Towards Efficient Partial Evaluation

Karoline Malmkjær [*]
Department of Computing and Information Sciences
Kansas State University[†]
(karoline@cis.ksu.edu)

Abstract

In general, a partial evaluator needs to keep track of the tasks that have already been completed or initiated, so that it can recognize when to stop unfolding. In the MIX-style polyvariant specialization algorithm, this is accomplished by a global log. This is a very general technique, so it is not surprising that the algorithm is not particularly efficient. In many special cases a simpler technique would suffice.

In this paper, we identify some classes of such special cases by considering the purpose of the global log. We outline how a partial evaluator can take advantage of these special cases and we propose analyses to detect them automatically. We discuss examples to illustrate the effect on specialization and to demonstrate that we can even obtain better residual programs.

The work presented here is still in its early stages, and we do not have a full system incorporating the proposed improvements.

1 The bookkeeping of polyvariant specialization

The underlying principle of the self-applicable MIX-type partial-evaluation algorithms [JSS89] is related to Jones and Mycroft's minimal function-graphs [JM86]. Each time a procedure is called, the call is compared to a list of already processed calls (sometimes called "seen-before" [BD91] or "déjà-vu" [JGS93] — in MIX it is split into "Pending" and "Out" [Ses86]). If the current call is found in this log of calls, it is not necessary to process the procedure, and we can simply refer to the result. If it is not found, a reference is created and the call and the reference are entered in the log. Then the procedure is processed and the result is recorded under the reference. In partial evaluation, the reference is typically a freshly created name for the residual procedure. In abstract interpretation, the reference might for example be a non-terminal in a grammar [JM86].

This log is the central administrative structure in the algorithm. It ensures a smooth handling of branches, loops, and recursion in one mechanism and is the key tool for making the specialization terminate in any non-trivial case [Ses86].

[*]Supported by DIKU, Department of Computer Science, University of Copenhagen.
[†]Manhattan, KS 66506, USA

In general, the program points that we remember in the log do not have to be procedures. In a simple imperative language, labels are a possibility, and Similix uses (essentially) dynamic conditionals. In a partial evaluator, the selected program points are called *specialization points.* During partial evaluation, the specialization points are *residualized* (a specialized instance is created), and the remaining program points are *unfolded* (the specialized text is inlined).

If the domain of values is finite or has only finite chains (and the analysis is monotone), this technique ensures termination, since only finitely many different calls can be logged for each procedure. In the case of partial evaluation, the domain of values has infinite chains and termination cannot be guaranteed. The partial evaluator may loop because it attempts *infinite residualization*[1].

The log, however, also entails a certain administrative overhead. Clearly a comparison between a procedure call (that is, a procedure name and a list of arguments) and a log of previous calls can become quite expensive. When the log is structured as a list, for example, we have to traverse the entire list if the call is not in the log. The log will typically be at least linear in the size of the static data, though this depends on the number of static operations. Thus it would be an advantage to find a faster way of managing the treatment of procedure calls. In a partial evaluator, however, it is not clear whether this is possible in general without risking non-termination or getting less specialized results. This is because the partial evaluator is expected to follow all possible evaluation paths, that is, it must process all calls.

Administratively, some improvement can be obtained by changing the simple list-structure of the log. One could keep a separate log of static values for each of the specialization points/procedure names (sorting according to the specialization points) and this "mini-log" could possibly also be organized more efficiently (hash-coding springs to mind). This is essentially a question of encoding, *i.e.*, of doing the same administration in a more efficient way.

Here we investigate a different goal: discovering fundamental properties that allow us to *reduce* the raw amount of necessary administration. We want to find special cases in which faster specialization is possible because the log can be ignored or its management simplified.

The rest of this paper is organized as follows: In section 2, we define a property of source programs that can be determined by a simple analysis and we outline how a partial eval-

[1]The other potential cause of non-termination in a partial evaluator is *infinite unfolding.* This can always be changed into infinite residualization, however, by reclassifying the unfolded calls to be residualized.

uator could be modified to take advantage of this property. In section 3, we define a somewhat more elusive property of source programs, which can be used to simplify the annotations of a binding-time analysis based partial evaluator, but does not require any changes to the actual specializer. In both cases, we give an analysis for detecting the properties. Section 5 describes related work, and section 6 concludes and outlines future work.

We use Similix [BD91, Bon91a] as our main example of a partial evaluator, but most of our reasoning is generally applicable to other partial evaluators.

2 A possible improvement: constrained static values

To obtain improvements we look for special cases that are reasonably common and in which log administration can be cut down.

A good candidate is the case where a variable is bound to a static value that is always a subpart of the static input. This is for example the case if the source program is *compositional* (in the sense of denotational semantics) in that argument: the result on the whole argument is a composition of the results on the parts [Sch86].

If the possible values of a parameter are constrained in this way, we know that the parameter can be bound to only finitely many different values and furthermore that the set of all these values is known from the beginning of the specialization.

To argue that this case is common, let us give a couple of examples.

Consider the case where the static datum is a proper list and the source program only takes heads or tails of this list and never performs, for example, cons or append operations. Then we can conclude that any static value must be either an element or a suffix of the list.

A more interesting example is the situation where the source program is an interpreter and the static datum is a program, that is, an abstract syntax tree. The procedures of the interpreter are typically called with arguments that are sub-structures of the static program. It should be possible to detect this automatically.

An interpreter handling higher-order values often represents them as a program point (or a piece of abstract syntax) and an environment. If we allow partially-static structures, such values are constrained in the first component when we specialize the interpreter: the first component is always a sub-structure of the static input.

Note that the partial evaluator itself takes a syntax tree as input and is constrained in this argument. Thus this technique should also improve self-application.

To get an idea of how much could be saved by avoiding lookups in the log, we note that after generation of cogen (*i.e.,* after specializing the partial evaluator with respect to itself) in Similix, the log has 272 entries. In other words, during specialization, the specializer looks through the entire log and fails to find the entry 272 times. In comparison, there are only 207 successful lookups in the log during specialization, or about 43% of the total number of lookups[2].

For a more modest example, the specialization of the MP-interpreter (distributed with the Similix package, see figure 2) with respect to the program in MP computing the power function gives 4 entries in the log, corresponding to the number of while-loops and conditional statements in the power program. There are 3 successful lookups in the log, corresponding to just the number of while-loops in the power program.

Generating a compiler from the MP-interpreter gives 38 entries in the log and 26 successful lookups.

When compiling larger programs, there are typically many, similar, static values, corresponding to similar expressions in the static program[3]. This means that the lookup in the log can be quite slow in determining whether an entry matches or not.

It is not surprising that most of the calls are not in the log; this can be deduced from the call structure of the source programs in question. Most of the calls have an argument that is a proper subpart of a previous argument.

If we can determine that the possible static values are constrained to be subparts of the static input, then we can structure the log based on the static input and improve the lookup.

2.1 How to take advantage of constrained values

To take advantage of a constrained value, we could create a table corresponding to the sub-structures of the value and annotate the structure with the indices into the table. Whenever a particular sub-structure is encountered, a simple constant-time lookup will be sufficient to determine if it has been encountered before. For example, in the case of a list, the table could be a vector, indexed by the position of the tail in the list. In the case of a syntax tree, one might use labels on the nodes that could also function as offsets into a table.

Alternately, one could let the log have the same structure as the static datum. For each computation leading to a value that is a sub-structure of the static structure, one would also compute the corresponding sub-structure of the log (or the log would simply be an annotation on the data-structure). When a procedure is called with a particular value, the log is immediately accessible.

Probably the most common case will be that the set of possible values can be determined to be constrained in this way for some of the static variables but not for all. In this case the entries in the log have to be "mini-logs", corresponding to the unconstrained (or not detectably constrained) static values.

In languages like Scheme, though, some improvement could be obtained just by replacing the equality test on the constrained value by an `eq?` test for the simple kinds of constraints[4]. With this modification, we would still have to run through the entries in the log, but each comparison would be faster. Note that for self-application, this requires that `eq?` is handled properly by the partial evaluator. This

[2]These experiments were conducted with the distribution version of Similix-4.

[3]Consider, for example, how often the eval-procedure in a Scheme interpreter will be called with an expression of the form `(if (null? l) '() <some-expr>)`.

[4]It would not be sufficient for handling the more subtly constrained values, such as the environment when specializing a procedure, since they are constructed at different places, but in the same way. This might be handled using hash cons-ing [Got74], though it is not clear whether this would mesh well with the partial evaluator. Since we do not yet have an analysis for detecting these cases, this is not a practical concern now, though.

would be feasible in Similix, since the administration is handled in the separately defined primitives.

In terms of self-application, we note that the specialization points are static to the partial evaluator at self-application time. The static data of the program, however, are dynamic at self-application time. So the best arrangement, if we have a partial evaluator with partially-static structures, is to have a table of specialization points and at each entry have a table of constrained static values corresponding to that specialization point. Then we can hope for the access to the specialization points to be reduced at self-application time. The administration of the argument part of the log, however, will be part of the generating extension (the specialized specializer), where it will be handled according to the new, faster, strategy. This kind of design seem to be the next logical step in obtaining realistic compilers by partial evaluation.

2.2 Example: the MP interpreter

```
P in Program        C in Command
B in Block          V in Variable
E in Expression     Cst in Constant

P = (program (pars V1*) (vars V2*) B)
B = (C*)
C = (:= V E)
  | (if E B1 B2)     # first branch iff exp not ()
  | (while E B)      # loop iff Exp not ()
E = Cst
  | V
  | (car E)
  | (cdr E)
  | (cons E1 E2)
  | (atom E)         # () iff not atom
  | (equal E1 E2)    # () iff not equal
```

Figure 1: Syntax of MP programs, as given in the Similix package.

To illustrate the effect of the proposed technique on compilers generated by partial evaluation, we consider the MP example from the Similix package [BD91, Bon91b]. Similix is a self-applicable partial evaluator for Scheme [CR91]. MP is a small imperative language with while-loops designed for experiments with a partial evaluator. Its BNF is given in figure 1. In figure 2 we show the MP-interpreter from the Similix package [Bon91b]. Note that this interpreter is not compositional, as it interprets a while-loop by recursion over the same command. As can be seen, the MP-interpreter uses only destructors and predicates on the source (MP) program. This is also determined by the proposed analysis. This means that the optimization would apply, if the Similix specializer was modified to take advantage of this constraint.

The advantage would also propagate to the compilers generated by self-application.

Currently the MP-compiler generated by Similix operates by maintaining a log similar to the one of Similix itself. When compiling a while-loop, the compiler generates a specialized version of the (interpreter) procedure that handles commands. This will correspond to a compiled ver-

```
(define (run P value*)
  (let* ([V2* (P->V2* P)]
         [env (init-environment (P->V1* P) V2*)])
    (init-store! value* (length V2*))
    (evalBlock (P->B P) env)))

(define (evalBlock B env)
  (if (emptyBlock? B)
      "Finished block"
      (evalCommands
       (headBlock B) (tailBlock B) env)))

(define (evalCommands C B env)
  (if (emptyBlock? B)
      (evalCommand C env)
      (begin (evalCommand C env)
             (evalCommands
              (headBlock B) (tailBlock B) env))))

(define (evalCommand C env)
  (cond
    [(isAssignment? C)
     (update-store!
      (lookup-env (C-Assignment->V C) env)
      (evalExpression (C-Assignment->E C) env))]
    [(isConditional? C)
     (if (is-true? (evalExpression
                    (C-Conditional->E C) env))
         (evalBlock (C-Conditional->B1 C) env)
         (evalBlock (C-Conditional->B2 C) env))]
    [(isWhile? C)
     (if (is-true? (evalExpression
                    (C-While->E C) env))
         (begin (evalBlock (C-While->B C) env)
                (evalCommand C env))
         "Finished loop")]
    [else "Error - unknown command"]))

(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"]))
```

Figure 2: The MP interpreter from the Similix package.

sion of the body of the loop. Before actually generating it, though, it first checks whether a call to this procedure with the same command is already in the log (which we know, at the meta-level, it is not, unless there is another while-loop in the source program with the same text).

The procedure in the interpreter contains a recursive call with the same static argument (corresponding to the body of the loop – this implements the looping). In the compiler, this appears as another check through the log to see if the call is already there (which we now, at the meta-level, know that it is).

With the proposed technique, these two searches through the log would both be replaced by an `eq?` test on the text of the while-loop or by a lookup in a table (depending on how exactly we implement the optimization in the specializer). In the first lookup, a newly generated name of the residual procedure would be added to the log. The second lookup would serve to find the name of the residual procedure.

2.3 An analysis detecting constrained values

To determine if a variable in a source program will always be bound to a constrained static value, we can use a simple abstract interpretation of the binding-time analyzed program (since we don't need any results for the dynamic variables).

We use the usual two-point domain **2** with bottom meaning "constrained" and top meaning "possibly not constrained". The intended meaning of "constrained" is that the possible concrete values are smaller than the static input in some well-founded ordering. The original static input is of course constrained. In a partial evaluator such as Similix, this will not change, since the program is automatically rewritten so that the goal function is never called.

Any primitive operation that destructs data, such as car and cdr, preserves constraint. Any primitive operation that constructs data, such as cons or append, makes the value unconstrained[5]. Predicates produce a boolean value, which is inherently constrained, but have no effect on the value they test, so it is not necessary to worry about any "feedback" effect. Note that this analysis will give appropriate results even if the input is circular. Circular data may be used in a constrained way, since the circularity will bring us back to somewhere we have been before and this place will be marked correctly according to whether it has been processed or not.

In figure 3 we outline such an analysis for a first-order Scheme-like language with primitive operations cons, car and cdr. For simplicity in the presentation, all arguments are handled, rather than introducing the distinction between static and dynamic. Since a binding-time analysis guarantees that variables classified as static will not depend on dynamic variables, this distinction can safely be added to the constraint analysis, for example by ignoring all dynamic variables or binding them to "constrained" straight away.

The analysis is a quite straightforward minimal function-graph analysis. It has a global fixed-point over a cache (or log, but we already use this term with other connotations here) that contains entries for each procedure in the program being analyzed. Each entry contains the values of the parameters of the procedure, determined as the least upper bound of the arguments at the possible call-sites, and the value of the result of the procedure, determined by analyzing it in the symbolic environment where the parameters are bound to the values.

[5] If a constructor only re-constructs something previously encountered, the value is of course still constrained, but detecting such cases would require a considerably more sophisticated analysis.

This analysis is of course rather straightforward and many refinements could be imagined. Consider for example the case of a list that is referenced using an index. The index variable is typically initialized to zero and incremented with add1 until the length of the list. So if the list is static input, the possible values of the index are bounded by the initial value and the length. This means that the set of possible values is essentially determined at the beginning of the specialization, but that would not be detected by the analysis suggested above.

A more complex analysis for detecting such cases could for example work by using algebraic properties. It could construct a kind of restricted algebra or "algebraic data type" of the operations on each value. In the case of the index, this algebra would contain the constant zero, the operation add1 and the predicate checking equality to the length of the list. If the analysis is able to establish that the length is static and fixed, it can conclude that the index is constrained.

For algebraic data types that are not "well-founded", such as integers with zero?, sub1, and add1, we would need an even more complex analysis to determine if the source program uses them in a constrained way. Clearly it would be preferable to have more explicit user declarations of the algebras, in the line of Similix's ".adt-files". Coupled with some sort of type discipline, this would enable detection of existing constraints in many more cases.

If we continue expanding the analysis in this way, it will eventually turn into a kind of termination analysis (see for example [Hol91]), since it essentially tries to determine whether the set of all values computed during specialization is finite. Thus it is clear that we cannot find all variables that are constrained in the most general sense. Furthermore, we are only interested in notions of constraint that allow us to determine the set of values and construct a corresponding table reasonably fast compared to the specialization time.

2.4 The effect of simplification on residual programs

It is worth noticing that a partial evaluator using this technique will not produce exactly the same results as the usual polyvariant partial evaluator. The main difference is that the usual list-structured log causes sharing when the same value is encountered at two different places, for example if two sub-expressions of a static input program to an interpreter happen to be identical. The resulting effect resembles "common-subexpression elimination".

With the proposed technique, this will not happen without additional analysis. In the case of compilers, for example, this is not surprising: we don't expect a usual compiler to notice that we have written the same expression twice in a program and only compile one version. To detect this, the compiler needs explicit common-subexpression elimination tools, which are expensive. And in fact the usual partial evaluator only obtains this effect by comparing the potentially common expressions against each other, which is also expensive.

2.5 Another kind of constraint

Even if the value of a static variable is not the subpart of the static input, there is an obvious case where it is still

Syntax:

$$\begin{array}{lcl}
\mathrm{P} & ::= & (\mathrm{define}\ (\mathrm{F}_0\ \mathrm{I}_1\ \ldots\ \mathrm{I}_{n_0})\ \mathrm{E}_0)\ \ldots\ (\mathrm{define}\ (\mathrm{F}_m\ \mathrm{I}_1\ \ldots\ \mathrm{I}_{n_m})\ \mathrm{E}_m) \\
\mathrm{E} & ::= & \mathrm{C}\ |\ \mathrm{I}\ |\ (\mathrm{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)\ |\ (\mathrm{cons}\ \mathrm{E}_1\ \mathrm{E}_2)\ |\ (\mathrm{car}\ \mathrm{E})\ |\ (\mathrm{cdr}\ \mathrm{E})\ |\ (\mathrm{F}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n)
\end{array}$$

Domains:

$$\begin{array}{rcl}
v \in \mathit{Val} & = & \mathbf{2} \\
\rho \in \mathit{Env} & = & \mathrm{Id} \to \mathit{Val} \\
\varphi \in \mathit{Cache} & = & \mathrm{Id} \to \mathit{Val}^* \times \mathit{Val}
\end{array}$$

Functions:

$$\begin{array}{l}
\mathcal{A}_{\mathrm{P}} : \mathrm{Pgm} \to \mathit{Val}^* \to \mathit{Cache} \\
\mathcal{A}_{\mathrm{P}}\ \llbracket(\mathrm{define}\ (\mathrm{F}_0\ \mathrm{I}_1\ \ldots\ \mathrm{I}_{n_0})\ \mathrm{E}_0)\ \ldots\ (\mathrm{define}\ (\mathrm{F}_m\ \mathrm{I}_1\ \ldots\ \mathrm{I}_{n_m})\ \mathrm{E}_m)\rrbracket\, v^* = \\
\mathit{fix}\,\lambda\,\varphi.\ \mathit{let}\ (v_0,\varphi_0) = \mathcal{A}\ \llbracket \mathrm{E}_0\rrbracket(\mathit{mk\text{-}env}\ [\mathrm{I}_0,\ldots,\mathrm{I}_{n_0}]\ (\mathit{lookup\text{-}args}\ \mathrm{F}_0\ (\mathit{update\text{-}args}\ \mathrm{F}_0\ v^*\ \varphi))) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\ldots\ \mathit{let}\ (v_m,\varphi_m) = \mathcal{A}\ \llbracket \mathrm{E}_m\rrbracket(\mathit{mk\text{-}env}\ [\mathrm{I}_0,\ldots,\mathrm{I}_{n_m}]\ (\mathit{lookup\text{-}args}\ \mathrm{F}_m\varphi))\varphi \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ (\mathit{update\text{-}res}\ F_m\ v_m\varphi_m)\ldots) \\
\qquad \mathit{in}\ (\mathit{update\text{-}res}\ F_0\ v_0\ \varphi_0)
\end{array}$$

$$\mathcal{A} : \mathrm{Expr} \to \mathit{Env} \to \mathit{Cache} \to \mathit{Val} \times \mathit{Cache}$$

$$\begin{array}{rcl}
\mathcal{A}\ \llbracket\mathrm{C}\rrbracket\,\rho\,\varphi & = & (\bot,\varphi) \\
\mathcal{A}\ \llbracket\mathrm{I}\rrbracket\,\rho\,\varphi & = & (\mathit{lookup}\ \mathrm{I}\,\rho,\varphi) \\
\mathcal{A}\ \llbracket(\mathrm{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)\rrbracket\,\rho\,\varphi & = & \mathit{let}\ (v_0,\varphi_0) = \mathcal{A}\ \llbracket\mathrm{E}_0\rrbracket\,\rho\,\varphi \\
& & \mathit{in}\ (\mathcal{A}\ \llbracket\mathrm{E}_1\rrbracket\,\rho\,\varphi_0)\sqcup(\mathcal{A}\ \llbracket\mathrm{E}_2\rrbracket\,\rho\,\varphi_0) \\
\mathcal{A}\ \llbracket(\mathrm{cons}\ \mathrm{E}_1\ \mathrm{E}_2)\rrbracket\,\rho\,\varphi & = & \mathit{let}\ (v_1,\varphi_1) = \mathcal{A}\ \llbracket\mathrm{E}_1\rrbracket\,\rho\,\varphi \\
& & \mathit{in}\ \ \mathit{let}\ (v_2,\varphi_2) = \mathcal{A}\ \llbracket\mathrm{E}_2\rrbracket\,\rho\,\varphi_1 \\
& & \quad \mathit{in}\ (\top,\varphi_2) \\
\mathcal{A}\ \llbracket(\mathrm{car}\ \mathrm{E})\rrbracket\,\rho\,\varphi & = & \mathcal{A}\ \llbracket\mathrm{E}\rrbracket\,\rho\,\varphi \\
\mathcal{A}\ \llbracket(\mathrm{cdr}\ \mathrm{E})\rrbracket\,\rho\,\varphi & = & \mathcal{A}\ \llbracket\mathrm{E}\rrbracket\,\rho\,\varphi \\
\mathcal{A}\ \llbracket(\mathrm{F}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n)\rrbracket\,\rho\,\varphi & = & \mathit{let}\ (v_1,\varphi_1) = \mathcal{A}\ \llbracket\mathrm{E}_1\rrbracket\,\rho\,\varphi \\
& & \mathit{in}\ \ldots\ \mathit{let}\ (v_n,\varphi_n) = \mathcal{A}\ \llbracket\mathrm{E}_n\rrbracket\,\rho\,\varphi_{n-1} \\
& & \qquad \mathit{in}\ (\mathit{lookup\text{-}res}\ \mathrm{F}\,\varphi_n,\ \mathit{update\text{-}args}\ \mathrm{F}\,[v_1,\ldots,v_n]\,\varphi_n)
\end{array}$$

where

$$\begin{array}{l}
\mathit{lookup\text{-}args} : \mathrm{Id} \to \mathit{Cache} \to \mathit{Val}^* \\
\mathit{lookup\text{-}args} = \lambda\,F\,\varphi.\ (\varphi\,F) \downarrow 1 \\
\mathit{lookup\text{-}res} : \mathrm{Id} \to \mathit{Cache} \to \mathit{Val} \\
\mathit{lookup\text{-}res} = \lambda\,F\,\varphi.\ (\varphi\,F) \downarrow 2 \\
\mathit{lookup} : \mathrm{Id} \to \mathit{Env} \to \mathit{Val} \\
\mathit{lookup} = \lambda\,i\,\rho.\ (\rho\,i) \\
\mathit{mk\text{-}env} : \mathrm{Id}^* \to \mathit{Val}^* \to \mathit{Env} \\
\mathit{mk\text{-}env} = \lambda\,[\mathrm{I}_0,\ldots,\mathrm{I}_n]\,[v_0,\ldots,v_n].\ \lambda\,i.\ i = \mathrm{I}_j \to v_j \parallel \top \\
\mathit{update\text{-}args} : \mathrm{Id} \to \mathit{Val}^* \to \mathit{Cache} \to \mathit{Cache} \\
\mathit{update\text{-}args} = \lambda\,F\,v^*\,\varphi.\ \mathit{let}\ (a_F^*, r_F) = \varphi\,F \\
\qquad\qquad\qquad\qquad\qquad \mathit{in}\ \ \mathit{let}\ w^* = \mathit{map} \sqcup (\mathit{zip}\ v^*\ a_F^*) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ \lambda\,F'.\ F' = F \to (w^*, r_F) \parallel (\varphi\,F') \\
\mathit{update\text{-}res} : \mathrm{Id} \to \mathit{Val} \to \mathit{Cache} \to \mathit{Cache} \\
\mathit{update\text{-}res} = \lambda\,F\,v\,\varphi.\ \mathit{let}\ (a_F^*, r_F) = \varphi\,F \\
\qquad\qquad\qquad\qquad \mathit{in}\ \lambda\,F'.\ F' = F \to (a_F^*, v \sqcup r_F) \parallel (\varphi\,F')
\end{array}$$

Figure 3: Analysis detecting constrained input

constrained: if it belongs to a finite domain. It is interesting for partial evaluation to take advantage of this in the cases where the finite domain is small[6].

The most typical example of a finite domain (and particularly interesting because it is the only one where most partial evaluators automatically take advantage of the finiteness) is the domain of booleans. The result of a predicate is either true or false[7]. In the case where a control decision has to be made based on the actual value of a predicate (a conditional expression), the partial evaluator will simply assume that the value is true and complete the processing, then assume that it is false, complete the processing, and finally combine the two results.

The result is a residual expression that first has the choice between the finite set of values and then the specialized code for each choice.

Clearly, a similar technique can be applied to any other domain that is known in advance to be small and finite. This includes values that belong to an infinite domain, but only finitely many can occur during specialization with respect to given static data (for example, when specializing an interpreter, the procedure names of the source program), that is, they are constrained in the way outlined above.

This suggests another way for the specializer to take advantage of the results of the constraint analysis. For example, any dynamic equality test between a dynamic variable and a variable bound to an element of the finite set can be replaced by a dynamic selection between the elements, followed by specialized code for each element, where the previously dynamic variable is now statically bound to the selected value[8].

Conversely, the technique we propose above (simplifying the administrative lookups) can also be applied to variables that are bound to values in a small finite domain, such as booleans. This would require an extension of the analysis to distinguish between the different kinds of constraint, so that the specializer can lay out a table corresponding to the elements of the finite set at the beginning of specialization. Clearly this technique would be very useful with a typed language.

2.6 Summary

To use this technique, a polyvariant specializer will have to be modified in the following way:

- add the constraint analysis to the pre-processing (probably after the binding-time analysis and the selection of specialization points). The analysis annotates all constrained procedure parameters with the static input that their value must be part of. It also annotates the corresponding calls.
- modify the specializer to construct a table from the parameter annotations and the actual static value and annotate the static value with the indices pointing into the table.
- modify the log lookup procedure to access the table with the given index in case of a constrained argument.

[6]Small, in this connection, means either less than ten or linear in the size of the static data with a medium to large constant factor.

[7]Or an error, which is not relevant to the present discussion.

[8]This is sometimes referred to as "The Trick" [JGS93] and is usually obtained by rewriting the source program manually.

The expected benefit is to avoid searching through the log, replacing the search by table lookups.

3 Further improvement: strictly increasing static values

In the case where we can determine in advance that a particular static argument is always strictly increasing according to some ordering, we can perform a rather dramatic improvement. That the value is increasing is just another way of saying that the partial evaluator will never have to specialize with respect to the same value of the static parameter as one it has seen before. The value of that static parameter will always be different.

3.1 How to take advantage of uniqueness

If a parameter is guaranteed to be different at each call, we can simply forget about it in the log. But in fact we can do even better. Since the specialization point is never called more than once with the same pattern of static values, the corresponding residual program points are all called exactly once in the residual program. So we can re-classify the specialization point to be unfolded, before starting the actual specialization. Then no log administration is needed at all.[9]

This also implies that it is very simple to modify a partial evaluator to take advantage of this property, if it can be detected. The specialization points involved can simply be re-annotated to unfoldable in pre-processing, and the actual specialization proceeds as before.

Note that, in taking advantage of the uniqueness, we again lose the "common-subexpression elimination" effect of the usual polyvariant specializer. If we want to preserve this, we could, for example, change the strategy so that a head of a unique list is no longer considered unique (since it might be identical to a previous head). A tail, however, will always be different from an earlier tail (unless the list is circular), so this will not affect common-subexpression elimination.

The classification of more calls as unfoldable is an advantage both because it speeds up specialization and because it speeds up post-processing, since fewer corridor calls[10] have to be unfolded. Furthermore, it allows a better propagation of static information across the result of the unfolded procedure, giving a binding-time improvement as a side-effect.

The interest in binding-time improvement is motivated by the goal of making the partial evaluator produce programs that are closer to (good) hand-written algorithms. Since a programmer of course knows about and uses the uniqueness of a particular parameter when designing an algorithm, it is not surprising that the automatic use of this property can also improve the residual programs.

3.2 Example: Consel and Danvy's pattern matching

For a good example of this, we consider one of the pattern matching examples from Consel and Danvy [CD91].

In that paper, they consider the program reproduced in figure 4. They observe that specialization gives redundant results, with static computations in the residual program,

[9]In the case where this results in infinite unfolding, that corresponds to infinite residualization in the unmodified program, so nothing is lost.

[10]Calls to procedures that are called exactly one place in the program [BD91].

```
; Result = Unit + Subst                    ; Subst = EmptySubst + ExtendSubst
(defconstr (Unit) (Subst outResS))         ; EmptySubst = Unit
                                           ; ExtendSubst = Var * Data * Subst
                                           (defconstr (EmptySubst) (ExtendSubst outSubv outSubd outSubs))

; Pattern * Data -> Result
(define (main p d)
   (match p d (EmptySubst)))

; Pattern * Data * Subst -> Result
(define (match p d s)
   (let ([tag (car p)])
      (cond [(equal? tag 'PatCst) (if (equal? (cadr p) d) (Subst s) (Unit))]
            [(equal? tag 'PatVar) (Subst (ExtendSubst (cadr p) d s))]
            [(equal? tag 'PatSeq) (match-Seq (cadr p) d s)]
            [else (error 'match "mal-formed pattern: ~s" p)])))

; List(Pattern) * Data * Subst -> Result
(define (match-Seq l d s)
  (if (null? l)
      (if (null? d) (Subst s) (Unit))
      (if (null? d) (Unit)
          (let ([res (match (car l) (car d) s)])
            (if (Unit? res) (Unit)
                (match-Seq (cdr l) (cdr d) (outResS res)))))))
```

Figure 4: The direct-style pattern matching program from Consel and Danvy, with its datatype declarations, converted into Similix.

```
; Data -> Result
(define (main-0 d_0)
  (let ([s_1 (EmptySubst)])
    (if (null? d_0) (Unit)
        (let ([res_3 (Subst (ExtendSubst 'x (car d_0) s_1))])
          (if (Unit? res_3) (Unit)
              (let* ([s_4 (OutResS res_3)] [d_5 (cdr d_0)])
                (if (null? d_5) (Unit)
                    (let ([res_7 (if (equal? 3 (car d_5)) (Subst s_4) (Unit))])
                      (if (Unit? res_7) (Unit)
                          (let ([s_8 (OutResS res_7)])
                            (if (null? (cdr d_5)) (Subst s_8) (Unit))))))))))))
```

Figure 5: Specialized version of the pattern matcher w.r.t. the pattern `(PatSeq ((PatVar x) (PatCst 3)))`. The interpretive overhead of dissecting the pattern has been removed, but the residual program still contains redundant tests, first constructing a result with static components and then testing this result.

```
; Data -> Result
(define (main-0 d_0)
  (if (null? d_0)
      (Unit)
      (let* ([d_1 (car d_0)] [d_2 (cdr d_0)])
        (cond [(null? d_2) (Unit)]
              [(equal? 3 (car d_2)) (if (null? (cdr d_2))
                                        (Subst (ExtendSubst 'x d_1 (EmptySubst)))
                                        (Unit))]
              [else (Unit)]))))
```

Figure 6: Specialized version of the pattern matcher w.r.t. the same pattern as in figure 5, when the calls are classified "unfoldable". This is obtained automatically by modifying Similix-5 to take advantage of the uniqueness and is identical to the improved version from Consel and Danvy. In this version, the redundant tests have been removed and the result substitution is only constructed if the entire match is completed.

as demonstrated by the residual program in figure 5. The problem is that the result of for example the call to `match` in `match-Seq` is partially static, but it has to be classified as dynamic because the call is residual. So the test on `(Unit? res)` cannot be resolved at specialization time.

Their solution is to transform the program into continuation-passing style [Plo75, Ste78]. One property of continuation-passing style is that values are always passed forward, in calls or by applying the continuation, and so the problem goes away. In this way they obtain good residual programs, with no redundant computation.

Bondorf [Bon92] suggests an alternative technique for improving binding-times, using continuation-based specialization instead of CPS-transformation of the source program. This technique is not quite strong enough to handle the pattern matching example, though, since it cannot propagate static information out from specialization points.

Using the idea of uniqueness analysis, however, we observe that the static parameters `p` and `l`, of the procedures `match` and `match-Seq`, are obviously unique: they are used linearly, the program applies only the destructors `car`, `cdr`, and `cadr` to these parameters, and they are destructed at every call.

So we can determine before specialization that all calls to `match` and `match-Seq` can be unfolded. Then the type of their result can be classified partially static and the condition on `(Unit? res)` can be resolved at specialization time. This way, we obtain the same good result as Consel and Danvy, without any program transformation on the source program. Instead, the effect is obtained by a combination of continuation-based specialization, propagating across dynamic conditionals, and uniqueness analysis[11]. Furthermore, specialization will be faster, because the administration of specialization points has been completely eliminated.

This way of producing better residual programs by automatically improving the static data-flow is different from the previous works in this direction [CD91, Bon92], because it is essentially a side-effect of a design introduced for different reasons (faster specialization). It appears to be an orthogonal solution, since it will solve some of the problems addressed in these papers, in the particular, but frequent, cases where a parameter of the procedures involved can be identified as "unique".

3.3 Defining uniqueness

It is worth noticing that this uniqueness property comes in at least two varieties.

The straightforward variety can be exemplified by a counter that is increased with each recursive call until some static upper bound is reached. This value is literally distinct at each call.

The other variety can be exemplified by a list of characters, where we call a procedure `f` with the successive heads of the list. These characters may not be literally distinct, but we can put an ordering on them, based on their position in the list, that makes them increase strictly. In other words, we can *pretend* that they are distinct. If the specializer assumes that they are distinct, it will still terminate and the residual program will be correct. The drawback is that the residual program may contain redundant code — it may be bigger than necessary. The advantage is that specialization may be significantly faster, as long as there are only few redundancies.

The trade-off between specialization speed/residual speed on the one hand and residual size on the other hand appears to be intrinsic to partial evaluation [JGS93]. This should not surprise us, since the space/speed trade-off is intrinsic to algorithm design.

In our example, a human can quickly determine what is best. If the list is short compared to the number of different characters, it is worth optimizing the specialization. If the list is long, the extent of redundancy and the size of the residual program increase. Also, the potential gain at specialization time is smaller, since the partial evaluator has to find the character in a comparatively small set, and if it succeeds (which it is likely to do) it saves the specialization of the body of the procedure `f`.

3.4 Outline: automatic detection of uniqueness

Not surprisingly, uniqueness is a considerably harder property to pin down by program analysis. Essentially we need to determine that at each binding of a parameter to a value, that value is different from all other values that the parameter will be bound to in the execution path. This prohibits any of the usual techniques for obtaining a terminating analysis.

The property is not uncommon, though. For example, the usual append program, specialized with respect to its first argument, has the property.

To find an analysis that detects this property in a safe way, we can note that it is strongly related to linearity conditions such as single-threading [Sch85, Ses89]. In fact, if we have already determined that a particular variable is single-threaded, we can design an analysis that traces the increments to the variables and checks for each procedure/specialization point whether the variable is always incremented before reaching this point again.

Note that although such an analysis may check increments for all the parameters in the program, we can only use its results to optimize partial evaluation for parameters that we also know are single-threaded. If not, the program might duplicate a value and then increment both the duplicates. From the point of view of the analysis, this leads to two "new" values, but in reality, they are identical. So a specialization assuming that the values were new could lead to a code explosion in the residual program. This problem of the analysis arises because the full property that we are looking for is not just a data-flow property.

If the value is single-threaded, however, the analysis will determine if the same concrete value is passed to a procedure more than once.

For this analysis, we represent each value as an array of tags, one for each of the procedures in the program. Every time the value is incremented, all the tags are set to "new". When a procedure is called, we analyze the body in the environment where the parameters are bound to the argument values with the tag for that procedure set to "old". We assume an ordering where old is greater than new and extend this pointwise to arrays.

If a procedure is ever called with an argument that has the tag "old" for that procedure, then the corresponding parameter is not unique.

[11] This result was obtained using Similix-5 [Bon93], modifying it to unfold the calls, according to the uniqueness of the pattern parameters.

Domains:

$$\begin{array}{rcl} v \in Val & = & (\mathit{Direction} \times \mathit{Tag}^*)^\top \\ t \in \mathit{Tag} & = & \{\mathit{new}, \mathit{old}\} \\ d \in \mathit{Direction} & = & \{\mathit{none}, \mathit{cons}, \mathit{dest}\} \\ \rho \in \mathit{Env} & = & \mathrm{Id} \to \mathit{Val} \\ \varphi \in \mathit{Cache} & = & \mathrm{Id} \to \mathit{Val}^* \times \mathit{Val} \end{array}$$

Functions:

$\mathcal{A}_\mathrm{P} : \mathrm{Pgm} \to \mathit{Val}^* \to \mathit{Cache}$

$\mathcal{A}_\mathrm{P}\ [\![(\mathrm{define}\ (\mathrm{F}_0\ \mathrm{I}_1\ \ldots\ \mathrm{I}_{n_0})\ \mathrm{E}_0)\ \ldots\ (\mathrm{define}\ (\mathrm{F}_m\ \mathrm{I}_1\ \ldots\ \mathrm{I}_{n_m})\ \mathrm{E}_m)]\!]\ v^* =$

$$\begin{array}{l} \mathit{fix}\ \lambda\, \varphi.\ \mathit{let}\ (v_0, \varphi_0) = \mathcal{A}\ [\![\mathrm{E}_0]\!] \\ \qquad\qquad\qquad\qquad (\mathit{mk\text{-}env}\ \mathrm{F}_0\ [\mathrm{I}_0, \ldots, \mathrm{I}_{n_0}]\ (\mathit{lookup\text{-}args}\ \mathrm{F}_0\ (\mathit{update\text{-}args}\ \mathrm{F}_0\ v^*\ \varphi))) \\ \qquad\qquad\qquad\qquad (\ldots\ \mathit{let}\ (v_m, \varphi_m) = \mathcal{A}\ [\![\mathrm{E}_m]\!] (\mathit{mk\text{-}env}\ \mathrm{F}_m\ [\mathrm{I}_0, \ldots, \mathrm{I}_{n_m}]\ (\mathit{lookup\text{-}args}\ \mathrm{F}_m \varphi))\ \varphi \\ \qquad\qquad\qquad\qquad\quad \mathit{in}\ (\mathit{update\text{-}res}\ F_m\ v_m\ \varphi_m) \ldots) \\ \qquad \mathit{in}\ (\mathit{update\text{-}res}\ F_0\ v_0\ \varphi_0) \end{array}$$

$\mathcal{A} : \mathrm{Expr} \to \mathit{Env} \to \mathit{Cache} \to \mathit{Val} \times \mathit{Cache}$

$$\begin{array}{rcl} \mathcal{A}\ [\![\mathrm{C}]\!]\ \rho\, \varphi & = & (\top, \varphi) \\ \mathcal{A}\ [\![\mathrm{I}]\!]\ \rho\, \varphi & = & (\mathit{lookup}\ \mathrm{I}\ \rho, \varphi) \\ \mathcal{A}\ [\![(\mathrm{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\ \rho\, \varphi & = & \mathit{let}\ (v_0, \varphi_0) = \mathcal{A}\ [\![\mathrm{E}_0]\!]\ \rho\, \varphi \\ & & \mathit{in}\ (\mathcal{A}\ [\![\mathrm{E}_1]\!]\ \rho\, \varphi_0) \sqcup (\mathcal{A}\ [\![\mathrm{E}_2]\!]\ \rho\, \varphi_0) \\ \mathcal{A}\ [\![(\mathrm{cons}\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\ \rho\, \varphi & = & \mathit{let}\ (v_1, \varphi_1) = \mathcal{A}\ [\![\mathrm{E}_1]\!]\ \rho\, \varphi \\ & & \mathit{in}\ \ \mathit{let}\ (v_2, \varphi_2) = \mathcal{A}\ [\![\mathrm{E}_2]\!]\ \rho\, \varphi_1 \\ & & \quad \mathit{in}\ (\mathit{cons\text{-}vals}\ v_1\ v_2, \varphi_2) \\ \mathcal{A}\ [\![(\mathrm{car}\ \mathrm{E})]\!]\ \rho\, \varphi & = & \mathit{let}\ (v, \varphi) = \mathcal{A}\ [\![\mathrm{E}]\!]\ \rho\, \varphi\ \mathit{in}\ (\mathit{car\text{-}val}\ v, \varphi) \\ \mathcal{A}\ [\![(\mathrm{cdr}\ \mathrm{E})]\!]\ \rho\, \varphi & = & \mathit{let}\ (v, \varphi) = \mathcal{A}\ [\![\mathrm{E}]\!]\ \rho\, \varphi\ \mathit{in}\ (\mathit{cdr\text{-}val}\ v, \varphi) \\ \mathcal{A}\ [\![(\mathrm{F}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n)]\!]\ \rho\, \varphi & = & \mathit{let}\ (v_1, \varphi_1) = \mathcal{A}\ [\![\mathrm{E}_1]\!]\ \rho\, \varphi \\ & & \mathit{in}\ \ldots\ \mathit{let}\ (v_n, \varphi_n) = \mathcal{A}\ [\![\mathrm{E}_n]\!]\ \rho\, \varphi_{n-1} \\ & & \qquad \mathit{in}\ (\mathit{lookup\text{-}res}\ \mathrm{F}\ \varphi_n, \mathit{update\text{-}args}\ \mathrm{F}\ [v_1, \ldots, v_n]\ \varphi_n) \end{array}$$

where

$\mathit{cons\text{-}vals} :\ \mathit{Val} \to \mathit{Val} \to \mathit{Val}$

$\mathit{cons\text{-}vals} = \lambda\ v_1\, v_2.$

$\qquad (v_1 = \top) \vee (v_2 = \top) \vee (v_1 \downarrow 1 = \mathit{dest}) \vee (v_2 \downarrow 1 = \mathit{dest}) \to \top\ [\!]\ (\mathit{cons}, \mathit{mk\text{-}new}\ v_2 \downarrow 2)$

$\mathit{car\text{-}val} :\ \mathit{Val} \to \mathit{Val}$

$\mathit{car\text{-}val} = \lambda\ v.\ (v = \top) \vee (v \downarrow 1 = \mathit{cons}) \to \top\ [\!]\ (\mathit{dest}, \mathit{mk\text{-}new}\ v \downarrow 2)$

$\mathit{mk\text{-}env} : \mathrm{Id} \to \mathrm{Id}^* \to \mathit{Val}^* \to \mathit{Env}$

$\mathit{mk\text{-}env} = \lambda\ \mathrm{F}\ [\mathrm{I}_0, \ldots, \mathrm{I}_n]\ [v_0, \ldots, v_n].\ \lambda\ i.\ i = \mathrm{I}_j \to (\mathit{mk\text{-}old}\ v_j\ \mathrm{F})\ [\!]\ \top$

$\mathit{mk\text{-}new} :\ \mathit{Tag}^* \to \mathit{Tag}^*$

$\mathit{mk\text{-}new} = \lambda\ [t_0, \ldots, t_m].\ [\mathit{new}_0, \ldots, \mathit{new}_m]$

$\mathit{mk\text{-}old} :\ \mathit{Val} \to \mathrm{Id} \to \mathit{Val}$

$$\begin{array}{l} \mathit{mk\text{-}old} = \lambda\ v\ F_j.\ v = \top \to \top\ [\!]\ \mathit{let}\ (d, [t_0, \ldots, t_m]) = v \\ \qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ (d, [t_0, \ldots, t_{j-1}, \mathit{old}, \ldots, t_m]) \end{array}$$

$\mathit{update\text{-}args} : \mathrm{Id} \to \mathit{Val}^* \to \mathit{Cache} \to \mathit{Cache}$

$\mathit{update\text{-}res} : \mathrm{Id} \to \mathit{Val} \to \mathit{Cache} \to \mathit{Cache}$

Figure 7: Analysis detecting increments to values

We also need to keep track of "direction" in the analysis. For example if the value is a list, then cons-ing a value onto the list or taking the cdr of the list both increment it. But after the first operation, only the same operation will increment — the "direction" has been established. So for each value, we need a field giving its direction. Before any operation has been performed, we don't know the direction (corresponding to a bottom value in the domain of directions). If a direction has been determined, for lists for example either "construction" or "destruction", this is noted in the field. If a cons operation is applied to a value with "destruction" direction, we have lost the ability to determine if the concrete value is always new, corresponding to a top element in the domain of values. So the ordering on the domain of directions is discrete, except for the bottom element.

For each procedure parameter, we keep the least upper bound of all the values it has been bound to in a global cache. The cache is updated with the current arguments every time a procedure is called. As before, we take the fixed point over the cache. The value of a parameter is always incremented between two calls to a procedure if the value is tagged "new" in the field corresponding to that procedure.

An analysis following the outline above is shown in figure 7.

4 Partial evaluation without binding-time analysis

The discussion so far has considered polyvariant specialization with binding-time analysis (*i.e.*, "off-line" partial evaluation). Several of the observations do not depend on the presence of a binding-time analysis, though, but apply to any polyvariant specializer using a log.

Even when the binding-time analysis is done "on-the-fly", the log has the same function, and so a pre-analysis for constraints could be used in much the same way. It might even be possible to do the constraint analysis on the fly, starting out by assuming that all static values are also constrained and shifting to the usual log-organization whenever this is discovered to be false. Such a scheme would probably not work well with self-application, though, because it lacks a clear binding-time separation.

The uniqueness analysis should also be applicable, if done as a pre-phase, since it would allow certain procedures to be classified as "definitely unfold", rather than taking a cautious view and having to residualize anything dubious.

5 Related work

To the best of our knowledge, no similar work on partial evaluators has been published.

Both the analyses outlined have a strong resemblance to the minimal function-graph [JM86] family of analyses.

The new aspect here is the application of the results, rather than the actual analyses. In particular the constraint-analysis is relatively straight-forward to develop, once the desired property has been identified and structured properly. It is, for example, equally simple to express the analysis as a set of constraints, in the style of Henglein [Hen91] and use that algorithm for detecting the property.

Note that the ideas presented here are different from the ideas of removing "superfluous" static parameters [RW91], which is typically a concern in partial evaluation of imperative languages [And92]. In that case, one tries to identify parameters that will cause different instances of a specialization point to be produced, but that have no influence on the residual code at that specialization point. If such parameters are not identified, they will cause many identical specialized versions to appear in the residual program. Since they are characterized by having no effect on the specialized version, such "specialization dead" parameters can also be ignored in the log, but with exactly the opposite effect: when one of these are encountered, we want to use the *same* specialized version as the one corresponding to a different value. In the case of a strictly increasing value, we know that we always want to create a new specialized version, without having to look in the log.

6 Conclusion and issues

We have focused on a central aspect of the polyvariant specialization algorithm, the log of previously treated calls. We have mentioned the standard ways of maintaining such an administration in a more efficient way than a simple list structure.

Seeking a more conceptual improvement, we have classified special cases where the full algorithm is too general. Based on this, we have identified two un-related properties of source programs that allow more efficient specialization, proposed algorithms for (safe) detection of these properties, and outlined how the partial evaluator can take advantage of the results. Not surprisingly, the property that requires the most complex analysis is also the one that is the most easy to take advantage of. We have given examples of the improvements that can be obtained, and as a side-effect, we have found that the resulting simplifications may also lead to binding-time improvements.

There is still much work to be done, both in extending the present analyses to handle partially-static structures and higher-order functions, designing more clever analyses for the detection of less narrowly defined properties, and in writing or modifying a partial evaluator to take advantage of these properties. The detection and use of such properties, however, seem necessary to the task of moving partial evaluators from the academically interesting "program that runs" to the realm of practical applications.

Acknowledgements

Thanks to Anders Bondorf, for letting me play with a pre-release version of Similix-5, to the anonymous referees, for asking several good questions that I had to find answers to and for helpful references, and to Olivier Danvy, for numerous discussions.

References

[And92] L. O. Andersen. C program specialization. Technical Report 92/14, DIKU, University of Copenhagen, Denmark, May 1992.

[BD91] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[Bon91a] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[Bon91b] A. Bondorf. Similix manual, system version 4.0. DIKU, University of Copenhagen, Denmark. September, 1991.

[Bon92] A. Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference in Lisp and Functional Programming, San Francisco, California. (Lisp Pointers, vol. V, no. 1, 1992)*, pages 1–10. ACM, 1992.

[Bon93] A. Bondorf. Similix manual, system version 5.0. DIKU, University of Copenhagen, Denmark, April 1993.

[CD91] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 496–519. ACM, Springer-Verlag, 1991.

[CR91] W. Clinger and J. Rees. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[Got74] E. Goto. Monocopy and associative algorithms in an extended Lisp. Technical report, University of Tokyo, Japan, May 1974.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Springer-Verlag, 1991.

[Hol91] C. K. Holst. Finiteness analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 473–495. ACM, Springer-Verlag, 1991.

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993. To appear.

[JM86] N. D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 296–306. ACM, 1986.

[JSS89] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[RW91] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 321–333. ACM, 1991.

[Sch85] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):299–310, April 1985.

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

[Ses86] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 236–256. Springer-Verlag, 1986.

[Ses89] P. Sestoft. Replacing function parameters by global variables. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, pages 39–53. IFIP and ACM, ACM Press and Addison-Wesley, September 1989.

[Ste78] G. L. Steele, Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.