Pragmatics of Type-Directed Partial Evaluation

Olivier Danvy

Computer Science Department
Aarhus University *
http://www.brics.dk/~danvy

Abstract. Type-directed partial evaluation stems from the residualization of arbitrary static values in dynamic contexts, given their type. Its algorithm coincides with the one for coercing a subtype value into a supertype value, which itself coincides with the one of normalization in the λ-calculus. Type-directed partial evaluation is thus used to specialize compiled, closed programs, given their type.
Since Similix, let-insertion is a cornerstone of partial evaluators for call-by-value procedural programs with computational effects. It prevents the duplication of residual computations, and more generally maintains the order of dynamic side effects in residual programs.
This article describes the extension of type-directed partial evaluation to insert residual let expressions. This extension requires the user to annotate arrow types with effect information. It is achieved by delimiting and abstracting control, comparably to continuation-based specialization in direct style. It enables type-directed partial evaluation of effectful programs (*e.g.*, a definitional lambda-interpreter for an imperative language) that are in direct style. The residual programs are in A-normal form.

1 Introduction

Type-directed partial evaluation [12] stems from the need to residualize arbitrary static values in dynamic contexts, in a partial evaluator [10, 23]. It requires the type of these values, including that of their free variables, if these values are higher-order, and in practice, works on the value — not on the text — of closed source programs. Its algorithm parallels the one for source binding-time improvements [15, 16], and coincides with the one of coercion in type systems with subtypes [21, 22], and with the one of normalization in proof theory [1] and logical frameworks [30]. Interpreting static expressions as executable code and dynamic expressions as code construction yields type-directed partial evaluation: the specialization of compiled code into the text of its (long $\beta\eta$) normal form [12].

Type-directed partial evaluation encounters the same problem as all other partial evaluators for call-by-value programs: computation duplication. For example, consider the following procedure (where the type constructor `=>` accounts for Scheme's uncurried procedures [8]).

* Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. (danvy@brics.dk)

```
(define foo      ;;; ((a -> b) * a * ((b * b) => c)) => c
  (lambda (f x k)
    ((lambda (v) (k v v)) (f x))))
```

Let us residualize the value of `foo`. (Its text is unavailable: it has been compiled away.)

```
> (residualize-basic foo ´(((a -> b) * a * ((b * b) => c)) => c))
(lambda (f0 a1 f2)
  (f2 (f0 a1) (f0 a1)))
>
```

A computation has been duplicated: that of the application of the first argument of `foo` to its second. Sometimes this duplication is of no consequence, *e.g.*, if the function denoted by the first argument of `foo` is pure and total. If the function is furthermore inexpensive, computation duplication then reduces to code duplication. In general, however, this duplication is not wanted.

The point of this paper is to remedy to this situation. We extend the language of types handled by type-directed partial evaluation to account for impure procedures, whose application should not be duplicated. Our treatment is proven [7] — we insert a residual let expression.

Let us residualize the value of `foo` again. This time, we specify that its first argument is effectful. (Syntactically, the type of this argument is written with a long arrow `-->`.)

```
> (residualize-mulets foo ´(((a --> b) * a * ((b * b) => c)) => c))
(lambda (f0 a1 f2)
  (let ([b3 (f0 a1)])
    (f2 b3 b3)))
>
```

A residual let expression has been inserted.

This let insertion naturally scales up to more voluminous programs, yielding residual programs in "CPS without continuations" (a.k.a. "nqCPS", "A-normal forms" [19], "monadic normal forms" [20], *etc.*), as illustrated below.

```
> (residualize-mulets (lambda (f x k)
                        ((lambda (v) (k v v)) (f (f x))))
                      ´(((b --> b) * b * ((b * b) => c)) => c))
(lambda (f0 b1 f2)
  (let* ([b3 (f0 b1)]
         [b4 (f0 b3)])
    (f2 b4 b4)))
>
```

The reader should keep in mind that inserting let expressions is something of a challenge, since in contrast to all other existing partial evaluators, we have no access to the text of the source program. In the interaction above, `residualize-mulets` is a Scheme procedure, and Scheme obeys call-by-value.

The rest of this paper is structured as follows. We first start with a side-issue about naming residual variables (Section 2). This side-issue is pragmatically trivial, but solving it does improve the readability of residual programs. Thus equipped, we review the problem of residual computational effects in partial evaluation, and its solutions (Section 3). We then specialize Section 3 with respect to type-directed partial evaluation (Section 4). This makes it possible to specialize a *direct-style* interpreter for Paulson's Tiny language (Section 5). As a corollary of Section 4, we describe the CPS transformation of compiled programs in normal form (Section 6). The technique of Section 4 relies on delimiting and abstracting control, which clashes with similar control operations to handle disjoint sums. Section 7 outlines how to make both coexist. After a comparison with related work (Section 8), we conclude (Section 9).

2 What is in a name?

Under lexical scope, names of local variables do not matter. In practice, though, they contribute to program readability, and thus programmers usually pick "meaningful" identifiers. One reason why automatically generated programs are hard to read is precisely because they have uninformative identifiers.

Schism's and Similix's residual identifiers, for example, depend on the history of the static computation and are spiced with source scope information [5, 9]. They can be very long. Often, to the relative delight of the user, they differ only subtly one from another.

Our strategy for picking residual names (surprise) is type-directed.

2.1 Implicit naming

Variables of base types are named after the first letter of their type, catenated with a gensym-generated number. A similar scheme applies for variables of higher types. Their name stub is "`p`" for products, "`s`" for sums, "`b`" for booleans, and "`f`" for functions.

2.2 Explicit naming

The user can also specify name stubs in the type language, through the directive `with`. The directive `alias` specifies a full name — which may come handy if no name clash is expected.

2.3 An example

For example, the type `((a * (b with "Y") * (c alias "Juliet")) => c)` denotes an uncurried Scheme procedure with three arguments, the second of which is associated with the name stub "`Y`", and the third of which is associated with the name "`Juliet`".

```
> (residualize-mulets
    (lambda (x y z) z)
    `((a * (b with "Y") * (c alias "Juliet")) => c))
(lambda (a0 Y1 Juliet) Juliet)
>
```

3 Sound call unfolding under call-by-value

To propagate constants across procedure boundaries, a partial evaluator unfolds calls. Not all parameters may be static, however, and thus under call-by-value, call unfolding is unsound in general. Against this backdrop, and to tame partially static structures, Torben Mogensen suggested to insert a residual let expression for each dynamic parameter, and to pass on the residual identifier naming the dynamic actual parameter instead of this parameter [26] — the rationale being that under call-by-value, this parameter can be duplicated without compromising the dynamic semantics of the source program, as illustrated in Section 1.

This simple solution, put at the core of Similix, before it even had partially static values, has scaled up remarkably well, *e.g.*, to provide a simple solution to the thorny problem of automating call unfolding [32], and also to provide a sound treatment for dynamic side-effects [7]. Doubled with a variable-splitting mechanism [27], it provides a simple and elegant treatment of both partially static values and higher-order values [4].

In the next section, we adapt this let-insertion technique to type-directed partial evaluation.

4 The particular case of type-directed partial evaluation

Lacking access to the source code, it is impossible to insert residual let expressions at call sites — they are compiled, along with the rest of the source program. So let us go upstream from calls: since the point of residual let expressions is to stop dynamic expressions from flowing across a procedure boundary, let us see *where these dynamic expressions come from*. Essentially, they come from residual calls to effectful procedures, which form these dynamic expressions.

Thus we choose (1) to annotate the type of effectful procedures, (2) to insert a residual let expression naming their result when they are unfolded, and (3) to return the residual name to the current context. Point (3) requires one to delimit and to abstract control [13, 14, 17], comparatively to the strategy for continuation-based partial evaluation in direct style [6, 25].

The corresponding specification of type-directed partial evaluation is shown in Figure 1. It is an extension of type-directed partial evaluation [12].

The domains Value and Expr are defined inductively, following the structure of types, and starting from the same set of (dynamic) base types. TLT is the domain of (well-typed) two-level terms; it contains both Value and Expr.

The down arrow is read *reify*: it maps a static value and its type into a two-level λ-term that statically reduces to the dynamic counterpart of this static

$$
\begin{aligned}
t \in \text{Type} &::= b \mid t_1 \to t_2 \mid t_1 \longrightarrow t_2 \mid t_1 \times t_2 \\
v \in \text{Value} &::= c \mid x \mid \overline{\lambda} x : t.v \mid v_0 \,\overline{@}\, v_1 \mid \\
&\quad\ \overline{\text{pair}}(v_1, v_2) \mid \overline{\text{fst}}\ v \mid \overline{\text{snd}}\ v \\
e \in \text{Expr} &::= c \mid x \mid \underline{\lambda} x : t.e \mid e_0 \,\underline{@}\, e_1 \mid \\
&\quad\ \underline{\text{pair}}(e_1, e_2) \mid \underline{\text{fst}}\, e \mid \underline{\text{snd}}\, e \mid \\
&\quad\ \underline{\text{let}}\ x = e_1\ \underline{\text{in}}\ e_2
\end{aligned}
$$

$$
\begin{aligned}
\text{reify} &= \lambda t.\lambda v : t.{\downarrow}^t\ v \\
&: \text{Type} \to \text{Value} \to \text{TLT} \\
{\downarrow}^b\ v &= v \\
{\downarrow}^{t_1 \to t_2}\ v &= \underline{\lambda} x_1.\overline{\text{reset}}_{t_2}\ {\downarrow}^{t_2}\ (v\,\overline{@}\,{\uparrow}^{t_2}_{t_1}\ x_1) \\
{\downarrow}^{t_1 \longrightarrow t_2}\ v &= \underline{\lambda} x_1.\overline{\text{reset}}_{t_2}\ {\downarrow}^{t_2}\ (v\,\overline{@}\,{\uparrow}^{t_2}_{t_1}\ x_1) \\
&\quad \text{where } x_1 \text{ is fresh.} \\
{\downarrow}^{t_1 \times t_2}\ v &= \underline{\text{pair}}({\downarrow}^{t_1}\ \overline{\text{fst}}\ v, {\downarrow}^{t_2}\ \overline{\text{snd}}\ v)
\end{aligned}
$$

$$
\begin{aligned}
\text{reflect} &= \lambda t'.\lambda t.\lambda e : t.{\uparrow}^{t'}_{t}\ e \\
&: \text{Type} \to \text{Type} \to \text{Expr} \to \text{TLT} \\
{\uparrow}^t_b\ e &= e \\
{\uparrow}^t_{t_1 \to t_2}\ e &= \overline{\lambda} v_1.{\uparrow}^t_{t_2}\ (e\,\underline{@}\,{\downarrow}^{t_1}\ v_1) \\
{\uparrow}^t_{t_1 \longrightarrow t_2}\ e &= \overline{\lambda} v_1.\overline{\text{shift}}\ \kappa : t_2 \to t\ \overline{\text{in}}\ \underline{\text{let}}\ x_2 = e\,\underline{@}\,{\downarrow}^{t_1}\ v_1 \\
&\qquad\qquad \underline{\text{in}}\ \overline{\text{reset}}_t\ (\kappa\,\overline{@}\,{\uparrow}^t_{t_2}\ x_2) \\
&\quad \text{where } x_2 \text{ is fresh.} \\
{\uparrow}^t_{t_1 \times t_2}\ e &= \overline{\text{pair}}({\uparrow}^t_{t_1}\ \underline{\text{fst}}\, e, {\uparrow}^t_{t_2}\ \underline{\text{snd}}\, e)
\end{aligned}
$$

Reset and reflect are annotated with the type of the value expected by the delimited context.

$$
\begin{aligned}
\text{residualize} &= \text{statically-reduce} \circ \text{reify} \\
&: \text{Type} \to \text{Value} \to \text{Expr}
\end{aligned}
$$

Fig. 1. Type-directed residualization with let insertion

value. Conversely, the up arrow is read *reflect*: it maps a dynamic expression into a two-level λ-term representing the static counterpart of this dynamic expression.

In residualize, reify (resp. reflect) is applied to types occurring positively (resp. negatively) in the source type.

Figure 1 generalizes the specification of type-directed partial evaluation from the POPL'96 proceedings in the following sense. Removing $\overset{!}{\longrightarrow}$ from the type language yields the original specification — remembering the algebraic property of reset [13, 14]:

Property 1 *For any expression e containing no occurrence of shift, $reset(e) = e$.*

In the presence of effectful procedures, the result of type-directed partial evaluation contains series of flat let expressions. These are characteristic of nqCPS.

5 An example: Paulson's Tiny interpreter in direct style

Section 2.3 of "Type-Directed Partial Evaluation" [12] addresses the classical example of Paulson's Tiny interpreter [29], a.k.a. MP in partial-evaluation circles [4, 7, 23, 27].

$$
\begin{aligned}
\langle pgm \rangle &::= \langle name \rangle^* \ \ \langle cmd \rangle \\
\langle cmd \rangle &::= \mathbf{skip} \mid \langle cmd \rangle \ ; \langle cmd \rangle \mid \langle ide \rangle := \langle exp \rangle \mid \\
&\qquad \mathbf{if} \, \langle exp \rangle \, \mathbf{then} \, \langle cmd \rangle \, \mathbf{else} \, \langle cmd \rangle \mid \\
&\qquad \mathbf{while} \, \langle exp \rangle \, \mathbf{do} \, \langle cmd \rangle \, \mathbf{end} \\
\langle exp \rangle &::= \langle int \rangle \mid \langle ide \rangle \mid \langle exp \rangle \, \langle op \rangle \, \langle exp \rangle \mid \mathbf{read} \\
\langle op \rangle &::= + \mid - \mid \times \mid = \mid \geq
\end{aligned}
$$

It is a simple exercise to write the corresponding definitional interpreter *in direct style* (in contrast to the POPL'96 proceedings where it was written in continuation style), to apply it to, *e.g.*, the factorial program, and to residualize the result.

Figure 2 displays this residual program. It is the direct-style counterpart of the residual program displayed in the POPL'96 proceedings. Thanks to the naming scheme of Section 2, it is also directly readable. (N.B. We could have taken advantage of the single-threadedness of the store to generate a unique residual identifier for it [31].)

What is particularly lovely here is that this residual program has been generated straight out of an ordinary direct-style interpreter, which was compiled with an ordinary Scheme compiler. Yet not only is the residual program perfectly readable but in addition it was generated at the speed and efficiency of compiled code.

6 Corollary: CPS transformation of compiled programs

It is very simple to translate nqCPS terms into CPS [11, 24]. Let expressions, for example, in the context of a continuation k, are essentially desugared as follows:

$$[\![\text{let } v = f@x \text{ in } e]\!] \, k = f@x@(\lambda v.[\![e]\!] \, k)$$

This makes it simple to adapt Figure 1 to produce CPS terms. For lack of time, we do not display the adapted figure in this submission. The corresponding program is available through the author's home page.

```
(lambda (add sub mul equal gt read fix true? lookup update)
  (lambda (s0)
    (let* ([i1 (read)]
           [s2 (update 1 i1 s0)]
           [s3 (update 2 1 s2)]
           [s18 ((fix (lambda (f4)
                        (lambda (s5)
                          (let* ([i6 (lookup 1 s5)]
                                 [i7 (gt i6 0)])
                            (true? i7
                                   (lambda (s8)
                                     (let* ([i9 (lookup 1 s8)]
                                            [i10 (lookup 2 s8)]
                                            [i11 (mul i10 i9)]
                                            [s12 (update 2 i11 s8)]
                                            [i13 (lookup 1 s12)]
                                            [i14 (sub i13 1)]
                                            [s15 (update 1 i14 s12)])
                                       (f4 s15)))
                                   (lambda (s17) s17)
                                   s5)))))
                 s3)]
           [i19 (lookup 2 s18)])
      (update 0 i19 s18))))
```

This residual program is a specialized version of the Tiny interpreter with respect to the factorial source program. As can be observed, it is a direct-style Scheme program in A-normal form, threading the store throughout. The while loop has been mapped into a fixed-point declaration (reflecting the semantics of while loops in the Tiny interpreter). All the location offsets have been computed at partial-evaluation time.

Fig. 2. Residual factorial program (*without* hand-renaming)

7 Out of control: let insertion vs. disjoint sums

In the POPL'96 proceedings, shift and reset are used to handle disjoint sums [12, Section 3]. This use clashes with the let insertion of Section 4. There is, however, a natural hierarchy in these control abstractions, where the treatment for disjoint sums clearly supersedes the treatment for let insertion. This is thus a case for shift_2 and reset_2 [13]. We leave this aspect for a future work.

8 Related work

Section 1 has already situated type-directed partial evaluation among related work: it stems from the need to residualize static values in dynamic contexts at

higher type; its algorithm coincides with the one for higher-order coercions, and also with the one for λ-calculus normalization. We express it using the two-level λ-calculus [28]. This coincidence of algorithms shows that there is as much computational power in residualization as in an offline monovariant partial evaluator for the λ-calculus. Picking a particular representation of staticness (compiled syntax constructions) and of dynamicness (compiled syntax constructors) makes it possible to specialize closed compiled programs, given their type.[2]

In his PhD thesis [18], Filinski defines extensional mappings between monadic values (and programs them in Standard ML). In particular, this makes it possible to define an extensional CPS transformation, in the particular case of the identity monad and of the continuation monad. Composing this transformation with Berger and Schwichtenberg's algorithm (also programmed in Standard ML) yields the same effect as the CPS transformation of Section 6.

9 Conclusion and issues

We have extended type-directed partial evaluation with two pragmatic features: the ability to specify identifiers and to insert let expressions in residual programs. This makes it possible to ensure sound call unfolding (which matters since type-directed partial evaluation is inherently call-by-value), to specialize direct-style programs containing dynamic computational effects, and to improve the readability of residual programs.

References

1. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
2. Dines Bjørner, Andrei P. Ershov, and Neil D. Jones. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.
3. Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.
4. Anders Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. DIKU Report 90-17.
5. Anders Bondorf. Similix manual, system version 3.0. Technical Report 91/9, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1991.
6. Anders Bondorf. Improving binding times without explicit cps-conversion. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and*

[2] The referee encouraged us to stress the distinction between constructions and constructors: a constructor generates a construction. This distinction proves essential in the context of program-generating programs.

Functional Programming, LISP Pointers, Vol. V, No. 1, pages 1–10, San Francisco, California, June 1992. ACM Press.

7. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.
8. William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.
9. Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.
10. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
11. Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. Special Issue on ESOP'92, the Fourth European Symposium on Programming, Rennes, February 1992.
12. Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM Press.
13. Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.
14. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.
15. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, 8(3):209–227, 1995. An earlier version appeared in the proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.
16. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. Technical report BRICS RS-95-41, DAIMI, Computer Science Department, Aarhus University, Aarhus, Denmark, August 1995.
17. Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988.
18. Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995.
19. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.
20. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [3], pages 458–471.
21. Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1993. Special Issue on ESOP'92, the Fourth European Symposium on Programming, Rennes, February 1992.

22. Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In Boehm [3], pages 213–226.
23. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.
24. Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, USA, July 1994.
25. Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.
26. Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Bjørner, Ershov, and Jones [2], pages 325–347.
27. Torben Æ. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, March 1989.
28. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
29. Larry Paulson. Compiler generation from denotational semantics. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.
30. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
31. David A. Schmidt. Detecting global variables in denotational definitions. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.
32. Peter Sestoft. Automatic call unfolding in a partial evaluator. In Bjørner, Ershov, and Jones [2], pages 485–506.

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