)    ⇐ [];
---map(x::xs,f) ⇐ f(x)::map(xs,f);
```

(Note on Hope syntax: the dec statement is used to declare the type of functions, and *::* is the infix constructor for the *list* data type.)

Presently, this function is *not* fully lazy as there is a GT-MFE, *A*B*, in the lambda abstraction of the *map* call. To make this function fully lazy, this GT-MFE must be extracted with an intermediate function (or a *let* construct as used in [PJL91]). Below is an extraction of the GT-MFE using (*int_F*) as an intermediate function.

```
---F(xs,A,B) ⇐ map(xs,int_F(A*B));
---int_F(Z)  ⇐ lambda a ⇒ a+Z end;
```

We suggest that it is quite acceptable to consider only ground-type MFEs for extraction. This is true as long as there is a way of extracting out all such MFEs, may they be explicitly present or implicitly hidden under levels of function calls. The rationale behind this is that ground-type MFEs are the major source of redundancy when expressions are duplicated. In contrast, function-type MFEs require additional inputs before being complete for computation. We shall see in Section 5 that ignoring the function-type MFEs need only suffer from a minor loss of laziness. This loss of laziness can be tolerated because it is related to the space-leak problem.

2.2 T2: Conversion of grounded GT-MFEs to constant functions

In a given lambda abstraction, *grounded*[4] sub-expressions (with neither free nor bound variables) are also considered as free expressions. Instead of extracting them and risk the introduction of (unnecessary) intermediate functions, we can introduce constant functions (CF) in place of grounded GT-MFEs.

For example, the function *F2* below, contains a grounded GT-MFE *sqr(4)* in its lambda abstraction.

[4] An expression is *ground* if it does not contain free variables. Note that this is a different notion from ground-type which is for expressions which return first-order objects.

```
dec F2 : list(num) → list(num);
---F2(xs) ⇐ map(xs,lambda a ⇒ a+sqr(4) end);
```

It is possible to introduce a constant function *sq4*, to replace the grounded GT-MFE *sqr(4)*, as shown below.

```
---F2(xs) ⇐ map(xs,lambda a ⇒ a+sq4 end);
---sq4    ⇐ sqr(4);
```

This constant function need only be evaluated once with its result shared by all calls to it, making the above program fully lazy. This technique is adapted from [PJ87]. Instead of all CAFs (Constant Applicative Form), we convert only ground-type CAFs (or grounded GT-MFEs) to constant functions.

2.3 T3: Direct unfolds of non-recursive function calls.

Not all GT-MFEs are explicitly available for extraction in lambda abstractions. Some of them may be hidden underneath function calls (in the calls' function bodies). As an example, consider the function *F3*, as shown below.

```
---F3(xs,A,B) ⇐ map(xs,lambda a ⇒ G3(a,A,B) end);
---G3(x,y,z)  ⇐ x+y*z;
```

This function contains an implicit GT-MFE, *A*B*, which is hidden beneath the non-recursive *G3* call. One way of making this GT-MFE explicit is to directly unfold the non-recursive *G3* function call, to give:

```
---F3(xs,A,B) ⇐ map(xs,lambda a ⇒ a+A*B end);
```

Direct unfolding results in fewer intermediate function calls. It also helps to reveal hidden GT-MFEs for extraction by the first technique, **T1**. However, we may only apply direct unfolding to non-recursive function calls which do not have large non-linear arguments. (A large argument is a sub-expression that is not a variable or a constant.) This restriction is meant to avoid two problems. Firstly, recursive functions may cause non-termination if they are considered for direct unfolding. Secondly, function calls with large non-linear arguments can cause loss of efficiency (by duplicating code) when they are directly unfolded.

2.4 T4: Parameter Linearisation

Lastly, not all non-recursive function calls can be directly unfolded to reveal implicit GT-MFEs. Some of these calls are prevented from direct unfolding because it contains large non-linear arguments. An example is the following function, *F4*.

```
---F4(xs,A,B) ⇐ map(xs,lambda a ⇒ G4(a*a,A,B) end);
---G4(x,y,z)  ⇐ x*(y*z)-x;
```

The RHS of *F4* contains a *G4* call which hides an implicit GT-MFE, *A*B*. However, this call cannot be directly unfolded because there is a large non-linear argument, *a*a*, which can cause loss of efficiency when duplicated.

One way of revealing the hidden GT-MFE is to linearise the problematic parameter with an intermediate function, *G4a*, as shown below.

```
---F4(xs,A,B) ⇐ map(xs,lambda a⇒ G4(a*a,A,B) end);
---G4(x,y,z)  ⇐ G4a(x,y*z);
---G4a(x,yz)  ⇐ x*yz-x;
```

This linearisation enables the *G4* function to be directly unfolded. Unfolding this call reveals the hidden GT-MFE, *A*B*, as shown below.

```
---F4(xs,A,B) ⇐ map(xs,lambda a ⇒ G4a(a*a,A*B) end);
```

3 Full Laziness Tranformation Algorithm

The above collection of techniques can be combined together to form the full laziness method. This method can be used to convert a program to fully lazy form prior to lambda lifting. It can also be used to make the higher-order removal method fully lazy (see Section 4).

We can formalise the full laziness method as a transformational algorithm that is made up of three main sets of syntax-directed rules, named $\mathcal{L}$, $\mathcal{E}$ and $\mathcal{F}$. The first rule, $\mathcal{L}$, is used to strategically place the second rule $\mathcal{E}$ at each lambda abstraction. The second rule, $\mathcal{E}$, is used to perform direct unfolding and extract out all GT-MFEs from each lambda abstraction. It uses the third rule, $\mathcal{F}$, to find both explicit and implicit GT-MFEs. The $\mathcal{F}$ rule is also used to linearise the parameters of certain non-recursive functions which contain implicit GT-MFEs but cannot be unfolded. Before presenting the three sets of rules, we first introduce a simple higher-order language for which the full laziness algorithm will be based.

3.1 The Language

Consider a simple higher-order language with functions of the form:

$$---f(v_1,\ldots,v_n) \Leftarrow tf;$$

with its RHS term, *tf*, described by:

$$\begin{array}{ll} t ::= v \mid f \mid c(t_1,\ldots,t_n) & \mid t(t_1,\ldots,t_n) \\ & \mid \mathit{case}\ t\ \mathit{in}\ p_1 \Rightarrow t_1;..;p_n \Rightarrow t_n\ \mathit{end} \\ & \mid \mathit{lambda}\ (v_1,\ldots,v_n) \Rightarrow tf\ \mathit{end} \\ p ::= v \mid c(p_1,\ldots,p_j) & \end{array}$$

This simple language contains variables (v), constructor terms ($c(t_1,\ldots,t_n)$), applications ($t(t_1,\ldots,t_n)$), user-defined functions (f), case constructs and lambda abstractions. These constructs are basic expression structures found in almost all modern functional languages. We shall base our full laziness method on this language.

The above language is an uncurried language. However, it is still possible to have *curried applications*. Curried applications are all those applications, except *function calls*, $f(t_1,\ldots,t_n)$, and *variable applications*, $va(t_1,\ldots,t_n)$, where:

$$va ::= v \mid va(t_1,\ldots,t_n)$$

Curried applications are more difficult to handle. Fortunately, they can be eliminated by a technique, called lump uncurrying [CD92], which replaces each curried application by an equivalent uncurried function call. This technique has been formalised as a transformation algorithm, named $\mathcal{A}$, which could transform any well-typed expression of the full higher-order form to an equivalent expression without curried applications. The new expression form without curried application, called the *applyless form*, can be specified by the grammar:

$$\begin{array}{lll} t & ::= v \mid f \mid & c(t_1,\ldots,t_n) \mid va(t_1,\ldots,t_n) \mid f(t_1,\ldots,t_n) \\ & & \mid \mathit{case}\ t\ \mathit{in}\ p_1 \Rightarrow t_1;..;p_n \Rightarrow t_n\ \mathit{end} \\ & & \mid \mathit{lambda}\ (v_1,\ldots,v_n) \Rightarrow tf\ \mathit{end} \end{array}$$

WHERE f is an *applyless* function. An *applyless* function is a function whose RHS term is *applyless*.

In this paper, we shall base the full laziness algorithm on this simpler applyless expression form.

3.2 The $\mathcal{L}$ Rule

The main rule, $\mathcal{L}$, for the full laziness transformation must be applied to the RHS of each function. This must be applied in a *bottom-up* order, so that auxiliary (or child) functions become *fully lazy* before their parent functions. A function, f_1, is considered to be a *child* function of another function, f_2, if f_2 calls f_1 but not vice-versa. If f_1 also calls f_2, then we have *sibling* or *mutually recursive* functions.

The $\mathcal{L}$ rule, given in Figure 1, basically places an $\mathcal{E}$ rule application for each lambda abstraction. This is intended to lead to the extraction of GT-MFEs via the techniques of T1, T2, T3 and T4. However, each user-defined (global) function:

```
        ---f(v1,...,vn) ⇐ tf;
```

is also a lambda abstraction - the outermost one. This outermost lambda abstraction does not contain free variables but it may still be possible to find grounded GT-MFEs which must be converted to constant functions. To extract these, we place an $\mathcal{E}$ invocation (before the $\mathcal{L}$ call) on the RHS of each function definition, as follows:

```
        ---f(v1,...,vn)) ⇐ ℰ[ℒ[tf]] {v1,...,vn}
```

To illustrate the use of the $\mathcal{L}$ rules, let us consider the following simple program.

```
---P(x)    ⇐ dup(lambda a ⇒ h1(x,a) end);
---h1(x,a) ⇐ h2(x*x,sqr(a));
---h2(x,a) ⇐ x*x+(a+a);
---dup(v)  ⇐ v(3)+v(4);
```

The only function with a lambda abstraction is the function P. To make this program fully lazy, we apply the $\mathcal{E}$ and $\mathcal{L}$ rules to its RHS term, as shown below.

```
---P(x)     ⇐ ℰ[ℒ[dup(lambda a ⇒  h1(x,a) end)]] {x};
            ⇐ ℰ[dup(ℒ[lambda a ⇒  h1(x,a) end])] {x};
            ⇐ ℰ[dup(ℰ[lambda a ⇒  ℒ[h1(x,a)] end] {})] {x};
            ⇐ ℰ[dup(ℰ[lambda a ⇒  h1(x,a) end] {})] {x};
```

The above steps manage to place two $\mathcal{E}$ calls. These calls are capable of extracting out all the GT-MFEs (both explicit and implicit) in P. The outer $\mathcal{E}$ call, with x as its bound variable, will be used to convert all grounded GT-MFEs to constant functions. The inner $\mathcal{E}$ call will be used to extract out all GT-MFEs from the inner lambda abstraction. In the next sub-section, we shall look at how the GT-MFEs of this example are extracted through the $\mathcal{E}$ rule.

3.3 The $\mathcal{E}$ Rule

Both explicit and implicit GT-MFEs are extracted by the $\mathcal{E}$ rules of Figure 2. These rules repeatedly extract out GT-MFEs which are free relative to a set of bound variables, *bv*. An expression is *free* relative to a set of bound variables, if none of the bound variables are contained in the given expression.

At the start, each expression is subjected to an auxiliary rule, called $\mathcal{U}$. This rule performs direct unfolding on each non-recursive function call, which will not cause loss of efficiency when unfolded. Note the use of the $\otimes$ annotation (in $\mathcal{U}7$) to identify non-linear parameters. This rule avoids unfolding function calls with large non-linear arguments. It also avoids unfolding function calls whose bodies are lambda abstractions. This is because such function calls will not contain any GT-MFEs after a bottom-up order of transforming functions. The non-recursive function calls are allowed to contain instantiated function-type arguments when unfolded. This may cause curried applications to be formed from variable applications. To remove them, we simply apply the $\mathcal{A}$ rule to each of the unfolded calls.

The actual extraction is done with the help of another rule, named $\mathcal{F}$. This rule is repeatedly applied until no more GT-MFEs are left in the given expression. This repetitive application is invoked by $\mathcal{E}1a$ and $\mathcal{E}1b$. It terminates with $\mathcal{E}2$ when no more new GT-MFEs are found. For each GT-MFE extracted, the rule uses an auxiliary meta-function, $\mathcal{CF}$, to check if the GT-MFE is either:

1. a grounded GT-MFE, and/or
2. a curried application.

If it is a grounded GT-MFE, the $\mathcal{CF}$ rule will convert it into a constant function. If it is a curried application, the $\mathcal{A}$ rule will be used to transform it to uncurried form. There is a possibility of getting curried applications because we are performing this full laziness transformation before the elimination of function-type arguments. These function-type arguments may be used to instantiate variable applications to curried applications when rule $\mathcal{F}7b2.2.1$ (see Section 3.4) searches for implicit GT-MFEs.

The earlier example from the previous section contains two un-evaluated $\mathcal{E}$ rule applications. We are now ready to show the main steps of extracting GT-MFEs by these two $\mathcal{E}$ applications. At each invocation of the $\mathcal{E}$ rules, the $\mathcal{F}$ rule is called to search for both explicit and implicit GT-MFEs. We show the results of these $\mathcal{F}$ rule applications as brief comments preceded by the symbol '!'. (More detailed steps of the $\mathcal{F}$ rule will be presented in the next section.) Each of the $\mathcal{F}$ invocations returns a tuple of three items, namely the new expression after extraction, a list of extracted GT-MFEs and a corresponding list of new variables to replace the extracted GT-MFEs. The presence or absence of GT-MFEs is used to determine what action must be carried out by the $\mathcal{E}$ rule, as shown below.

```
 ---P(x)     ⇐ ℰ[dup(ℰ[lambda a ⇒  h1(x,a) end] {})] {x};
             ! ℱ[lambda a ⇒  h2(x * x, sqr(a)) end] {} gives
             !     (lambda a ⇒ h2(A,sqr(a)) end,[x*x],[A])
             ! ℰ1a:extract and define fn1
             ⇐ ℰ[dup(fn1(x * x))] {x};
             ! ℱ[dup(fn1(x * x))] {} gives
             !     (dup(fn1(x*x)),[],[])
             ! ℰ2:terminate
             ⇐ dup(fn1(x*x));
Define fn1 by (ℰ)
 ---fn1(A)   ⇐ ℰ[lambda a ⇒  h2(A,sqr(a)) end] {};
             ! ℱ[lambda a ⇒  h2(A,sqr(a)) end] {}
             ! (lambda a ⇒ h2'(B,sqr(a)) end,[A*A],[B]) gives
             !     ℰ1a:extract and define fn2
             ⇐ fn1(A*A);
Define fn2 (by ℰ)
 ---fn2(B)   ⇐ ℰ[lambda a ⇒  h2'(B,sqr(a)) end] {}
             ! ℱ[lambda a ⇒  h2'(B,sqr(a)) end] {} gives
             !     (lambda a ⇒ h2'(B,sqr(a)) end,[],[])
             ! ℰ2:terminates
             ⇐ lambda a ⇒ h2'(B,sqr(a)) end;
Define h2' (by ℱ)
 ---h2'(B,a) ⇐ B+(a+a);
```

The result of the above extractions is the following fully lazy program for P.

```
---P(x)     ⇐ dup(fn1(x*x));
---fn1(A)   ⇐ fn2(A*A);
---fn2(B)   ⇐ lambda a ⇒ h2'(B,sqr(a)) end ;
---h2'(B,a) ⇐ B+(a+a);
```

```
(1) $\mathcal{L}[v]$                                              $\Leftarrow v$
(2) $\mathcal{L}[f]$                                              $\Leftarrow f$
(3) $\mathcal{L}[c(t_1,\ldots,t_n)]$                              $\Leftarrow c(\mathcal{L}[t_1],\ldots,\mathcal{L}[t_n])$
(4) $\mathcal{L}[va(t_1,\ldots,t_n)]$                             $\Leftarrow \mathcal{L}[va](\mathcal{L}[t_1],\ldots,\mathcal{L}[t_n])$
(5) $\mathcal{L}[case\ t\ in\ p_1 \Rightarrow t_1;..;p_n \Rightarrow t_n\ end]$   $\Leftarrow case\ \mathcal{L}[t]\ in\ p_1 \Rightarrow \mathcal{L}[t_1];..;p_n \Rightarrow \mathcal{L}[t_n]\ end$
(6) $\mathcal{L}[lambda\ (v_1,\ldots,v_n) \Rightarrow tf\ end]$        $\Leftarrow \mathcal{E}[lambda\ (v_1,\ldots,v_n) \Rightarrow \mathcal{L}[tf]\ end]$
(7) $\mathcal{L}[f(t_1,\ldots,t_n)]$                              $\Leftarrow f(\mathcal{L}[t_1],\ldots,\mathcal{L}[t_n])$
```

Figure 1: Rule $\mathcal{L}$

```
$\mathcal{E}[expr]\ bv$
    LET $(expr\bullet, fe_1 \cdots fe_p, ve_1 \cdots ve_p) = \mathcal{F}[\mathcal{U}[expr]]\ bv$ IN
    1) IF $p > 0$ THEN
       LET $(expr\bullet', fe'_1 \cdots fe'_q, ve'_1 \cdots ve'_q) = \mathcal{CF}[expr\bullet]$ IN
       a) IF $q > 0$ THEN         $\Leftarrow f\_new(fe'_1,\ldots,fe'_q, vr_1,\ldots,vr_z)$
                                     WHERE $vr_1 \cdots vr_z$=free_vars$[expr\bullet'] - \{ve_1 \cdots ve_q\}$
                                     DEFINE---$f\_new(ve'_1,\ldots,ve'_q, vr_1,\ldots,vr_z) \Leftarrow \mathcal{E}[expr\bullet']\ bv$
       b) IF $q = 0$              $\Leftarrow \mathcal{E}[expr\bullet']\ bv$
    2) IF $p = 0$ THEN            $\Leftarrow expr$
WHERE
1)  $\mathcal{CF}[(e,[],[])]$                       $\Leftarrow (e,[],[])$
2)  $\mathcal{CF}[(e, fe :: fe\_lt, ve :: ve\_lt)]$
          a) IF $fe$ IS *grounded* THEN   $\Leftarrow \mathcal{CF}[(e[fc/ve], fe\_lt, ve\_lt)]$
                                          DEFINE ---$fc \Leftarrow \mathcal{A}[fe]$ ; constant function
          b) OTHERWISE                    $\Leftarrow (e', \mathcal{A}[fe] :: fe\_lt', ve :: ve\_lt')$
                                          WHERE $(e', fe\_lt', ve\_lt') = \mathcal{CF}[(e, fe\_lt, ve\_lt)]$
1)  $\mathcal{U}[v]$                                 $\Leftarrow v$
2)  $\mathcal{U}[f]$                                 $\Leftarrow f$
3)  $\mathcal{U}[c(t_1,\ldots,t_n)]$                 $\Leftarrow c(\mathcal{U}[t_1],\ldots,\mathcal{U}[t_n])$
4)  $\mathcal{U}[va(t_1,\ldots,t_n)]$                $\Leftarrow \mathcal{U}[va](\mathcal{U}[t_1],\ldots,\mathcal{U}[t_n])$
5)  $\mathcal{U}[case\ t\ in\ p_1 \Rightarrow t_1;..;p_n \Rightarrow t_n\ end]$   $\Leftarrow case\ \mathcal{U}[t]\ in\ p_1 \Rightarrow \mathcal{U}[t_1];..;p_n \Rightarrow \mathcal{U}[t_n]\ end$
6)  $\mathcal{U}[lambda\ (v_1,\ldots,v_n) \Rightarrow tf\ end]$   $\Leftarrow lambda\ (v_1,\ldots,v_n) \Rightarrow \mathcal{U}[tf]\ end$
7)  $\mathcal{U}[f(t_1,\ldots,t_n)]$
    IF $f$ is not recursive and $\forall a \in 1 \ldots n, t_a^{\otimes}$ IS A VARIABLE or CONSTANT (*direct unfold*)
                                          $\Leftarrow \mathcal{U}[\mathcal{A}[tf[t_1/v_1,\ldots,t_n/v_n]]]$
    OTHERWISE                             $\Leftarrow f(\mathcal{U}[t_1],\ldots,\mathcal{U}[t_n])$
```

Figure 2: Rule $\mathcal{E}$ and its Auxiliaries, $\mathcal{U}$ & $\mathcal{CF}$

Two intermediate functions, *fn1* and *fn2*, were introduced by the $\mathcal{E}$ rule to hold the extracted GT-MFEs. Another new function, *h2'*, was introduced by the $\mathcal{F}$ rule to help extract out an implicit GT-MFE from a function call which could not be directly unfolded, namely *h2(A,sqr(a))*. Each of $\mathcal{E}$ calls terminates whenever no more GT-MFEs are found by the $\mathcal{F}$ rule.

3.4 The $\mathcal{F}$ Rule

The search for GT-MFEs is carried out by the $\mathcal{F}$ rule of Figure 3. This rule takes an expression, *e*, together with a set of bound variables, *bv*, in order to return a tuple of three items, namely:

1. a new expression, $e\bullet$, obtained from the original expression, e, by extracting some or all of its GT-MFEs. (Note the use of notation, $e\bullet$, to denote an expression which has just been processed by one application of the $\mathcal{F}$ rule.)
2. a list of ground-type MFEs, $fe_1 \cdots fe_p$, extracted from *e* which are free relative to the set of bound variables, *bv*.
3. a corresponding list of new variables, $ve_1 \cdots ve_p$, to replace the extracted GT-MFEs, $fe_1 \cdots fe_p$.

Every expression examined by $\mathcal{F}$ is initially tested to see if it is a ground-type MFE by $\mathcal{F}0$. If it is not, then one of the remaining cases of the $\mathcal{F}$ rule is invoked instead.

Of particular interest is rule $\mathcal{F}7$ which deals with function calls. Initially, the arguments of these calls are tested to see if there are GT-MFEs among them. If there are, then these GT-MFEs are extracted through $\mathcal{F}7a$. However, if there are no GT-MFEs in the arguments and if the function call being dealt with is neither recursive nor has a lambda abstraction for its body, then we can search for implicit GT-MFEs. This search for implicit GT-MFEs begins in rule $\mathcal{F}7b.2$. Initially, a procedure called divide_arg is used to classify the arguments of the function call as either *free* or *bound*, depending on whether it contains bound variables from *bv*. If none of the arguments are free, then there are no implicit GT-MFEs (see $\mathcal{F}7.2.1$). If free arguments exist, then we recursively call $\mathcal{F}$ to search for implicit GT-MFEs in the body of the function call (see $\mathcal{F}7b.2.2$). If implicit GT-MFEs are found, we use the technique of **T4** to define a new intermediate function in the function body, followed by a direct unfold for the call (see $\mathcal{F}7b.2.2.2$). There is also another auxiliary rule, $\mathcal{F}\_list$. This rule is a more general form of $\mathcal{F}$. It deals with the extraction from a list of expressions rather than from a single expression.

Notice that the $\mathcal{F}$ rule will locate and extract out all explicit GT-MFEs before the implicit ones. Each invocation of the $\mathcal{F}$ rule often find some of the GT-MFEs. This means that we may have to apply the $\mathcal{F}$ rule a number of times before all the GT-MFEs are found. This repetitive location and extraction of GT-MFEs is handled outside the $\mathcal{F}$ rule (e.g. by the $\mathcal{E}$ rule).

We are now really to illustrate the location and extraction of GT-MFEs by $\mathcal{F}$, using the earlier example. Altogether, there were four invocations to the $\mathcal{F}$ rule but only two of them produced any GT-MFEs. These two invocations are illustrated in Figure 4.

The first invocation of the $\mathcal{F}$ rule results in the extraction of an explicit GT-MFE, x*x. The second invocation is slightly more complicated because it involves the application of technique **T4** by step $\mathcal{F}7.2.2.1$. This invocation manages to extract an implicit GT-MFE, A*A, and introduces an intermediate function, *h2'*.

3.5 Outline of Termination Proof

There are a number of rules used by our full laziness transformation algorithm. The main rules are $\mathcal{L}$, $\mathcal{E}$ and $\mathcal{F}$. These rules fall under the following invocation or calling hierarchy, namely $\mathcal{L} > \mathcal{E} > \mathcal{F}$. Apart from these rules, there are also a number of auxiliary rules, e.g. $\mathcal{U}$, $\mathcal{CF}$, $\mathcal{F}\_list$ and divide_arg, but these auxiliary rules can be trivially shown to be terminating. We shall therefore be concerned with just the main rules in this proof outline.

To prove that the entire full laziness transformation algorithm terminates, we have to prove that each of the above recursive rules terminates. Starting, with the $\mathcal{F}$ rule (which lies at the bottom of the calling hierarchy), we can see that each of its recursive rules either (1) operates on successively smaller sub-expressions or (2) operates on the bodies (RHS term) of non-recursive functions (in order to locate hidden GT-MFEs). These rules can have a well-founded decreasing measure that is made up of the function calling hierarchy number[5] combined with the size of expression.

The $\mathcal{L}$ rule operates on successively smaller expressions. It will therefore terminate if the $\mathcal{E}$ rule terminates. The $\mathcal{E}$ rule is also recursive but has a termination property that is tied closely to the $\mathcal{F}$ rule. In particular, it repeatedly extracts GT-MFEs (via $\mathcal{F}$) from its expression and will terminate when no more GT-MFEs are found. To show that this repeated extraction terminates, we must show that each application of $\mathcal{F}$ extracts out some GT-MFEs, in such a way that fewer GT-MFEs are left behind. To prove this, we simply define a GT-MFE measure to count the maximum number of extractable GT-MFEs (by $\mathcal{F}$) that is present in any given expression. The equational definition for this measure will essentially mirror the recursion structure of the $\mathcal{F}$ rule. It will always return a finite value because such a recursion structure (for $\mathcal{F}$) has been shown to be terminating. Having defined this GT-MFE measure, we could prove (by induction) that each expression which results from an application of $\mathcal{F}$ will either have a measure that is less than the measure of the original expression, or is zero (i.e. no more GT-MFEs). This well-founded decreasing measure is sufficient to prove that the $\mathcal{E}$ rule terminates.

4 Integrating the Full Laziness Method into Higher-Order Removal

Our full laziness method can be used independently to make programs *fully lazy*. It can also be combined with the higher-order removal method to preserve full laziness while removing higher-order expressions. To obtain a fully lazy higher-order removal method, we must first subject our programs to the full laziness transformation. This will help extract out all GT-MFEs from each lambda abstraction. This fully lazy form is more amenable to the extraction of GT-MFEs from non-linear function-type arguments which is our next step. (Linear function-type arguments do not cause loss of

[5] A number related to the calling hierarchy such that those functions on top of calling hierarchy always have *larger* numbers than those below them.

```
(0) $\mathcal{F}[expr]\ bv$
    IF $expr$ is a GT-MFE without any of the variables in $bv$ THEN
                                          $\Leftarrow (ve, [e], [ve])$
(1) $\mathcal{F}[v]\ bv$                  $\Leftarrow (v, [], [])$
(2) $\mathcal{F}[f]\ bv$                  $\Leftarrow (f, [], [])$
(3) $\mathcal{F}[c(t_1, \ldots, t_n)]\ bv$    $\Leftarrow (c(t\bullet_1, \ldots, t\bullet_n), fe\_lt, ve\_lt)$
                                          WHERE $(t\bullet_1 \cdots t\bullet_n, fe\_lt, ve\_lt) = \mathcal{F}\_list[t_1 \cdots t_n]\ bv$
(4) $\mathcal{F}[va(t_1, \ldots, t_n)]\ bv$   $\Leftarrow (va\bullet(t\bullet_1 \cdots t\bullet_n), fe\_lt <> fe\_lt', ve\_lt <> ve\_lt')$
                                          WHERE $(t\bullet_1 \cdots t\bullet_n, fe\_lt, ve\_lt) = \mathcal{F}\_list[t_1 \cdots t_n]\ bv$
                                                $(va\bullet, fe\_lt', ve\_lt') = \mathcal{F}[va]\ bv$
(5) $\mathcal{F}[case\ t\ in\ p_1 \Rightarrow\ t_1; ..; p_n \Rightarrow\ t_n\ end]\ bv$   $\Leftarrow (case\ t\bullet\ in\ p_1 \Rightarrow\ t\bullet_1; ..; p_n \Rightarrow\ t\bullet_n\ end$
                                                $, fe\_lt <> fe\_lt_1 <> \cdots <> fe\_lt_n$
                                                $, ve\_lt <> ve\_lt_1 <> \cdots <> ve\_lt_n)$
                                          WHERE $(t\bullet, fe\_lt, ve\_lt) \quad = \mathcal{F}[t]\ bv$
                                                $(t\bullet_1, fe\_lt_1, ve\_lt_1) = \mathcal{F}[t_1]\ bv \cup$ free_vars$[p_1]$
                                                   $\vdots$                  $\vdots$
                                                $(t\bullet_n, fe\_lt_n, ve\_lt_n) = \mathcal{F}[t_n]\ bv \cup$ free_vars$[p_n]$
(6) $\mathcal{F}[lambda\ (v_1, \ldots, v_n) \Rightarrow\ tf\ end]\ bv$   $\Leftarrow (lambda(v_1, \ldots, v_n) \Rightarrow\ tf\bullet end, fe\_lt, ve\_lt)$
                                          WHERE $(tf\bullet, fe\_lt, ve\_lt) = \mathcal{F}[va]\ bv \cup \{v_1, \ldots, v_n\}$
(7) $\mathcal{F}[f(t_1, \ldots, t_n)]\ bv$
    LET $(t\bullet_1 \cdots t\bullet_n, fe_1 \cdots fe_p, ve_1 \cdots ve_p) = \mathcal{F}\_list[t_1 \cdots t_n]\ bv$ IN
    a) IF $p \geq 1$ THEN                 $\Leftarrow (f(t\bullet_1, \ldots, t\bullet_n), fe_1 \cdots fe_p, ve_1 \cdots ve_p)$
    b) IF $p = 0$ THEN (Given $--- f(v_1, \ldots, v_n) \Leftarrow tf$)
       b.1) IF $f$ is recursive, primitive or $tf$ is a lambda abstraction THEN
                                          $\Leftarrow (f(t_1, \ldots, t_n), [], [])$
       b.2) ELSE LET $(freed, bounded)$=divide_arg$[t_1 \cdots t_n]\ bv$ IN
            b.2.1) IF $freed = \{\}$ THEN     $\Leftarrow (f(t_1, \ldots, t_n), [], [])$
            b.2.2) IF $freed \neq \{\}$ THEN
                   LET $(tf\bullet, fe'_1 \cdots fe'_q, ve'_1 \cdots ve'_q) = \mathcal{F}[tf]\ \{v_i | i \in 1 \ldots n, t_i \in bounded\}$ IN
                   b.2.2.1) IF $q > 0$    $\Leftarrow (f\_n(vr_1, \ldots, vr_z)[t_1/v_1, \ldots, t_n/v_n]$
                                            $, fe'_1 \cdots fe'_q[t_1/v_1, \ldots, t_n/v_n], ve'_1 \cdots ve'_q)$
                                          WHERE $vr_1 \cdots vr_z =$ free_vars$[tf\bullet]$
                                          DEFINE $--- f\_n(vr_1, \ldots, vr_z) \Leftarrow tf\bullet$
                   b.2.2.2) IF $q = 0$    $\Leftarrow (f(t_1, \ldots, t_n), [], [])$

WHERE
    $\mathcal{F}\_list[[]]\ bv$           $\Leftarrow ([], [], [])$
    $\mathcal{F}\_list[t :: ts]\ bv$      $\Leftarrow (t\bullet :: ts\bullet, fe\_t <> fe\_ts, ve\_t <> ve\_ts)$
                                          WHERE $(t\bullet, fe\_t, ve\_t) = \mathcal{F}[t]\ bv$
                                                $(ts\bullet, fe\_ts, ve\_ts) = \mathcal{F}\_list[ts]\ bv$

    divide_arg$[[]]\ bv$                  $\Leftarrow ([], [])$
    divide_arg$[t_i :: ts]\ bv$           $\Leftarrow$ IF $t_i$ does not contain any variables of $bv$
                                          THEN $(t_i :: freed, bounded)$
                                          ELSE $(freed, t_i :: bounded)$
                                          WHERE $(freed, bounded)$=divide_arg$[ts]\ bv$
```

Figure 3: Rule $\mathcal{F}$ and its Auxiliaries $\mathcal{F}\_list$ & divide_arg

```
$\mathcal{F}[lambda\ a \Rightarrow\ h2(x * x, sqr(a))\ end]\ \{\}$                                        ! $\mathcal{F}6$
      $= (lambda\ a \Rightarrow\ tf\ end, fel, vel)$ where $(tf, fel, vel) = \mathcal{F}[h2(x * x, sqr(a))]\ \{a\}$   ! $\mathcal{F}7a$
      $= (lambda\ a \Rightarrow\ tf\ end, fel, vel)$ where $(tf, fel, vel) = (h2(A, sqr(a)), [x * x], [A])$
      $= (lambda\ a \Rightarrow\ h2(A, sqr(a))\ end, [x * x], [A])$
$\mathcal{F}[lambda\ a \rightarrow\ h2(A, sqr(a))\ end]\ \{\}$                                            ! $\mathcal{F}6$
      $= (lambda\ a \Rightarrow\ tf\ end, fel, vel)$ where $(tf, fel, vel) = \mathcal{F}[h2(A, sqr(a))]\ \{a\}$     ! $\mathcal{F}7b.2.2.1$define $h2'$
      $= (lambda\ a \Rightarrow\ tf\ end, fel, vel)$ where $(tf, fel, vel) = (h2'(B, sqr(a)), [A * A], [B])$
      $= (lambda\ a \Rightarrow\ h2'(B, sqr(a))end, [A * A], [B])$
DEFINE $h2'$
$--- h2'(B,a) \Leftarrow B+(a+a);$
```

Figure 4: Two Examples of $\mathcal{F}$ Rule Applications

laziness when duplicated. Hence, there is no need to extract their GT-MFEs.)

During the elimination of instantiated function-type arguments, we must ensure that all GT-MFEs are extracted from non-linear function-type arguments before these are eliminated by unfold/fold transformations. The algorithm to eliminate higher-order arguments, called $\mathcal{R}$, consists of seven syntax-directed rules (see [CD92, Chi90]). The main elimination rule, $\mathcal{R}6$, deals with function calls which may contain function-type arguments. This rule has the form:

```
(6) R[f(t1,...,tn)] ⇐ ...lower order expr...
```

To make this elimination rule fully lazy, we simply modify $\mathcal{R}6$ to be the following.

```
(6) R[f(t1⊗,...,tm⊗,tm+1,...,tn)]
    LET (t•1,...,t•m,fe1,...,fep,ve1,...,vep)
                    =F_list[U[t1],...,U[tm]] {} IN
    IF p > 0 THEN  ⇐ f_n(vr1,...,vrz,R[fe1],...,R[fep])
        WHERE vr1 ··· vrz =
            free_var[t•1,...,t•m,tm+1,...,tn]−{ve1 ··· vep}
        DEFINE ---f_n(vr1,...,vrz,ve1,...,vep) ⇐
                    R[f(t•1,...,t•m,tm+1,...,tn)];
    ELSE            ⇐ ...lower order expr...
```

Notice that the non-linear function-type parameters have been separated out as $t_1^{\otimes},\ldots,t_m^{\otimes}$. The above modified rule uses $\mathcal{U}$ to perform direct unfolds, where possible, on the non-linear function-type arguments. It also uses $\mathcal{F}$*_list* to search and extract out the GT-MFEs from $t_1^{\otimes},\ldots,t_m^{\otimes}$. Each successful extraction uses an intermediate function (*f_n*) to hold the extracted GT-MFEs. This step is repeated by $\mathcal{R}6$ until no more GT-MFEs are found. When this happens, the non-linear function-type arguments can be safely duplicated (without loss of laziness) by the elimination step of $\mathcal{R}6$.

As an example of this fully lazy higher-order removal, consider the following set of three (applyless) functions.

```
dec main : list(num) # num # num → list(num);
---main(xs,A,B) ⇐ map(xs,lambda y ⇒ f_1(y,A*B) end);
---f_1(y,A)     ⇐ f_2(y*y,A+A);
---f_2(y,A)     ⇐ y*y+sqr(A);
```

The above program can be converted by $\mathcal{L}$ to the following fully lazy form:

```
---main(xs,A,B) ⇐ map(xs,ll_x(A*B);
---ll_x(A)      ⇐ ll_x2(A+A);
---ll_x2(A)     ⇐ ll_x3(sqr(A));
---ll_x3(A)     ⇐ lambda y ⇒ f_2a(y*y,A) end;
---f_2a(y,A)    ⇐ y*y+A;
```

In making the above program fully lazy, three intermediate functions, *ll_x*, *ll_x2* and *ll_x3*, were introduced to lift out the GT-MFEs of the argument, *lambda y ⇒ f_1(y,A*B) end*. This program also used a new function, *f_2a*, which was introduced to help recover an implicit GT-MFE. Higher-order expressions are still present in the above program but these were *not* introduced by the full laziness method. To remove them, we can use our new fully lazy higher-order removal method to obtain the following first-order program.

```
---main(xs,A,B)     ⇐ main_1(xs,A*B);
---main_1(xs,A)     ⇐ main_2(xs,sqr(A+A)) ;
---main_2([],A)     ⇐ [];
---main_2(x::xs,A)  ⇐ f_2a(x*x,A)::main_2(xs,A) ;
---f_2a(y,A)        ⇐ y*y+A;
```

Notice the introduction of two new intermediate functions, *main_1* and *main_2*. These are used to lift out GT-MFEs, prior to the elimination of the higher-order *map* call. Also, the previous intermediate functions, *ll_x*, *ll_x2* and *ll_x3*, have now been eliminated by direct unfolding.

5 Pros and Cons

In this section, we shall discuss the advantages and disadvantages of the new full laziness method when compared to previous methods of preserving full laziness. The original motivation for this new method is to design techniques of preserving full laziness which are compatible with higher-order removal. This objective constrained us to techniques which could extract out only ground-type MFEs. One immediate advantage of this constrain is that we avoid expensive higher-order facility. This is likely to result in more efficient (lower-order) target programs.

One of the basic techniques used is to perform direct unfolds, where possible. This technique helps to avoid unnecessary intermediate functions. It can be used in place of (or in conjunction) with the *sharing analysis* of [HG85] to avoid unnecessary intermediate functions from unshared function-type arguments.

Another advantage of the proposed full laziness method is that it is not dependent on suitable ordering of parameters for maximal laziness. Past full laziness techniques have relied on the extraction of (explicit) MFEs to preserve laziness. The laziness of these techniques are often affected by the way the parameters are ordered. However, our method extracts out both explicit and implicit GT-MFEs and is not sensitive to parameter ordering.

As an example, consider the program:

```
---p(y)    ⇐ dup(lambda x ⇒ g(x,y) end);
---g(x,y)  ⇐ sqr(y)+x;
---dup(v)  ⇐ v(3)+v(4);
```

The fully lazy lambda lifting technique of Hughes is not able to maximise laziness of this program unless the parameters of *g* are swapped around. However, our method is able to convert the above program to the following fully lazy form:

```
---p(y)    ⇐ dup(aux(sqr(y)));
---aux(y)  ⇐ lambda x ⇒ y+x end;
---dup(v)  ⇐ v(3)+v(4);
```

A potentially serious disadvantage of our method is that some loss of laziness may be possible! This is due to the fact that none of the function-type MFEs are extracted and shared. Let us informally examine how this loss of laziness can occur.

In general, function-type MFEs must wait for additional arguments before they form complete expressions for computation. As a result, duplicating FT-MFEs will, at most, cause the constructions of graph (or code unrolling) to be repeated at run-time. In the case of non-recursive functions, we can use direct unfolds to unroll codes at compile-time. This helps to minimise those loss of laziness associated with graph construction. However, recursive functions cannot be directly unfolded because we may run into non-termination problem. As a result, we may sometimes be prevented from maximising the laziness of recursive functions.

However, this may be a blessing in disguise because recursive functions can cause *space leak* problems when they are made fully lazy. This is because partially applied recursive function calls may be completely unrolled (unfolded) at run-time into a large graph/code, which awaits additional arguments. Such unrolled codes may help to reduce computation but they also require large storage spaces. This can happen unexpectedly - hence the space leak problem. We illustrate this phenomenon with the following example from [PeytonJ87]. (We assume the use of a curried language.)

```
---q            ⇐ drop 1000;
---drop n xs  ⇐ if n=0 then xs else drop (n-1) (tl xs);
```

The above function, *q*, contains a partially applied function call, *drop 1000*, which could be shared. This program can be converted by the fully lazy lambda lifting technique to:

```
---q                ⇐ drop 1000;
---drop n           ⇐ L (n=0) (drop (n-1)) ;
---L bool nxt xs ⇐ if bool then xs else nxt (tl xs);
```

With this fully lazy program, the function call *drop 1000* (of *q*) can be fully unrolled by one of its invocations (at run-time), in the following manner:

```
q ⇒ drop 1000
   ⇒ L false drop 999
   ⇒ L false L false drop 998
          ⋮
   ⇒ L false L false L false ... drop 0
   ⇒ L false L false L false ... L true drop -1
```

The above fully unrolled code (the last line) can be retained for re-use by other calls of function *q*. The sharing of this unrolled code helps to avoid some re-computation (involved in building this code/graph). However, it also takes up considerable amount of storage space and may force the program to abort if there is insufficient heap space. This form of laziness is dependent on the higher-order facility. It is not compatible with higher-order removal.

In contrast to the above program, the original function of *drop* is actually tail-recursive and could be optimised into an iterative loop which uses constant space. This could result in a space and time-efficient program. Other programs may not be tail-recursive but can still avoid the space-leak problem by sacrificing this form of laziness from partially applied recursive function calls.

A second more serious problem is that those FT-MFEs, which are *case* constructs, may cause loss of laziness when duplicated. Consider the following higher-order program:

```
dec Floss: num # truval → num;
dec app_twice: (alpha → beta) # alpha → beta;
---Floss(x,b)      ⇐ app_twice(case b in true ⇒ sqr;
                                    false ⇒ cube end,x);
---app_twice(f,x) ⇐ f(f(x)) ;
---sqr(x)           ⇐ x*x ;
---cube(x)          ⇐ x*x*x ;
```

where the *app_twice* call in function *Floss* contains a FT-MFE, *case b in true ⇒ sqr; false ⇒ cube end.* This MFE is also a higher-order argument which is to be eliminated. The present fully lazy $\mathcal{R}$ rule cannot extract it out because this would result in another intermediate function with the same higher-order argument. Instead, it can eliminate this function-type argument by the following sequence of steps which duplicates the FT-MFE.

```
---Floss(x,b)
    ⇐ app_twice(case b in true ⇒ sqr;
                          false ⇒ cube end,x);
                  ! unfold app_twice
    ⇐ case b in true ⇒ sqr; false ⇒ cube end
        (case b in true ⇒ sqr; false ⇒ cube end)(x);
                  ! eliminate curried application
    ⇐ case b in true ⇒
            sqr(case b in true ⇒ sqr x; false ⇒ cube x end);
        false ⇒
            cube(case b in true ⇒ sqr x; false ⇒ cube x end)
        end;
```

This duplication causes a loss of laziness because the *b* argument of the original *case* construct is a free variable. This argument is already known when the higher-order call is invoked. As a result, the same *case* selection will be executed twice! A possible remedy for this is to float out *case* constructs using the following *case* transformation.

$$\cdots \textit{case } t \textit{ in } p_1 \Rightarrow t_1; ..; p_n \Rightarrow t_n \textit{ end} \cdots$$
$$= \textit{case } t \textit{ in } p_1 \Rightarrow \cdots t_1 \cdots; ..; p_n \Rightarrow \cdots t_n \cdots \textit{ end}$$

whenever possible. In particular, this transformation can be carried out if the free variables of *t* are not bound by the context expression.

By moving *case* arguments outwards, this transformation will allow the selection variables to be shared more often. Applying this to the earlier function *Floss*, followed by higher-order removal, gives us the following first-order program:

```
---Floss(x,b)    ⇐ case b in true ⇒ app_twice(sqr,x);
                                false ⇒ app_twice(cube,x) end;
                                ! unfold app_twice
                    ⇐ case b in true ⇒ sqr(sqr(x));
                                false ⇒ cube(cube(x)) end;
```

The use of this *case* transformation to move selection variables outwards (to be shared) is similar to the techniques for binding-time improvements of *if* expressions with dynamic condition and static branches [CD91, HG91]. In their work, they use the continuation-passing style (CPS) style transformations to bring producers and consumers of intermediary data together. This was aimed at improving the partial evaluation transformation. A similar effect has been achieved by the above *case* transformation. However, we have a different motivation. We use this *case* transformation to help preserve full laziness without introducing higher-order expressions.

6 Conclusion

In this paper, we have presented a new method for preserving full laziness which does not depend on the higher-order facility. This method uses a collection of techniques which could extract out both explicit and implicit ground-type MFEs. An important contribution of this paper is that the new full laziness method is *compatible* with the higher-order removal method. It could be used to help remove non-linear function-type arguments with minimal loss of laziness. Preliminary investigations have also shown that this new full laziness method can produce better target programs.

Further work remains to be done. One area worth investigating is a more rigorous treatment of the notion of full laziness. One interesting exploration of full laziness was made by Holst and Gomard [HG91]. They extended the conventional notion of full laziness (for first-order lazy programs) by exploring additional program transformations needed to match the better sharing property of partial evaluation for eager programs.

We have informally suggested that the extractions GT-MFEs and FT-MFEs result in different forms of laziness. Most of the full laziness come from GT-MFEs, whilst the laziness of FT-MFEs can be largely recovered by direct unfolds and *case* transformations. The only unrecovered laziness is that due to recursive functions but this laziness is associated with the space-leak problem. These small results are only the beginning towards a better understanding of the issues of full laziness.

References

[CD91] Charles Consel and Olivier Danvy. For a better support of static data flow. In *5th ACM Conference on Functional programming Languages and Computer Architecture*, pages 496–519, Cambridge, Massachusetts, August 1991.

[CD92] Wei-Ngan Chin and John Darlington. Higher-order removal transformation technique for functional programs. In *15th Australian Computer Science Conference, Australian CS Comm Vol 14, No 1*, pages 181–194, Hobart, Tasmania, January 1992.

[Chi90] Wei-Ngan Chin. *Automatic Methods for Program Transformation.* PhD thesis, Imperial College, University of London, March 1990.

[HG85] Paul Hudak and B Goldberg. Serial combinators. *LNCS*, 201:382–389, 1985.

[HG91] Carsten Kehler Holst and Carsten Krogh Gomard. Partial evaluation is fuller lazinessq. In *ACM Symposium on Partial Evaluation and Program Manipulation*, pages 223–233, New Haven, Connecticut, August 1991.

[Hug82] John Hughes. Supercombinators: A new implementation method for applicative languages. *ACM Symposium on Lisp and Functional Programming*, pages 1–10, 1982.

[PJ87] Simon L Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International, 1987.

[PJL91] Simon L Peyton-Jones and David Lester. A moudlar fully-lazy lambda lifter in haskell. *Software - Practice and Experience*, 21(5):479–506, May 1991.

[Wad77] C P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford University, 1977.

Call Unfolding Strategies for Equational Logic Programs

David Sherman[*] Robert Strandh[†]
Laboratoire Bordelais de Recherche en Informatique,
Université Bordeaux I

Abstract

For a programming system based on term rewrite rules such as equational logic programming, a serious efficiency problem of the generated code is the creation of terms that only serve to drive further pattern matching. In this paper, we define a terminating call unfolding strategy based on fine-grain partial evaluation that removes much of this unnecessary term allocation for programs in intermediate EM code generated from equational logic programs.

Our approach is based on calculation, for each instruction, of two sets that reflect the usage of registers in finite execution paths of the program. These sets are calculated using fixed-point iteration over the graph representation of the intermediate code.

1 Introduction

Finding call unfolding strategies for partial evaluation is an annoying problem. On the one hand, the basic problem—knowing how much unfolding is necessary to expose a particular computation if it occurs—is undecidable. Nearly any recursive definition can lead to unbounded unfolding by the partial evaluator. On the other hand, there are a great number of common cases when it is clear that a certain amount of unfolding will permit significant performance gains. Consider the simple example of an expression $g(a,b) + f(a,b)$ in a functional setting where $f(x,y) = g(x,y)$; unfolding the f call once permits us to share the g call. Caught between possible untermination in the first case and obviously poor performance in the second, we search for heuristics, terminating strategies that give "good enough" results.

In this work we consider call unfolding in the context of equational logic programming. The actual unfolding and partial evaluation is performed on an imperative intermediate language, EM code, but as we will see the strategies depend fiercely upon properties of forward-branching equational logic programs and the pattern-matching automata we create for them.

[*]Funded by French Government under Bourse Chateaubriand N°065760C 1 2201H, with partial support from NSF grants CCR 9016905 and CDA 8822657.

[†]Partially funded by the ESPRIT basic research working group "Computing by Graph Transformations".

Our presentation here is in the same vein as Sestoft's consideration of call unfolding for functional languages in [Ses88]. In comparison with approaches for general functional languages, we have a somewhat degenerate case. Instead of calls to many functions, we start with calls to just one label. We have no partial input to these calls other than the values constructed in the original program. We have no notion of binding-time analysis to help us choose which calls can be safely unfolded.

On the other hand, EM code programs for equational logic programming have a fairly rigid and predictable structure. The programs operate in definite phases: match a left-hand side, build a right-hand side, restart the automaton. Moreover, the order in which the result of a previous step is inspected mirrors the order in which its parts were assembled. We will use these characteristics to advantage.

After briefly giving some background in section 2, we concentrate in section 3 on the call unfolding strategy we have developed, describing both the basic principles and the mechanics of implementing it. In section 4 we give some idea of the transformations made possible by this unfolding strategy, show a simple example, and measure the practical difference in terms of the number of basic operations performed by the compiled program. Finally, we mark some open questions in section 5, and conclude in section 6.

2 Background

The context of our work is the efficient implementation of equational logic programming. A given set of equations defines a rewrite system that reduces questions (subject terms) to provably correct answers (normal forms). The theoretical foundations of the work lie in the class of *forward-branching* systems[Str89], an important subclass of strongly sequential systems as defined by Huet and Lévy[HL91] and in turn indebted to Kahn's idea of a sequential predicate. Both classes are based on the construction of an *index tree*, which can be used to specify an efficient pattern-matching automaton for identifying needed redexes in the subject term. Forward-branching systems of equations have the important property that subterms participating in needed redexes can be reduced to head-normal form *before* they are inspected, letting us perform innermost stabilization without compromising the overall outermost reduction. We compile these automata into an intermediate code *EM code*[SS90] and significantly improve the running time of these systems by using semantics-preserving code transformations. We only consider transformations up to finding a head-normal form; transformation beyond head-normal forms would compromise the essential laziness of the system, a characteristic that is absolutely necessary to preserve logical completeness.

A critical efficiency concern is the construction of strictly intermediate forms, that is, portions of terms that are used solely to drive further pattern-matching in the reduction to head-normal form. This problem was identified in [Str87] and [Str88], where a form of *partial evaluation* was investigated. Wadler treated a similar problem for the case of lazy functional languages in [Wad88]. In [DSS91b] we treated the equational case more generally, showing how build (node construction) instructions can be used to drive program specialization, and in [DSS91a] we showed how recursive stabilization calls in the EM code program could be unfolded one instruction at a time.

EM code programs for forward-branching equations have a very regular structure: they form a tree where each internal node stabilizes and inspects a needed redex, and each leaf builds a right-hand side bottom-up and recursively calls the start of the program to continue reduction. It is these recursive calls that we specialize with respect to the partial input given by the build instructions that construct the right-hand side.

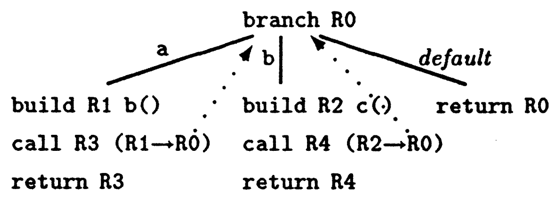
Since our main motivation is to eliminate unnecessary construction, we use the build instructions to drive the unfolding and specialization. We *push* these instructions down through the program, at the same time using them to specialize the program and delaying them as long as possible—ideally forever, that is, pushing them into little-executed branches or off of the program entirely. The key is a set of rewriting rules coupled with a register usage analysis. The analysis provides, for each instruction, two register sets named *usb* (used somewhere below) and *ueb* (used everywhere below). The former indicates whether a given register is used by the instruction or any instruction following it in the execution tree. The latter indicates whether the register is used in every possible path between the instruction to a return instruction, taking into account recursive calls. It should be clear that *ueb* is always a subset of *usb*. If the register written by a build instruction is not in the *usb* set of the following instruction, the build can be discarded. If, on the other hand the register written by the *build instruction* is in the *ueb* set, then the register written by the build is needed in every possible path to a return instruction and no further action is taken (the build is *stuck*).

The interesting case is the remaining: the register written by the build is in the *usb* of the following instruction but not in the *ueb*. There are two subcases. First, if the register written by the build is not used by the successor instruction, then the build instruction can be delayed, that is, pushed into the successor branches of the successor. Second, the register *is* used by the successor. The only way the register written by the build can be used by the successor instruction and yet not in the *ueb* set is if the successor instruction is a call. In this case, we can *unfold* the call. So we see that, practically speaking, the *ueb* set reflects not only whether registers are used in every path, but whether unfolding may pay off.

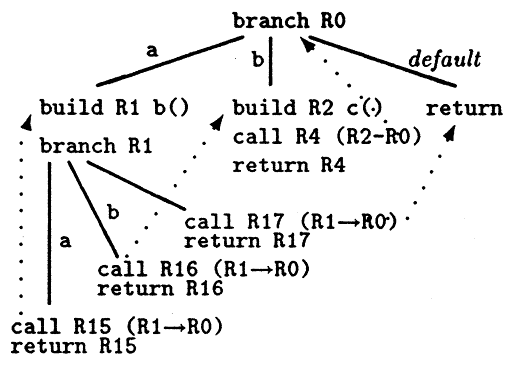
The semantics of EM code permit us to define *fine-grain* unfolding[DSS91a] of programs: we can unfold calls one instruction at a time while strongly preserving the program meaning at each step.

The rules that we use to unfold and specialize EM code programs are described in greater detail in [She92]. The question of *when* we unfold a call when we are able is the topic of section 3.

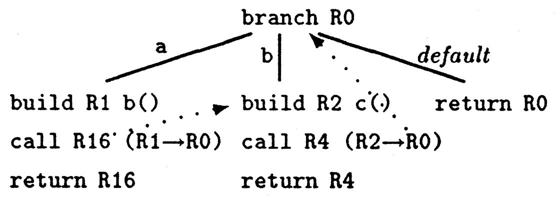
For a brief example, consider the two equations a = b and b = c. (In this and further examples we omit treatment of replace instructions, which would double the size of the examples. Since they only affect the speed of programs and not their correctness, we can safely gloss this detail in the interest of clarity.) The original automaton looks like this:



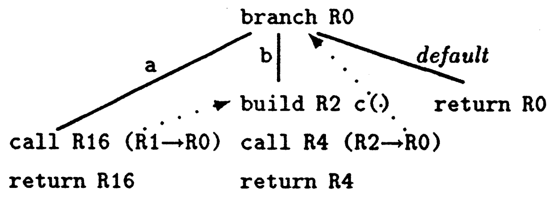
Our first step is to unfold the first instruction in the leftmost call, pulling out a branch instruction:



Now the build of b can be used to specialize the branch, leaving just the middle alternative.



Now, the body of the call does not use R0, so the call in the leftmost branch need not pass R1 as an argument. Therefore, the build is not stuck in front of the call, and can be pushed past it and subsequently off of the bottom of the program and discarded. The final program looks like this:



So when the transformed program sees an a, it immediately jumps to the b branch to build the correct result, c, without the unnecessary intermediate. We can rely upon the backend optimizations in our compiler to convert tail-recursive calls to jumps.

3 EM code Call Unfolding

In this section we describe a successful strategy for unfolding EM code calls for equational logic programs. We first describe the basic principle, and show that this principle provides a terminating strategy. Then, in section 3.2, we discuss the details of the computations of the dependency sets used by our strategy. Finally, in section 3.3, the quality of the resulting code is discussed.

3.1 Principle

Unfolding is driven by the `build` instructions rather than by the `call` instructions. A `call` is unfolded only when doing so permits a `build` to be pushed further down the instruction tree either into a path where it is always needed or into a path where it is never needed. The basic principle is that we can unfold instructions from a `call` for a given `build` until either:

1. the `build` is no longer used in the rest of the instruction tree; or
2. the build is needed in every path in the instruction tree.

When we say "used," we mean that the register is directly read by an instruction, not that it is an argument of a `call` instruction.

The *execution graph* of an EM code program has two kinds of edges, *flow* edges and *call* edges. The flow edges are those defined by the successor links in the instructions. The call edges are those defined by `call` instructions: we get a call edge between the calling node and the invoked node, and another call edge when we return to the caller. Following a register along a call edge in the execution graph engenders a register name translation based on the argument list and return register in the `call` instruction. The nodes of the execution graph are run-time states of the program. Since EM code programs are deterministic, the execution graph is defined straightforwardly, but we do not use it directly. First, it is usually infinite; second, when analyzing programs we sometimes want to skip over recursive invocations that represent arbitrarily long (or even infinite) paths in the execution graph. When we speak of *paths* in the following it should be understood that we mean paths in an approximation of the execution graph that may skip over recursive invocations, automatically performing the register name translation. Naturally the implementation of the partial evaluator must keep track of these details.

The heart of the unfolding strategy is the definition of two sets, *usb* and *ueb*, defined for each instruction in a given program. They are both a form of register analysis.

The set $usb(I)$, *used somewhere below*, is the set of registers needed by a `branch` or a `return` instruction J along a finite path from I to J. A register is directly needed if it is read by a `return` instruction or a `branch` instruction. The former determines a value to be returned to the caller, and the latter determines program state. A register can, of course, be indirectly needed by being read by an instruction (such as a `down`) writing a register which in turn is needed by a `return` or a `branch` instruction. If a register is used only along infinite paths from I then it is not in $usb(I)$. In practical terms *usb* is a register lifetime analysis that tells us what register values might be needed after a given instruction. The semantics of EM code tell us, not unreasonably, that if a register written by an instruction I is not in *usb* of the successor of I, then I can be discarded. In particular, if I is a `build` instruction, the memory allocated by I is not needed.

In our implementation, *usb* is used in a *dependency analysis* that removes instructions that cannot affect the result.

The set *ueb*, *used everywhere below*, is a set of registers such that if $R \notin ueb(I)$, then there is an integer n such that every path from I of length n reaches an instruction I' such that either

1. I' is a `return` instruction on the same invocation level as I that does not use R, or
2. $R \in ueb(I')$,

where a call C is followed only if $R \in usb(T)$ where T is the target of C. That is, $R \notin ueb(I)$ only if, along some execution path, there is a branch where R is not needed. The reason that I' must be on the same invocation level in the case of a `return`, is that otherwise we cannot guarantee an upper bound on the length of a path to a place where the register is needed, as we may go though a `call` instruction where the target, but not the successor, have such a bound on the length of the paths. On the other hand, requiring bounded paths both on the target and the successor of a call instruction would be too restrictive, especially when the target does not use the register in question. Thus the rule that makes such a call a single step.

In practical terms *ueb* tells us when we might profitably move a `build` instruction. Suppose that R is a register written by `build` instruction I with successor J. The first two cases are easy.

Case 1: $R \notin usb(J)$. I can be discarded.

Case 2: $R \in ueb(J)$. I is considered *stuck* and is no longer considered for unfolding.

Case 3: R is not read by instruction J. Push I to the successors of J.

The interesting case remains: $R \in usb(J)$ but $R \notin ueb(J)$, and it is not read by J.

Case 4: J must be a `call` instruction. Unfold the call.

If we suppose that we can construct the sets *usb* and *ueb*, then the termination of the unfolding follows directly. Each transformation of the program is driven by a `build` instruction; in each step, either: the `build` is discarded, the `build` is no longer considered for unfolding, or the finite paths from the `build` to places where transformations can take place are shortened by one step. We unfold calls in front of a `build` until either it is no longer needed, or it is still needed but needed everywhere.

3.2 Computing the Dependency Sets

It remains to show how we find *usb* and *ueb*. Both of these sets are constructed with a fixed-point analysis of the program. Currently we perform these analyses after every transformation, as discussed further in section 5. In this section we describe the constructions, and sketch proofs that these constructions satisfy the properties stated in the preceding section.

In the case of *usb* we start with $usb(I) = \emptyset$ for all I. Next, for R read by I where I is either a branch or a return instruction put R in $usb(I)$. Finally, propagate the *usb* sets to the predecessors of each instruction translating registers written by an instruction to registers used by the instruction. Propagation of a register stops when it reaches the instruction where the register was written or when it reaches an instruction I such that $R \in usb(I)$. For a branch instruction the union of the *usb* of the successors is computed. Finally, for a call instruction, the union of the *usb* of the successor and the target is computed. The last rule makes the *usb* calculation approximative, as the target of the call may contain only infinite paths, but we conclude, erroneously, that there is a finite execution path to its successor. For EM code programs generated from equational logic programs, this is a very good approximation. In the worst case, we do not detect the possibility of discarding a useless instruction.

For the *ueb* set, we initialize $ueb(I)$ to $usb(I)$ taking the pessimistic attitude that if a register is used somewhere, it is used everywhere. Fixed-point iteration is done by computing, for each branch instruction, the intersection of the *ueb* sets of its successors. The register read by the branch instruction is deleted from *ueb*. This rule essentially says that it is profitable to unfold if the term created in a build instruction is inspected after a finite number of steps. For a call instruction the union between the corresponding sets of the successor and the target is computed. Any modification is propagated to the predecessor as usual. Propagation of a register stops when it reaches an instruction where the register is written or read.

Proving that the computations of the *ueb* and *usb* sets preserve the properties stated above is done by induction over the fixed-point construction and case analysis on the type of instruction. For *ueb*, in particular, the basis case is $ueb = usb$, so every possible path has length zero and is trivially finite.

3.3 Quality of Transformed Code

It is obvious from the definition of the strategy that build instructions are preserved only if what is built is needed in every finite path from the build instruction to a return instruction on the same invocation level, or in other words, what is built is actually returned to the caller. However, we still have to show that this strategy improves on particular programs. For that, consider what happens when we combine this criteria with the particular structure of automata for forward-branching sets of equations.

Recall that an EM code program representing a pattern-matching automaton starts with a branch instruction for the root of the subject term, and then for each index node we have a recursive stabilization followed by an inspection of the stable form of the index. When we unfold a call in a leaf of an automaton, the first instruction that we unfold is typically a branch on the root symbol of the newly-constructed right-hand side. Since the right-hand side is constructed bottom-up, the root was built last and so the build that unfolded the call is the same node inspected by the unfolded branch! So quite often the unfolded branch at the beginning of the call body consumes the constructed node immediately. Furthermore, the automata for forward-branching sets of equations stabilize—with a recursive call—before inspecting, so subterms in the right-hand side trigger unfolding that consume them in the same way. Symbols that are roots of indexes in the index tree corresponding to the EM code program are built only in default branches, cases where no left-hand side was matched.

The important observation here is that the unfolding strategy is particularly suited to the structure of the EM code programs we generate. Tailoring the strategy to our problem domain was an important step in getting good results.

4 Results

The simple unfolding strategy outlined above produces good code for our programs. In this section we discuss the quality of the code produced by our partial evaluator. We first describe the kinds of transformations we can expect, then work through a simple example to show the improvements. Finally, we discuss some areas in which the results can be improved.

As we recall from the previous section, our unfolding strategy permits the partial evaluator to remove any build of a node that will be an index in the next reduction step. What does this mean in practical terms?

1. The root of a right-hand side is built only if it is a head-normal form; otherwise the effect of the build is represented by restarting the automaton in the appropriate state. For equations written in a functional style, the dispatch on the function name (head symbol) is done without need for construction.
2. Type constructors are not built if all references to them are accounted for in the next rewrite. This handles the annoying box-unbox-box overhead seen when dealing with boxed values (like integers).
3. Recursive definitions become automaton loops, that is, the recursion is directly in the automaton, not indirectly by way of construction and stabilization of an intermediate. The automaton loops may still contain calls and so forth, as necessary.

Thus the transformed EM code programs share some properties with lazy functional programs after some strictness analysis (we are reminded here of Fairbairn and Wray, [FW87]): nonstrict subcomputations are represented by suspensions built in the heap, and strict subcomputations are represented by code to directly compute the result.

4.1 An Example

To get a clearer idea of what is going on, let us look at an example. Consider a simple set of equations:

```
rev[x] = catrev[x;()];
catrev[();y] = y;
catrev[(x . xs); y] = catrev[xs; (x . y)].
```

The EM code program constructed for this example is shown in figure 1.

What happens when we run the partial evaluator on this example? The transformed EM code program is shown in figure 2. First, we see that the catrev that would have been built in the rev branch has been unfolded: rev[nil] is directly rewritten to the normal form nil, and rev[cons[...]] jumps into the catrev branch without intervening builds. Second, in the catrev branch we see that the traversal of the cons-chain is turned into a *automaton loop* that only builds the cons nodes

```
0       branch[R0,
                rev    : 21
                catrev : 35
                DEFAULT 44 ];
21      R16 := down[R0, 1];
125     R17 := build[nil,];
126     R15 := build[catrev, R16 R17];
127     R18 := call[0, R0:=R15];
128     return[R18];
35      R19 := down[R0, 1];
36      R20 := call[0, R0:=R19];
132     branch[R20,
                cons  : 38
                nil   : 140
                DEFAULT 143];
38      R23 := down[R20, 1];
39      R24 := down[R0, 2];
11      R22 := build[cons, R23 R24];
12      R25 := down[R20, 2];
137     R21 := build[catrev, R25 R22];
138     R26 := call[0, R0:=R21];
139     return[R26];
140     R27 := down[R0, 2];
141     R28 := call[0, R0:=R27];
142     return[R28];
143     return[R0];
44      return[R0];
```

Figure 1: Example before transformation.

for the eventual result. In fact the transformed program builds no node that is not necessary for either the final result or for logical correctness. Notice that the "default" branches are no longer trivial: if there is a failure of pattern-matching, we must build the correct head-normal form.

If we compare the number of basic operations performed by the modified code, we see substantial improvements. Figure 3 shows the number of **down**, **branch**, and **build** instructions before and after program transformation, for a subject term of 600 nested reverses of a list of 600 elements. The transformed program builds and inspects half as many nodes, and performs one-third fewer inspections.

5 Room for Improvement

There are several areas in which this work can be improved. The foremost open problem is to find a terminating strategy that permits us to unfold build instructions from calls. It is not hard to construct cases where desirable transformations depend upon exposing (by unfolding) a build hidden inside of a call. For a simple example, consider the equations

```
a = b;
f[b,x] = c;
g[x] = f[a;x].
```

The problem is that the result of a reduction participates in a reduction above it. We cannot infer that g[x] = c unless the a constructed by the third rule is unfolded and pushed through the unfolded f branch.

Unfortunately it becomes difficult to ensure termination if we permit builds to be unfolded. While the propagation step outlined in section 3.1 will eventually stop for a given build, that doesn't help if we are steadily increasing the number of them we propagate. On the other hand, prohibiting unfolded builds from triggering the unfolding of further builds seems too restrictive.

```
0       branch[R0,
                rev    : 21
                catrev : 35
                DEFAULT 44];
21      R16 := down[R0, 1];
22      R1042 := call[0, R0:=R16];
23      branch[R1042,
                cons  : 24
                nil   : 28
                DEFAULT 30];
24      R1054 := down[R1042, 1];
25      R17 := build[nil,];
26      R1043 := call[11, R20:=R1042 R23:=R1054 R24:=R17];
27      return[R1043];
28      R1055 := build[nil,];
29      return[R1055];
30      R1057 := build[nil,];
33      R1051 := build[catrev, R16 R1057];
34      return[R1051];
35      R19 := down[R0, 1];
36      R20 := call[0, R0:=R19];
37      branch[R20,
                cons  : 38
                nil   : 40
                DEFAULT 43];
38      R23 := down[R20, 1];
39      R24 := down[R0, 2];
11      R22 := build[cons, R23 R24];
12      R25 := down[R20, 2];
13      R1029 := call[0, R0:=R25];
14      branch[R1029,
                cons  : 15
                nil   : 18
                DEFAULT 45];
15      R1041 := down[R1029, 1];
16      R1030 := call[11, R20:=R1029 R23:=R1041 R24:=R22];
17      return[R1030];
18      R1113 := call[0, R0:=R22];
19      return[R1113];
45      R1038 := build[catrev, R25 R22];
46      return[R1038];
40      R27 := down[R0, 2];
41      R28 := call[0, R0:=R27];
42      return[R28];
43      return[R0];
44      return[R0];
```

Figure 2: Example after transformation.

Operation	Base	Transformed	Ratio
stabilize	1201	1201	1.0000
down	1441800	720600	0.4998
branch	1083601	723001	0.6672
build	721800	361200	0.5004

Figure 3: Basic operations for **rev** example, before and after partial evaluation.

Another deficiency of our present implementation is that the register usage analysis is too expensive. Currently, we update the register usage annotations on program subtrees after each program transformation. This is woefully inefficient, as one might hope that in most cases a transformation only makes small changes to the annotations. We have not yet found a satisfactory *incremental* definition of this analysis.

6 Conclusion

We have defined a call unfolding strategy for intermediate code from forward-branching equational logic programs. The purpose of the strategy is to eliminate unneeded term construction, an expensive overhead in equational logic programming as compared to applicative programming. Our strategy performs well on typical programs and has the advantage of being terminating. Although the algorithm currently used is not very efficient (quadratic in the size of the program), there are indications that this complexity can be improved by rather simple methods.

Although our strategy eliminates unnecessary term construction as can be determined by left hand sides, it is clear that a more advanced strategy is needed in order to take full advantage of the possibilities for optimizing intermediate code. A minimal requirement for such an advanced strategy is the ability to follow reductions, thus taking into account right hand sides in further optimization. Doing so clearly introduces the danger of nontermination. A simple terminating strategy that seems good enough has been tested in a slightly different context. This strategy allows us to follow reductions only if the relative size of the subject term decreases monotonically. We currently do not know whether such a strategy is possible on intermediate code.

A potentially much harder problem occurs because a program in EM code can be called externally. This makes it necessary to return full terms to any caller. Sometimes, however, it would be advantageous to return immediate mode data, such as integers. To handle this problem, we need to specialize our intermediate program so that the *caller* is responsible for so-called *boxing* of values if necessary. Our fine-grain unfolding does not handle this situation automatically, although it can be used to further optimize a specialized program.

References

[DSS91a] Irène Durand, David J. Sherman, and Robert I. Strandh. Fine-grain partial evaluation of intermediate code from equational programs. In *Journées de Travail sur l'Analyse Statique en Programmation Equationelle, Fonctionelle et Logique*. Bigre Journal 74, Octobre 1991.

[DSS91b] Irène Durand, David J. Sherman, and Robert I. Strandh. Optimization of equational programs using partial evaluation. In *Proceedings of the ACM/IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, 1991.

[FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Functional Programming Languages and Computer Architectures*. Springer-Verlag, 1987.

[HL91] Gérard Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems, ii. In *Computational Logic: Essays in Honor of Alan Robinson*, chapter 12, pages 415–443. The MIT Press, 1991.

[Ses88] Peter Sestoft. Automatic call unfolding in a partial evaluator. In Bjørner, Ershov, and Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. Elsevier, 1988.

[She92] David J. Sherman. EM code semantics, analysis, and optimization. Technical Report Rapport 92-04, Greco de Programmation du CNRS, 1992.

[SS90] David J. Sherman and Robert I. Strandh. An abstract machine for efficient implementation of term rewriting. Technical Report 90-012, University of Chicago Department of Computer Science, 1990.

[Str87] Robert I. Strandh. Optimizing equational programs. In *Proceedings of the Second International Conference on Rewrite Techniques and Applications*, 1987.

[Str88] Robert I. Strandh. *Compiling Equational Programs into Efficient Machine Code.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1988.

[Str89] Robert I. Strandh. Classes of equational programs that compile into efficient machine code. In *Proceedings of the Third International Conference on Rewrite Techniques and Applications*, 1989.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Second European Symposium on Programming*. Springer-Verlag, 1988.