Partial Evaluation and Mixed Computation
D. Bjørner, A.P. Ershov and N.D. Jones (Editors)
Elsevier Science Publishers B.V. (North-Holland)

Automatic Call Unfolding in a Partial Evaluator

Peter Sestoft

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark

Partial evaluation of a functional language may be done by specialization of functions, implemented by reduction of expressions and unfolding of function calls. Expression reduction is fairly straightforward, but it is difficult to decide when to unfold a function call and when to leave it in the program produced by partial evaluation. A call unfolding strategy must be adopted in order to make this decision in each particular case.

In the partial evaluators reported in the literature, this strategy is not formalized, and the user of a partial evaluator must guide the call unfolding by his own informal criteria. This is an obstacle to many applications of partial evaluation.

In the present paper we describe a simple two-phase call unfolding method which is fully automatic. The first phase is based on static call annotations, obtained by a local analysis to avoid infinite unfolding and a global analysis to avoid call duplication. The second phase does additional unfolding based on a global analysis. The method has been employed in a recent version of the partial evaluator mix, developed by Neil D. Jones, Harald Søndergaard, and the author.

1. INTRODUCTION

The program transformation technique called partial evaluation has attracted much attention in recent years and partial evaluators have been developed for a variety of languages and purposes. Most partial evaluators however require some user assistance, and so are not fully automatic program transformers. This is true also of the early versions of mix, a partial evaluator for a small subset of pure Lisp developed by Neil D. Jones, Harald Søndergaard, and the author.

In this paper we describe the problems that required user assistance in mix and propose a rather simple way of automating their solution, thus achieving fully automatic partial evaluation of a large class of programs. In particular, this class includes the partial evaluator itself, and so the partial evaluator is self-applicable.

The setting of the problem we attempt to solve is this: the partial evaluator produces a residual program from specializations of the given subject program's functions. This is done by a function specialization phase which *reduces expressions* and *unfolds function calls* in the subject program, making use of its known input. Reduction of an expression depends only on the form of the expression and the symbolic values of the variables occurring in it and so is quite straightforward. On the other hand, the decision whether a call should be unfolded or not requires a

more global view of the program, in order to avoid infinite unfolding for example. For this reason, call unfolding was left to be decided by the user in the early versions of mix. This implied that the user had to understand the workings of mix as well as of the program to be partially evaluated. Naturally, this has provided an obstacle to its application in some cases. Therefore it would be desirable to automate the decision on call unfolding in the function specialization phase.

The solution described in this paper exploits the results of the binding time analysis (done by the first phase of mix) to find calls that may safely be unfolded (during the function specialization phase), and forbids unfolding of all other calls. Because of the simplicity of this strategy, the program produced by function specialization will have many very small functions and will hardly be readable to humans. Therefore a postprocessing phase is employed that unfolds calls to such small functions.

The structure of the rest of the paper is as follows. First, we briefly introduce partial evaluation and outline the structure of the partial evaluator mix in Section 2. On that background we discuss call unfolding in Section 3. The method we have developed is presented in Section 4, and results from its use are discussed in Section 5. The paper closes with a summary, acknowledgements, and a list of references.

2. BACKGROUND

In this section we first give a very brief introduction to partial evaluation and then describe the main features of the partial evaluator mix to provide background for the discussion in Section 3 and the description of our method in Section 4.

2.1 Partial Evaluation

Partial evaluation deals with specialization of programs. Suppose we are given a program p with two input parameters, and suppose that the value d_1 of the first parameter is available while that of the second parameter is not. Then obviously the program p cannot be evaluated to yield a result. It may however be *partially evaluated* with respect to the known value of the first parameter. This yields another program r, a so-called *residual program*. This program will compute the "rest" of p when given a value d_2 of the second parameter. That is, when evaluated on the remaining input, the residual program r will give the same result as the original *subject program* p when evaluated on all of its input, or in symbols,

$$\text{Eval p } (d_1, d_2) = \text{Eval r } (d_2) \quad \text{for all data } d_2.$$

A program that will produce a residual program r when given a subject program p and part of its input d_1 is called a *partial evaluator*. A partial evaluator that may be used to partially evaluate itself (letting the subject program p be the partial evaluator itself) is said to be *self-applicable*.

A partial evaluator may be used for many interesting purposes, for instance, to compile by partial evaluation of an interpreter with respect to a source program. This is useful for cheaply implementing special purpose languages such as the language of pattern expressions as interpreted by a general purpose pattern matcher. Furthermore, a self-applicable partial evaluator may be used for generating compilers (by partial evaluation of the partial evaluator with respect to an interpreter), and for generating a compiler generator (by partial evaluation of the partial evaluator with respect to itself).

Precursors to partial evaluation are found in Kleene's S-m-n theorem, in optimizing compilers, and in the paper [Lombardi 1967]. Futamura was the first to describe partial evaluation as a topic in its own right and to see its possible applications in compilation and compiler generation [Futamura 1971]. The first expositions of the possibility of generating a compiler generator are found in [Beckman *et al.* 1976], [Turchin 1979] and [Ershov 1982]. A partial evaluator for Lisp (with property lists and imperative features) is described in [Beckman *et al.* 1976]; partial evaluation of imperative languages is discussed in [Ershov 1978] and [Bulyonkov 1984].

2.2 The Structure of the Partial Evaluator Mix

The subject language (*i.e.*, language of subject programs) of mix is Mixwell, a small subset of pure Lisp with lexical scoping. A Mixwell program is a list of definitions of functions f_1, ..., f_h with the first function f_1 as the goal function. Input to the program is through the variables of that function:

$$\begin{aligned} f_1\,(x_1,\ldots,x_{k1}) &= e_1 \\ &\ldots \\ f_h\,(y_1,\ldots,y_{kh}) &= e_h \end{aligned}$$

The body e_i of function f_i is constructed from variables appearing in the variable list of f_i, from constants (quote ...), and operators atom, car, cdr, cons, equal (as in Lisp), a conditional if, and a function call call. The only data type is well-founded (*i.e.*, non-circular) S-expressions as known from Lisp. All operators except the conditional if are strict in all positions, and defined functions are called by value.

The partial evaluator is divided into two major phases:

- a binding time analysis phase, and
- a function specialization phase.

The *binding time analysis phase* takes as input a description of which parts of the program's input are known, and does a global analysis of the subject program to compute a description of each variable of each function as either Static or Dynamic. A variable is described as Static if it can take on only values dependent on the known input, and as Dynamic if it may possibly take on a value dependent on the unknown input. During the function specialization phase, a Static variable will have an ordinary value such as '(a b) whereas a Dynamic one will in general have a symbolic value, possibly containing variables, for example (cons '(a b) (car x)). The variable list of every function is split into two variable lists: one for Static variables and one for Dynamic variables.

The *function specialization phase* takes as input the actual values of the known input and produces a residual program built from specializations of the subject program's functions $f_1, ..., f_h$. A function f_i is specialized by symbolically evaluating its body e_i in a symbolic environment that binds each of its variables to an expression. Symbolic evaluation of other expressions than function calls is simple reduction, but for a function call (call f $se_1...se_m$ $de_1...de_n$) it must be decided whether it is to be unfolded or suspended (*i.e.*, not unfolded). Here $se_1...se_m$ are the argument expressions for the Static variables of f, and $de_1...de_n$ are the argument expressions for the Dynamic variables of f.

If the call is to be *unfolded*, then the call (call f $se_1...se_m$ $de_1...de_n$) is replaced by the body of f with the symbolic values of the argument expressions $se_1...se_m$ $de_1...de_n$ substituted for variable occurrences, and the resulting expression is symbolically evaluated. Hence a call that is unfolded during the function specialization phase disappears completely.

If the call is to be *suspended*, then the call expression is replaced by a call to an f-variant f*, the variables of which are f's Dynamic variables. Let $se_1*...se_m*$ be the (ordinary) values of the Static argument expressions $se_1...se_m$, and let $de_1*...de_n*$ be the symbolic values of the Dynamic argument expressions $de_1...de_n$. The resulting expression then is (call f* $de_1*...de_n*$). The variant f* is constructed by specializing the body of f to the values $se_1*...se_m*$ of its Static variables. So symbolic evaluation of a call (call f $se_1...se_m$ $de_1...de_n$) that is to be suspended will result in a specialized call (call f* $de_1*...de_n*$) to a specialized function f*.

The *difficult* point here is to decide whether a call met during symbolic evaluation is to be unfolded or to be suspended. To make good use of the known input, not too many calls should be suspended, whereas to make symbolic evaluation terminate, not too many calls should be unfolded. Unfolding the "wrong" calls may produce enormous residual programs, or may make them monstrously slow. In Section 3 we shall study the various pitfalls to avoid when developing a call unfolding strategy.

For a much more comprehensive description of mix, see [Jones, Sestoft, Søndergaard 1985], [Sestoft 1986], or [Jones, Sestoft, Søndergaard 1987].

3. PROBLEMS WITH CALL UNFOLDING

This section discusses call unfolding strategies and the various problems to be avoided when choosing a call unfolding strategy.

3.1 Call Unfolding Strategies

Call unfolding takes place in the function specialization phase of the partial evaluator as described in Section 2.2 above. The topic of this section is the various possible kinds of call unfolding strategies. Two obvious extremes as regards call unfolding strategies are:

- Never unfold any call
- Always unfold all calls

Neither of these is useful in general. The first alternative, never to unfold any call, will lead to trivial residual programs and bad partial evaluation results as shown in Section 3.2 below. On the other hand, the second alternative, always to unfold all calls, will cause the partial evaluator to loop in all but trivial cases as shown in Section 3.3 below.

We must look for a call unfolding strategy which is an intermediate between these two extremes: a decision on unfolding must be made for each particular call expression. There are two very different ways to make this decision:

- by a *dynamic strategy*, making the decision anew each time the call expression is met during the function specialization phase.
- by a *static strategy*, making the decision for each textual call in advance of the function specialization phase.

The difference is that by a dynamic strategy, if the same textual call expression is met several times during function specialization, it may be unfolded on one occasion and not unfolded on another. This does not happen with a static strategy; in this case either the call expression is unfolded every time it is met, or it is suspended every time. Dynamic strategies are more flexible and may give better results than static

ones, and the class of dynamic strategies properly contains the class of static ones. Fuller and Abramsky describe a partial evaluator for Prolog employing a dynamic strategy based on loop detection [Fuller, Abramsky 1987].

The partial evaluator mix described in Section 2.2 above is restricted to the use of static strategies; this requires to decide on unfolding/suspension for each call appearing in the text of the subject program in advance of function specialization. This is mainly for reasons of simplicity: a dynamic strategy would require the function specialization phase of the partial evaluator to maintain some extra data structures to guide the dynamic unfolding decisions. The restriction to static strategies allows to represent the unfolding decision by a simple annotation of each call expression.

With static call unfolding strategies, four possible pitfalls can be clearly identified and will be discussed by means of examples below:

- too little unfolding, yielding trivial residual programs,
- too much unfolding, making partial evaluation loop,
- unfolding in the wrong place, yielding very slow residual programs,
- unfolding in the wrong place, yielding very large residual programs.

Furthermore, there are situations where no satisfactory decisions on call unfolding can be made in the (static) framework of mix. These have to do with infinite specialization. One such case will be discussed in Section 4.3 below.

3.2 Trivial Residual Programs

Too little unfolding happens only when we decide to suspend a call to a function with only Static variables. Consider the program

```
g (x z) = (if (call f x) then z else (cons z z))
f (y)   = (null y)
```

and assume x = 'nil to be known, z unknown. Then x and y are Static, z is Dynamic, and if the call to f were (wisely) unfolded, we would get the residual program

```
g (z)   = z
```

If however (stupidly) the call to f is suspended, then the only thing that can be done is to specialize f to the value 'nil of its Static parameter y, and so we would get

```
g (z)   = (if (call f) then z else (cons z z))
f ()    = 't
```


Moral: Calls to functions with only Static variables should always be unfolded. Note that this may still make the function specialization phase loop in case the subject program already contains a non-terminating loop that does not depend on Dynamic variables.

3.3 Infinite Unfolding

Too much unfolding may make function specialization loop infinitely. Consider the program

```
g (x z) = (if (equal x (car z)) then (cdr z)
          else (call g x (cdr z)))
```

and assume x = 'A to be known and z to be unknown. If we (wisely) choose to suspend the call to g, we will get g specialized to the value 'A of its Static variable x. The residual program would be

```
g (z)   = (if (equal 'A (car z)) then (cdr z)
          else (call g (cdr z)))
```

If on the other hand we (stupidly) attempted to unfold the call every time it is encountered during function specialization, then we would in effect try to build an infinite expression:

```
g (z)   = (if (equal 'A (car z)) then (cdr z)
          else (if (equal 'A (cadr z)) then (cddr z)
          else (if (equal 'A (caddr z)) then (cdddr z)
          else ...
```

The observable effect however is that partial evaluation will not terminate in this case. Moral: Every strategy for call unfolding must somehow ensure that infinite unfolding is not attempted. This may be done by imposing an arbitrary limit on the number of unfoldings that may take place, or (better) by allowing unfolding only where a bound on the number of unfoldings is known to exist. The latter idea is applied in our method as described in Section 4.1.1 below.

3.4 Call Duplication

Extremely slow residual programs may result from call duplication. This happens when an argument expression of a call to be unfolded contains a suspended call, and this (inner) call is duplicated by unfolding the outer call. Consider this program,

the run time of which is linear in the length of (list) z:

```
g (x z) = (if (null z) then x
          else (call f (call g x (cdr z))))
f (w)   = (cons w w)
```

with x = 'A known and z unknown. From the similarity to the previous example it should be clear that the call to g must be suspended, or else infinite unfolding will result. If we (wisely) suspend the call to f too, we will obtain the reasonable residual program

```
g (z)   = (if (null z) then 'A
          else (call f (call g (cdr z))))
f (w)   = (cons w w)
```

which still has run time linear in the length of z. If however (stupidly) the call to f is unfolded, we will get

```
g (z)   = (if (null z) then 'A
          else (cons (call g (cdr z)) (call g (cdr z))))
```

which has run time exponential in the length of z. The reason is that the call to g was duplicated and this happened because the body of f has two occurrences of the variable w in the argument position corresponding to the one with the call to g. We refer to this phenomenon as call duplication. Moral: A call to a function f with a suspended call to a function g in an argument expression should be unfolded only if the variable corresponding to that argument position appears at most once in the body of f. In fact, it is sufficient to require that the variable appears at most once in any conditional branch in the body of f.

3.5 Code Duplication

Extremely large residual programs may result from code duplication. This is a phenomenon similar to the above, but in this case the *size* (and not necessarily the run time) of the program explodes. This happens when an argument expression of a call to be unfolded contains a (sizeable) residual expression, and that expression is duplicated by unfolding the call. Consider the program

```
g (x z) = (if (null x) then z
          else (call f (call g (cdr x) z)))
f (w)   = (cons w w)
```


with x = '(A A A) known and z unknown. The call to g may be unfolded or suspended; neither will lead to trouble, so assume that it will be unfolded. If we (wisely) suspend the call to f, we get the residual program

```
g (z)   = (call f (call f (call f z)))
f (w)   = (cons w w)
```

and in general, if x is a list of length n, the body of g in the residual program will contain n nested calls to f. If however (stupidly) the call to f is unfolded, we will get (again for x = '(A A A)),

```
g (z)   = (cons (cons (cons z z) (cons z z))
                (cons (cons z z) (cons z z)))
```

and in general, if x is a list of length n, the body of g will have 2^n-1 cons operators and 2^n occurrences of the variable z. Moral: A call with a sizeable residual argument expression should be unfolded only if the variable corresponding to that argument position appears at most once in the body of the unfolded function.

4. SIMPLE ANNOTATIONS AND POSTPROCESSING

In this section we describe our method for deciding on call unfolding. It has two phases. First, a *preprocessing* phase computes annotations to direct the call unfolding that takes place during the function specialization phase. This way a simple static call unfolding strategy is implemented. Secondly, the residual program resulting from the function specialization phase is improved by a *postprocessing* phase that eliminates simple functions by unfolding the calls to them.

These two phases are described in Sections 4.1 and 4.2 below. The section closes with a discussion of the limitations of the method in Section 4.3.

The preprocessing phase of our call unfolding method is based on the results of the binding time analysis done by mix and hence must be preceded by that. Thus the phases fit with the phases of mix in the way illustrated by Figure 4.1 on the next page.

4.1 Avoiding Infinite Unfolding and Call Duplication

The basic idea is to use a *static* call unfolding strategy, that is, to decide on call unfolding *in advance* of function specialization. Unfolding decisions are made for each individual call expression appearing in the subject program and are represented by annotations of the call expressions.

If the call is to be suspended every time it is met during function specialization, then it is annotated with an "r" yielding callr. This will be referred to as a *residual call*, that is, one to be left in the residual program. If the call is to be always unfolded during function specialization, then it is not marked, and will be referred to as an *eliminable call*. So by definition every residual call met during function specialization will be suspended, and every eliminable call met will be unfolded.

The annotations (representing call unfolding decisions) must satisfy at least two requirements as discussed in Section 3: there must be no infinite unfolding, and no call duplication. A set of annotations that satisfies these two requirements is found by two analyses done in separate subphases of the call annotation phase. In the first subphase, sufficiently many calls are made residual to ensure that infinite unfolding cannot take place during function specialization. In the second subphase, more calls are made residual to remove possible call duplication risks until there is no such risk anymore. Obviously, making more calls residual cannot reintroduce a risk of infinite unfolding.

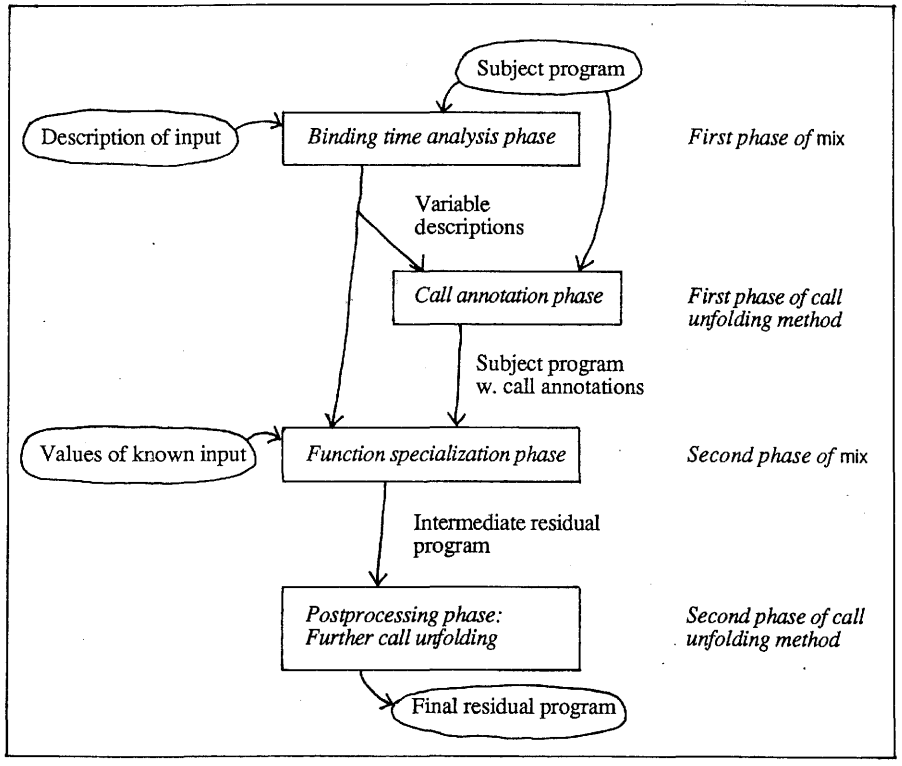

Figure 4.1: Structure of Mix with Call Unfolding

Both of the subphases are designed to avoid the pitfall of too little unfolding (when all variables in a call are Static), but no special care is taken of the risk of code duplication. Both of the subphases make use of the description of each variable as Static or Dynamic which is computed by the preceding binding time analysis phase of mix. The first analysis subphase, avoidance of infinite unfolding, is quite simple. It is based on the recognition of inductive variables and is described in Section 4.1.1 below. The second analysis subphase, avoidance of call duplication, is more involved. It alternates between a duplication risk analysis and a transformation that makes more calls residual. This second subphase is described in Section 4.1.2.

4.1.1 Simple Annotations Based on Inductive Variables

This first subphase of the call annotation phase makes call annotations that guarantee absence of infinite unfolding. When deciding on the annotation of calls, two cases are distinguished: a call to a function having only Static variables, and a call to a function with at least one Dynamic variable.

A call to a function with only Static variables is always made eliminable as was argued for in Section 3.2.

The decision for a call to a function with at least one Dynamic variable makes use of the concept of an *inductive variable*. The idea is to make a function call eliminable only if it can be ensured that it cannot be unfolded infinitely during function specialization. Infinite unfolding can of course happen only when a function calls itself recursively (possibly through calls to other functions).

To simplify the discussion (and the algorithm) we consider only the case of a direct recursive call from a function f to itself,

$$f\ (sv_1 \ldots sv_m\ dv_1 \ldots dv_n) = \ldots (\text{call } f\ se_1 \ldots se_m\ de_1 \ldots de_n) \ldots$$

Here the $sv_1 \ldots sv_m$ are the Static variables of f, and the $dv_1 \ldots dv_n$ are the Dynamic ones. Correspondingly, the $se_1 \ldots se_m$ are the Static argument expressions, and the $de_1 \ldots de_n$ are the Dynamic ones.

We shall say that a Static variable sv_i is *inductive* in the call from f to itself if and only if the corresponding argument expression se_i computes a value which is a proper substructure of the value of sv_i. For instance, se_i may be (car sv_i) or (cdr sv_i).

Let us define that a call satisfies the *structural induction condition* if there is at least one inductive Static variable in the call, and the remaining Static variables are either unchanged or inductive.

A call satisfying the structural induction condition cannot be unfolded infinitely often, and so it can safely be made eliminable. To see this, consider the totality of f's Static variables. In every call satisfying the structural induction condition, the total number of cons-cells in the values of the Static variables must

decrease. Since only calls having no Dynamic variables or satisfying the structural induction condition will be unfolded, infinite unfolding during function specialization must involve only calls satisfying the structural induction condition. Hence infinite unfolding would imply decreasing the number of cons-cells infinitely many times, which is impossible, and therefore infinite unfolding cannot happen. Notice that it is crucial that the only kind of data in Mixwell is non-circular S-expressions, for with circular S-expressions there would be no bound on the "decrease" of cons-cells.

This technique could of course be extended to cover indirect recursive calls, *i.e.*, recursive call chains involving more than one function call.

In summary, a function call will be made residual by this subphase of the preprocessing unless

- it is a call to a function with only Static variables, or
- it is a direct recursive call satisfying the structural induction condition.

In these two cases it will be made eliminable.

4.1.2 Duplication Risk Analysis

This second subphase of the call annotation phase analyses the annotations produced by the first subphase to see whether there is a call duplication risk. If so, more calls are made residual until no call duplication risk remains.

The core of this subphase is a *duplication risk analysis* that checks every eliminable call in the following way. Each argument expression is checked to see whether its symbolic value (during function specialization) may be an expression containing a call as a subexpression. If there is an argument expression with this property for which the corresponding variable (in the function called by the eliminable call) appears twice or more in the same conditional branch of the called function's body, then there is a call duplication risk. Whenever such a risk is discovered, the eliminable call is made residual, and the duplication risk analysis is done over again. If no duplication risk is found in the program, the current set of annotations is accepted and is used by the function specialization phase. This is guaranteed to lead neither to infinite unfolding nor to call duplication.

Such a set of annotations will eventually be found by the algorithm for the following two reasons. First, the algorithm must terminate since there are only finitely many eliminable calls and at least one of these is made residual by every iteration. Second, if the boundary case is reached where every call (except those having only Static variables) has been made residual, no duplication can happen.

The duplication risk analysis outlined above is fairly straightforward except for the problem of deciding whether the symbolic value of an argument expression

(during the function specialization phase) may contain a call expression. This requires a *call abstract interpretation* which is a global analysis of the subject program (including annotated calls), and which is quite similar to the binding time analysis used in mix.

The call abstract interpretation is an abstraction of the symbolic computation with expressions as data values that takes place during the function specialization phase of mix. It uses the abstract data values E and C, where E is the abstract value corresponding to symbolic expressions not containing a call, and C is the abstract value corresponding to expressions that may contain a call subexpression.

The call abstract interpretation will, for every function in the subject program, compute a description of its Dynamic variables and its result. A Dynamic variable dv of a function f is described as C if there is an eliminable (unfoldable) call to f in which the argument expression corresponding to dv has abstract value C. Otherwise the variable is described as E. Static variables are always described as E since the symbolic values of Static variables must be constant expressions; these cannot contain call subexpressions.

Given an assignment of abstract values to Dynamic variables (*i.e.*, an abstract environment), it is straightforward to compute the abstract value of an expression. The only non-trivial case is that of an eliminable call expression. In this case we define (slightly conservatively) that its abstract value is C if any of its argument expressions has abstract value C, or if the called function's body expression has abstract value C regardless of the abstract values of its Dynamic variables.

Below we describe the call abstract interpretation more formally for the sake of precision and brevity.

Let p be a Mixwell program with call annotations from the first call annotation subphase, $p = (f_i\ (sv_{i\,1} \ldots sv_{im})(dv_{i\,1} \ldots dv_{in}) = e_i)_{i=1,\ldots,h}$, let

$v \in D_{call} = \{E, C\}$

be the domain of abstract values, let

$\rho \in R = DVarnames \to D_{call}$

be an environment assigning an abstract value to each Dynamic variable of a function, and let

$\pi \in \Pi = FctNames \to R \times D_{call}$

be a global environment assigning abstract argument and result values to every function in the program p. For $\pi \in \Pi$ and $f \in FctNames$ we will write

$\pi_{arg}(f)$ for let $(\rho, v_{res}) = \pi(f)$ in ρ

$\pi_{res}(f)$ for let $(\rho, v_{res}) = \pi(f)$ in v_{res}.

All the sets above are equipped with reflexive partial orderings as follows:

D_{call}: $E < C$

R: $\rho_1 \le \rho_2$ iff $\forall\, dv \in DVarnames\ .\ \rho_1(dv) \le \rho_2(dv)$

Π: $\pi_1 \le \pi_2$ iff $\forall\, f \in FctNames\ .\ \pi_{1,arg}(f) \le \pi_{2,arg}(f) \wedge \pi_{1,res}(f) \le \pi_{2,res}(f)$

We define two functions to do the call abstract interpretation using these ordered sets. The function F computes a new approximation to the final description of each function's Dynamic variables, whereas the function A computes the abstract value of an expression in a given abstract environment.

We want a final description $\pi \in \Pi$ that is consistent and has as few C's as possible. This must be the least fixed point for the simultaneous equations

$$\pi = F[\![e_f]\!](\pi_{arg}(f))\pi \quad \text{for all f}$$
$$\pi_{res}(f) = A[\![e_f]\!](\lambda\, dv{:}DVarnames_f\,.\, E)\pi \quad \text{for all f}$$

↑ initial value

Here e_f is the body of function f and $DVarnames_f$ is the set of its Dynamic variables. This fixed point exists, because for any given program p, Π is a lattice of finite height, and the functions A and F given below are monotonic in π. This fixed point can be computed by a standard algorithm.

F: Mixwell-expr × R × Π → Π

$F[\![\text{variable v}]\!]\rho\pi = \pi$
$F[\![\text{(quote S-expression)}]\!]\rho\pi = \pi$
$F[\![\text{(car e)}]\!]\rho\pi = F[\![e]\!]\rho\pi$ same for cdr, atom
$F[\![\text{(cons } e_1\ e_2\text{)}]\!]\rho\pi = F[\![e_1]\!]\rho\pi \sqcup F[\![e_2]\!]\rho\pi$ same for equal
$F[\![\text{(if } e_1\ e_2\ e_3\text{)}]\!]\rho\pi = F[\![e_1]\!]\rho\pi \sqcup F[\![e_2]\!]\rho\pi \sqcup F[\![e_3]\!]\rho\pi$
$F[\![\text{(callr f(...)(}de_1 \ldots de_n\text{))}]\!]\rho\pi = \bigsqcup \{F[\![de_j]\!]\rho\pi \mid j=1,\ldots,n\}$
$F[\![\text{(call f(...)(}de_1 \ldots de_n\text{))}]\!]\rho\pi =$
let $\pi_{new} = \bigsqcup \{F[\![de_j]\!]\rho\pi \mid j=1,\ldots,n\}$
$\rho_{new} = [\, dv_j \mapsto A[\![de_j]\!]\rho\pi \text{ for } j=1,\ldots,n\,]$
in $\pi_{new}[\, f \mapsto (\pi_{arg}(f) \sqcup \rho_{new}, \pi_{res}(f))]$
where the called function f has Dynamic variables $dv_1, \ldots, dv_n$.

A: Mixwell-expr × R × Π → D_{call}

$A[\![\text{variable v}]\!]\rho\pi = E$ if v is Static
$\rho(v)$ if v is Dynamic
$A[\![\text{(quote S-expression)}]\!]\rho\pi = E$
$A[\![\text{(car e)}]\!]\rho\pi = A[\![e]\!]\rho\pi$ same for cdr, atom
$A[\![\text{(cons } e_1\ e_2\text{)}]\!]\rho\pi = A[\![e_1]\!]\rho\pi \sqcup A[\![e_2]\!]\rho\pi$ same for equal
$A[\![\text{(if } e_1\ e_2\ e_3\text{)}]\!]\rho\pi = A[\![e_1]\!]\rho\pi \sqcup A[\![e_2]\!]\rho\pi \sqcup A[\![e_3]\!]\rho\pi$
$A[\![\text{(callr f(...)(...))}]\!]\rho\pi = C$
$A[\![\text{(call f(...)(}de_1 \ldots de_n\text{))}]\!]\rho\pi =$
let $(\rho', v_{res}) = \pi(f)$ in $v_{res} \sqcup (\bigsqcup \{A[\![de_j]\!]\rho\pi \mid j=1,\ldots,n\})$

The call abstract interpretation realized by these functions is the basis of the duplication risk analysis. The duplication risk analysis used in this second subphase of the call annotation phase does not depend on the way annotations are made in the first subphase. It can be used with any call annotations as long as they are consistent with the Static/Dynamic classification of variables. Therefore the second subphase need not be modified in case the first subphase is improved to make better call annotations or is changed for other reasons.

4.2 Call Graph Analysis and Unfolding by Postprocessing

The annotations of every call as residual or eliminable produced by the call annotation phase are used by the subsequent function specialization phase. The simplicity of the first subphase of the call annotation phase implies that there will often be more residual calls than is in principle necessary to avoid infinite unfolding, call duplication and other anomalies. Every residual call encountered during function specialization gives rise to a (possibly new) residual function, and for that reason the residual program will often contain many very simple residual functions, some consisting of just a call to another function. This impairs readability and slows down execution of the residual program somewhat.

It is the purpose of the postprocessing phase described here to reduce the number of residual functions by doing further call unfolding in the residual program.

The postprocessing phase has two stages, or subphases. The first one does an analysis of the residual program to be processed to see which functions may be unfolded and which may not. The second subphase then does the unfolding of function calls (and some further reduction of expressions made possible by unfolding) while using the information gathered by the first subphase.

The important observation behind the analysis done by the first subphase is this: infinite unfolding must involve a recursive call chain from a function f (possibly through several others) back to itself, *e.g.*, f → g → h → f in

```
...
f (...)  =  ... (call g ...) ...
g (...)  =  ... (call h ...) ...
h (...)  =  ... (call f ...) ...
...
```

By suspending all calls to at least one function in such a recursive chain, infinite unfolding of the chain will be prevented. The idea now is to select one function (to be called a *cutpoint*) in each recursive call chain, and then suspend calls to that function.

The two subphases of the postprocessing phase will be described in Sections 4.2.1 and 4.2.2 below. But first we give definitions of the concepts of call graph and recursive call chain. The *call graph* of a Mixwell program r is a directed multigraph that has the program's functions as nodes, and has an edge from function f to function g for each call to g in the body of f. A *recursive call chain* is a cycle in the call graph, that is, a non-empty sequence of edges (*i.e.*, calls) f → ... → f such that the first and last nodes are the same.

4.2.1 Call Graph Analysis

The call graph analysis of a Mixwell program r works by traversing the call graph of r to find its recursive call chains and then select a cutpoint for each of these.

The call graph analysis does a recursive depth-first traversal of the graph, starting with the goal function, and is an instance of a general scheme for depth-first traversal of directed graphs [Aho, Hopcroft, Ullman 1974].

The algorithm maintains a marking of the functions that have already been visited and keeps account of the current path from the goal function to the function currently being visited (inclusive). Furthermore, the algorithm records the cutpoints for the recursive call chains found so far. A *visit* to a function f consists of the actions

- mark f visited, then
- extend the current path by f, then
- for every function g called by f,
  if g is on the current path, then
    a recursive call chain has been discovered: make g a cutpoint
  else if g is not already visited, then visit g,
- remove f from the end of the current path.

When the initial visit of the goal function is finished, (at least) one cutpoint has been found for every recursive call chain in the program. Note that the current path will never contain a recursive call chain, and that every function on the current path has already been marked visited.

Note that the algorithm visits each function once and does one traversal of its body. So provided the operations of marking and mark testing and the path manipulation operations each take constant time, the algorithm will run in time linear in the size of the program being analyzed.

4.2.2 Unfolding

The second subphase of the postprocessing phase traverses the residual program produced by the function specialization phase and unfolds calls to small functions while avoiding infinite unfolding, call duplication, and code duplication. The traversal starts with the goal function and consists in a symbolic evaluation (computing with expressions as values) in the same way as does the function specialization phase.

Again symbolic evaluation of expressions other than calls is straightforward, so we will discuss only the treatment of calls.

Consider a call (call g $e_1 \ldots e_n$) to a function g. First the reduced versions $e_1^* \ldots e_n^*$ of the argument expressions $e_1 \ldots e_n$ are computed; the argument expressions may themselves contain calls that should be unfolded. Either the call to g will be unfolded, *i.e.*, replaced by the body e of g with the symbolic expressions $e_1^* \ldots e_n^*$ substituted for occurrences of the corresponding variables $v_1 \ldots v_n$; or it will be left as it is, with $e_1 \ldots e_n$ replaced by $e_1^* \ldots e_n^*$.

Which of these two actions to take will be decided as follows. If g has been chosen as a cutpoint by the preceding call graph analysis, then the call will not be unfolded. If g has not been chosen as a cutpoint, it will be checked whether there is a risk of call duplication or code duplication when unfolding the call. The check works like this: if the reduced form e_j^* of an argument expression in the call to g contains a call itself, or is a sizeable expression, then it is checked whether the corresponding variable v_j appears more than once in any branch of g's body expression e. If so, there is a risk of call duplication or code duplication for variable v_j. If there is such a risk for any of g's variables, then the call will not be unfolded; otherwise it will.

The transformations done by the unfolding phase are not fully semantics preserving. The errors are however on the "safe" side: a postprocessed program may terminate more often than the one input to the postprocessing. This is due to the call-by-name nature of unfolding.

4.3 Limitations of the Method

In this section we shall look at a case where the partial evaluator mix will not work with our call unfolding method (or any other for that matter).

Consider the example program

```
g (x z)   = (if (null z) then x
             else (call g (cons 'A x) (cdr z)))
```

and assume x = '() is known and z is unknown. We will see that in the framework of

mix, no satisfactory call annotation is possible.

For if the call is made eliminable then clearly infinite unfolding will result. If on the other hand the call is made residual, an infinity of specialized versions of g will be produced, each specialized to a value of the Static variable x:

```
g()    (z) = (if (null z) then '() else (call g(A) (cdr z)))
g(A)   (z) = (if (null z) then '(A) else (call g(A A) (cdr z)))
g(A A) (z) = (if (null z) then '(A A) else (call g(A A A) (cdr z)))
...
```

It is a basic principle of mix to specialize each function to the possible values of its Static variables. For this to work, the number of possible values for each function must be finite, and in fact mix works well on programs satisfying this requirement. (The class of such programs has been called "analyzer programs with finitely defined memory" by Bulyonkov [Bulyonkov 1984, 1985].)

From the specialization point of view, the classification of x as Static by the binding time analysis is simply wrong. The set of possible values of x is not statically determined: it depends on the value of the Dynamic variable, and there is no statically determined bound on the size of the set of x's possible values.

These problems and related issues have recently been treated by Jones in a thorough reconsideration of the concept of binding time analysis [Jones 1988].

5. RESULTS AND ASSESSMENT

In this section we give some results from the use of our call unfolding method in the partial evaluator mix.

5.1 Simple Annotations Based on Inductive Variables

Partial evaluation using simple call annotations based on inductive variables gives residual programs that have a reasonable structure and a not too overwhelming size. These residual programs in general have very many small functions, which makes them quite unreadable to humans. On the other hand, they are usually almost as fast as and of approximately the same size as residual programs produced from subject programs that were carefully call annotated by hand. In particular, the method works well on the partial evaluator itself, and so it is fully automatic and self-applicable.

The method is very well suited for application to interpreters and other programs that "work by recursive descent": they decompose part of their input in the course of recursion or iteration. Such programs will often have several places

where the structural induction condition is satisfied, and this will result in a fair number of calls being made eliminable. Satisfactory results for precisely this class of programs are important, because compilation by partial evaluation of an interpreter is a very interesting application of partial evaluation.

A great advantage of using call annotations that are generated automatically is that in contrast to human-made ones, they are *guaranteed* not to give trivial residual programs, infinite unfolding, or call duplication.

The call annotation algorithms are quite fast, in particular the first subphase which does only a local analysis of the program. The second subphase (which does one or more global analyses) is slower than the first, but still spends only approximately 5 cpu seconds on a 500 line Mixwell program.

As to future developments, it is tempting to improve on the first subphase so that it would recognize more situations where calls can safely be unfolded. This could be done without affecting the second subphase at all, as the second subphase does not depend on the way the call annotations are made. To make better call annotations, the first subphase would have to take a more global view of the program than it does presently. Information about the structure of the call graph of the subject program should be relevant, and so should information about the behaviour of Static variables along recursive call chains in the graph. This would allow to take into account also indirect recursive calls satisfying the structural induction condition. Call annotations that are of the same quality as those produced by an experienced user are probably very hard to make automatically.

5.2 Effects of Unfolding by Postprocessing

We briefly illustrate the effect of postprocessing on mix-produced residual programs. Below cocom is the compiler generator produced by mix, comp is a compiler for a tiny imperative language (produced by cocom), and target is a target program (produced by comp) for a program to compute x^y, *i.e.*, x raised to the y'th power.

First we give examples of the size reduction achieved by postprocessing. The number of lines is for prettyprinted Lisp listings. As can be seen, the effect on the size of the programs is considerable. Also the readability of the programs is improved, mostly because the plethora of calls to functions with non-telling names are replaced by the called functions' bodies substituted in-line.

Program		Before unfolding	After unfolding
target	No. of functions	37	6
	No. of lines	112	36
	No. of cons cells	474	253
comp	No. of functions	148	24
	No. of lines	600	303
	No. of cons cells	3387	2426
cocom	No. of functions	400	49
	No. of lines	1904	1062
	No. of cons cells	11351	8853

Figure 5.1: Size Improvement by Postprocessing

We have also found that reasonable speed-ups (running-time reduced by between 5 and 50 per cent) have been achieved by applying the postprocessing. For some larger programs, such as the compiler generator cocom, the speed-up achieved by postprocessing is negligible, and the main reason for applying it is the desire to get programs that are readable by humans.

The postprocessing phase itself is tolerably fast, taking 11 cpu seconds for processing the compiler generator cocom mentioned above. However, it is a drawback that, in contrast to the call annotation phase, it cannot be optimized by partial evaluation of the partial evaluator itself. For this reason it would be desirable to obviate the need for a postprocessing phase altogether, with the implication that a more sophisticated way to find call annotations would be needed. Such an improvement would concern the first subphase of the call annotation phase only, and is discussed above at the end of Section 5.1.

6. SUMMARY

We have discussed the problem of call unfolding in a partial evaluator for (first order) pure Lisp, and we presented an automatic two-phase call unfolding method. Some results from its use were reported and discussed.

We concluded that the method works well for a large class of programs, notably interpreter-like programs working by recursive descent. This class includes the partial evaluator mix, and this is crucial for self-applicability of the partial evaluator. For other programs, the method will fail due to problems with infinite specialization.

For programs on which the method works well, the second (*i.e.* postprocessing) phase mainly contributes by improving the readability of the resulting programs.

7. ACKNOWLEDGEMENTS

I am most grateful towards Neil D. Jones and Harald Søndergaard for introducing me to partial evaluation and for our exciting collaboration on the development of the self-applicable partial evaluator mix.

Thanks also go to Niels Carsten Kehler Holst and Olivier Danvy for suggesting, among other things, improvements to the call graph analysis, and to Torben Mogensen for discussions on call unfolding strategies.

8. REFERENCES

[Aho, Hopcroft, Ullman 1974]
A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Beckman *et al.* 1976]
L. Beckman [*et al.*]. A partial evaluator, and its use as a programming tool. *Artificial Intelligence* 7, 4 (1976) 319-357.

[Bulyonkov 1984]
M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica* **21** (1984) 473-484.

[Bulyonkov 1985]
M. A. Bulyonkov. Mixed computations for programs over finitely defined memory with strict partitioning. *Soviet Mathematics Doklady* **32**, 3 (1985) 807-811.

[Ershov 1978]
A. P. Ershov. On the essence of compilation. In E.J. Neuhold (ed.): *Formal Description of Programming Concepts*, 391-420. North-Holland, 1978.

[Ershov 1982]
A. P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science* **18** (1982) 41-67.

[Fuller, Abramsky 1987]
D. A. Fuller and S. Abramsky. Mixed computation of Prolog programs. In D. Bjørner, A. P. Ershov, and N. D. Jones (eds.): *Workshop Compendium. Workshop on Partial Evaluation and Mixed Computation, Gl. Avernæs, Denmark, October 1987*, 83-101. Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, 1987.

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

[Jones 1988]
N. D. Jones. Automatic program specialization: a re-examination from basic principles. In D. Bjørner, A. P. Ershov, and N. D. Jones (eds.): *Workshop on Partial Evaluation and Mixed Computation, Gl. Avernæs, Denmark, October 1987*. North-Holland, 1988. (This volume).

[Jones, Sestoft, Søndergaard 1985]
N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud (ed.): *Rewriting Techniques and Applications. Lecture Notes in Computer Science* **202** (1985) 124-140. Springer-Verlag.

[Jones, Sestoft, Søndergaard 1987]
N. D. Jones, P. Sestoft, and H. Søndergaard. MIX: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. DIKU Report **87/8** (June 1987). DIKU, University of Copenhagen, Denmark.

[Lombardi 1967]
L. A. Lombardi. Incremental computation. In F. L. Alt and M. Rubinoff (eds.): *Advances in Computers* **8** (1967) 247-333. Academic Press.

[Sestoft 1986]
P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N.D. Jones (eds.): *Programs as Data Objects. Lecture Notes in Computer Science* **217** (1986) 236-256. Springer-Verlag.

[Turchin 1979]
V. F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices* **14**, 2 (1979) 46-54.

PARTIAL EVALUATION AND MIXED COMPUTATION

Proceedings of the IFIP TC2 Workshop on
Partial Evaluation and Mixed Computation
Gammel Avernæs, Denmark, 18–24 October, 1987

edited by

DINES BJØRNER
Technical University of Denmark
Lyngby, Denmark

ANDREI P. ERSHOV
USSR Academy of Sciences
Novosibirsk, USSR

NEIL D. JONES
University of Copenhagen
Copenhagen, Denmark

1988

NORTH-HOLLAND
AMSTERDAM · NEW YORK · OXFORD · TOKYO