Basic Research in Computer Science

BRICS RS-96-15 O. Danvy: Pragmatic Aspects of Type-Directed Partial Evaluation

Pragmatic Aspects of Type-Directed Partial Evaluation

Olivier Danvy

BRICS Report Series **RS-96-15**

ISSN 0909-0878 **May 1996**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Pragmatics of Type-Directed Partial Evaluation

Olivier Danvy

BRICS *

Computer Science Department

Aarhus University †

(danvy@brics.dk)

May 1996

Abstract

Type-directed partial evaluation stems from the residualization of static values in dynamic contexts, given their type and the type of their free variables. Its algorithm coincides with the algorithm for coercing a subtype value into a supertype value, which itself coincides with Berger and Schwichtenberg's normalization algorithm for the simply typed λ-calculus. Type-directed partial evaluation thus can be used to specialize a compiled, closed program, given its type.

Since Similix, let-insertion is a cornerstone of partial evaluators for call-by-value procedural languages with computational effects (such as divergence). It prevents the duplication of residual computations, and more generally maintains the order of dynamic side effects in the residual program.

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 programs with effects (*e.g.*, a definitional lambda-interpreter for an imperative language) that are in direct style. The residual programs are in A-normal form. A simple corollary yields CPS (continuation-passing style) terms instead. We illustrate both transformations with two interpreters for Paulson's Tiny language, a classical example in partial evaluation.

---

*Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55. Home page: http://www.brics.dk/~danvy.

1 Introduction

1.1 Background

During partial evaluation [11, 27], parts of a program are evaluated and parts are reconstructed. The parts that are reconstructed yield residual expressions forming the residual program. The parts that are evaluated yield static values. Either of two things can happen to a static value: it may be consumed statically or it may be residualized, *i.e.*, it may be turned into a residual expression whose evaluation will yield a corresponding dynamic value. Sometimes, during partial evaluation, a static value can be both consumed statically and residualized.
For example, in the following binding-time annotated Scheme program [8],

```
(lambda (a b c d)
  (let ([r (gensym! "x")])
    `(lambda (,r)
       ,(if (< a b)
            `(r ,(- c) ,c)
            d))))
```

`a`, `b`, `c`, and `d` denote static integers. The static integers denoted by `a` and `b` are consumed statically in the test; the static integer denoted by `d` is residualized; and the static integer denoted by `c` is both consumed statically (it is negated) and residualized.

Applying the procedure above to `1`, `2`, `3`, and `4` yields the residual program:

```
(lambda (x6)
  (x6 -3 3))
```

where `x6` is a fresh variable.

Had the binding-time analysis been more conservative, and required that static values be exclusively consumed or exclusively residualized, the residual program would be less specialized — namely it would read as follows.

```
(lambda (x6)
  (x6 (- 3) 3))
```

At base type, contemporary partial evaluators do not make this approximation, and thus they allow this double status of static values (*i.e.*, consumable and residualizable). At higher type, however, offline partial evaluators

with monovariant binding-time analyses do make this approximation [27]. Lacking a residualization function at higher type, they do not allow the double status of static values. Instead, they favor residualization. Thus the binding-time analysis dynamizes the offending values, and the specializer yields underspecialized programs. Better specialization requires the users to "improve the binding times" of their source programs [27, Chapter 12].

This residualization function operating at higher types forms the starting point of "type-directed partial evaluation" [13].

1.2 Type-directed partial evaluation

Type-directed partial evaluation stems from the desire to residualize arbitrary static values in dynamic contexts. Residualizing static values requires knowing the type structure of these values. If these values are higher-order, residualization also requires the type structure of their free variables. Its algorithm parallels the one for source binding-time improvements at higher type [17, 18], and coincides with the coercion algorithm in type systems with subtypes [25, 26], and with a normalization algorithm in proof theory [1] and logical frameworks [35]. This last coincidence suggests that it is possible to specialize compiled programs, by interpreting static expressions as executable code and dynamic expressions as code constructors. We have named this process "type-directed partial evaluation": the specialization of compiled code into the text of its (long $\beta\eta$) normal form [13]. A type-directed partial evaluator is thus unconventional in that it does not process the text of a source program, but its compiled (higher-order) value. The normalization effect is not obtained by symbolic interpretation — it happens *en passant* in the residualization algorithm.

We have described the principles and applications of type-directed partial evaluation elsewhere [13]. In this paper, we investigate some more pragmatic aspects, and merely assume from the reader some rudimentary knowledge of partial evaluation [11, 27] and of the Scheme programming language [8].

1.3 Computation duplication

A type-directed partial evaluator 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, and where `a`, `b`, and `c` denote base types).

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

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

```
> (residualize foo '(((a -> b) a ((b b) => c)) => c))
(lambda (x0 a1 x2)
  (x2 (x0 a1) (x0 a1)))
>
```

A computation is 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 (*i.e.*, side-effect free), total, and inexpensive. In general, however, both computation duplication and code duplication are not wanted.

The point of this paper is to remedy 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 standard [7] — we insert a residual let expression.

1.4 Let insertion

Let us residualize the value of `foo` again. This time, we specify that its first argument might perform a side effect (indicated by an annotated arrow `-!>`).

```
> (residualize foo '(((a -!> b) a ((b b) => c)) => c))
(lambda (x0 a1 x2)
  (let ([b3 (x0 a1)])
    (x2 b3 b3)))
>
```

A residual let expression has been inserted.

This let insertion naturally scales up, yielding residual programs in "CPS without continuations" (a.k.a. "nqCPS", "A-normal forms" [22], "monadic normal forms" [24], *etc.*), as illustrated below.

```
> (residualize (lambda (f x k)
                 ((lambda (v) (k v v)) (f (f x))))
               '(((b -!> b) b ((b b) => c)) => c))
```

```
(lambda (x0 b1 x2)
  (let* ([b3 (x0 b1)]
         [b4 (x0 b3)])
    (x2 b4 b4)))
>
```

Residual let expressions can also retain dynamic computations whose result is unused, as illustrated below.

```
> (residualize (lambda (f x y)
                 ((lambda (v) y) (f (f x))))
               '(((b -!> b) b c) => c))
(lambda (x0 b1 c2)
  (let* ([b3 (x0 b1)]
         [b4 (x0 b3)])
    c2))
>
```

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 interactions above, `residualize` is not a macro — it is a Scheme procedure and thus it processes (compiled) Scheme expressible values.

1.5 Overview

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 apply Section 3 to type-directed partial evaluation (Section 4). This makes it possible to specialize both a direct-style and a continuation-style interpreter for Paulson's Tiny language (Section 5). As a corollary of Section 4, we outline the CPS transformation of compiled programs in normal form (Section 6). After a comparison with related work (Section 7), we conclude (Section 8).

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. Our strategy for picking residual names is type-directed.

2.1 Implicit naming

The two special forms `define-base-type` and `define-compound-type` are used to declare types. By default, variables of declared types are named after the first letter of the declared type name, catenated with a gensym-generated number. Undeclared variables of compound types start with the letter `x` followed with a gensym-generated number. We refer to these letters as *name stubs*.

Let us illustrate implicit naming with the first Scheme session of Section 1.

```
> (define-base-type a)
> (define-base-type b)
> (define-base-type c)
> (define-compound-type fun-from-a-to-b (a -> b))
> (define-compound-type Bar
                        ((fun-from-a-to-b a ((b b) => c)) => c))
> (residualize (lambda (f a k)
                 ((lambda (v) (k v v)) (f x)))
               'Bar)
(lambda (f0 a1 x2)
  (x2 (f0 a1) (f0 a1)))
>
```

In this session, `a` is declared as a base type, and gives rise to the residual variable `a1` (the corresponding name stub is `a`); `fun-from-a-to-b` is declared as a compound type, and gives rise to the residual variable `f0` (the corresponding name stub is `f`); and the residual variable `x2` was generated out of the anonymous type `(b b) => c` (the corresponding name stub is `x`).

The definition of declared types is substituted for each later occurrence of their name. So for example, the type denoted by `Bar` is textually the same as the type specified in the first Scheme session of Section 1, modulo the name stubs.

2.2 Explicit naming

Users can specify name stubs in the declaration of a type. Daring users can also specify a full name with a directive `alias`. This may come in handy

if no name clash is expected. Name clashes do not occur when there is only one instance of a variable of a declared type. This can happen either statically (the variable is declared at the outset of a residual program) or dynamically (all variables of this type denote a single-threaded value [36]). Both instances are illustrated in Section 5.

2.3 An example

The type `(b (c -> b) c) => b` denotes an uncurried Scheme procedure with three arguments. We associate the name stub "`Y`" to the (base) type of the first argument, the name stub "`foo`" to the (compound) type of the second argument, and the name "`Juliet`" to the (base) type of the third argument — assuming case sensitivity.

```
> (define-base-type b "Y")
> (define-base-type c "Juliet" alias)
> (define-compound-type f (c -> b) "foo")
> (define-compound-type g ((b f c) => b))
> (residualize (lambda (x y z) (y z)) 'g)
(lambda (Y0 foo1 Juliet) (foo1 Juliet))
>
```

2.4 Summary

Explicit names and name stubs in the types determine the names of residual variables in residual programs.

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 argument instead of the argument itself [31]. As illustrated in Section 1, under call-by-value, let-declared identifiers can be duplicated without compromising the dynamic semantics of source programs.

This simple solution, put at the core of Similix, before it even had partially static values, has scaled up remarkably well, *e.g.*, to solve the thorny

problem of automating call unfolding [37], and also to treat dynamic side-effects soundly [7]. Doubled with a variable-splitting mechanism [32], 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. However, the only dynamic expressions that should not be duplicated are residual calls to procedures that may perform side effects. Therefore it it sufficient to name these residual calls and return the corresponding (fresh) identifiers to the current context. This follows the spirit of lightweight symbolic values [30], where the only dynamic expressions in the data flow are residual identifiers.

Thus we choose (1) to annotate the type of procedures that may perform side effects, (2) to insert a residual let expression naming their result when one of their calls is unfolded, and (3) to return the residual name to the context of this call. Point (3) requires us to relocate the context of the call in the body of the let expression. This relocation is achieved by abstracting delimited control, for example with shift and reset [14, 15, 19]. This approach is similar to the strategy for continuation-based partial evaluation [6, 29].

The complete specification of type-directed partial evaluation is shown in Figure 1, using the two-level λ-calculus [33], and in Figure 2, using Scheme. Overlined λ's and @'s denote ordinary λ-abstractions and applications. Underlined λ's and @'s denote the corresponding (hygienic) syntax constructors. 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 value. Reify is applied to types occurring positively in the source type. Conversely, the up arrow is read *reflect*: it maps a dynamic expression and its type into a two-level λ-term representing the static counterpart of this

$$
\begin{array}{rcl}
t \in \text{Type} & ::= & b \mid t_1 \times t_2 \mid t_1 \to t_2 \mid t_1 \xrightarrow{!} t_2 \\
v \in \text{Value} & ::= & c \mid x \mid \overline{\lambda} x : t.v \mid v_0 \,\overline{@}\, v_1 \mid \\
 & & \overline{\text{pair}}(v_1, v_2) \mid \overline{\text{fst}}\ v \mid \overline{\text{snd}}\ v \mid \\
 & & \overline{\text{shift}}\ k : t_1 \to t_2\ \overline{\text{in}}\ \underline{\text{let}}\ x : t_1 = e_0 \,\underline{@}\, e_1\ \underline{\text{in}}\ \overline{\text{reset}}_{t_2}\ k \,\overline{@}\, v \\
e \in \text{Expr} & ::= & c \mid x \mid \underline{\lambda} x : t.e \mid e_0 \,\underline{@}\, e_1 \mid \\
 & & \underline{\text{pair}}(e_1, e_2) \mid \underline{\text{fst}}\, e \mid \underline{\text{snd}}\, e \mid \overline{\text{reset}}_t\ e
\end{array}
$$

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

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

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

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

Figure 1: Type-directed residualization with let insertion

```
(define-record (Base name stub))
(define-record (Prod type type stub))
(define-record (Func type type stub))
(define-record (Proc type type stub))

(define residualize
  (lambda (v t)
    (letrec ([reify
              (lambda (t v)
                (case-record t
                  [(Base name stub) v]
                  [(Prod t1 t2 stub) `(cons ,(reify t1 (car v))
                                            ,(reify t2 (cdr v)))]
                  [(Func t1 t2 stub)
                   (let ([x1 (elaborate-new-name t1)])
                     `(lambda (,x1)
                        ,(Reset (reify t2 (v (reflect t1 x1))))))]
                  [(Proc t1 t2 stub)
                   (let ([x1 (elaborate-new-name t1)])
                     `(lambda (,x1)
                        ,(Reset (reify t2 (v (reflect t1 x1)))))))]))]
             [reflect
              (lambda (t e)
                (case-record t
                  [(Base name stub) e]
                  [(Prod t1 t2 stub) (cons (reflect t1 `(car ,e))
                                           (reflect t2 `(cdr ,e)))]
                  [(Func t1 t2 stub)
                   (lambda (v1)
                     (reflect t2 `(,e ,(reify t1 v1))))]
                  [(Proc t1 t2 stub)
                   (lambda (v1)
                     (let ([q2 (elaborate-new-name t2)])
                       (Shift k
                              `(let ([,q2 (,e ,(reify t1 v1))])
                                 ,(Reset
                                    (k (reflect t2 q2))))))))]))])
      (begin (reset-gensym!) (reify (parse-type t) v)))))
```

Figure 2: Type-directed partial evaluation with let insertion in Scheme

dynamic expression. Reflect is applied to types occurring negatively in the source type.

The generation of residual calls (to pure procedures) reads as follows [13].

$$\uparrow_{t_1 \to t_2} e \quad = \quad \overline{\lambda} v_1 : t_1 . \uparrow_{t_2} (e \, \underline{@} \downarrow^{t_1} v_1)$$

As illustrated in Section 1, we cannot let residual calls to impure procedures flow uncontrolled in the residualization context. Instead, we want (a) to insert a residual let expression naming this residual call and (b) let the freshly declared identifier flow instead. This requires us to abstract the residualization context of impure calls and to relocate it in the body of a residual let expression. (N.B. The residualization context is constructed with the static applications in the definition of reify.) We abstract it with shift, generate a residual let expression naming the residual call with a fresh name, and restore the context in the body of the let expression, providing it with the fresh name, appropriately eta-expanded.

$$\uparrow_{t_1 \overset{!}{\to} t_2} e \quad = \quad \overline{\lambda} v_1 : t_1 . \overline{\text{shift}} \, \kappa \, \overline{\text{in}} \, \underline{\text{let}} \, x_2 : t_2 = e \, \underline{@} \downarrow^{t_1} v_1 \, \underline{\text{in}} \, \overline{\text{reset}} \, (\kappa \, \overline{@} \uparrow_{t_2} x_2)$$

This technique of abstracting delimited control in a program transformation is getting to be standard by now. It originates in the specification of "one-pass" CPS transformations [14, 15] and is also used today in continuation-based partial evaluation [6, 29]. We illustrate it further in appendix.

Figure 1 is a conservative extension of the original specification [13] — remembering the algebraic property of reset [14, 15]:

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

In the presence of procedures that may perform side effects, and as illustrated in Section 5, 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

Paulson's Tiny language [34] is a classical example in partial-evaluation circles [4, 7, 10, 27, 32]. Its BNF reads as follows (see Figure 8).

$$\begin{array}{lcl} \langle pgm \rangle & ::= & \textbf{block} \, \langle decl \rangle^* \, \textbf{in} \, \langle cmd \rangle \, \textbf{end} \\ \langle decl \rangle & ::= & \langle ide \rangle^* \end{array}$$

$$
\begin{array}{rcl}
\langle cmd \rangle & ::= & \textbf{skip} \mid \\
 & & \langle cmd \rangle \ \textbf{;} \ \langle cmd \rangle \mid \\
 & & \langle ide \rangle \ \textbf{:=} \ \langle exp \rangle \mid \\
 & & \textbf{if} \ \langle exp \rangle \ \textbf{then} \ \langle cmd \rangle \ \textbf{else} \ \langle cmd \rangle \mid \\
 & & \textbf{while} \ \langle exp \rangle \ \textbf{do} \ \langle cmd \rangle \ \textbf{end} \\
\langle exp \rangle & ::= & \langle int \rangle \mid \langle ide \rangle \mid \langle exp \rangle \ \langle op \rangle \ \langle exp \rangle \mid \textbf{read} \\
\langle op \rangle & ::= & + \mid - \mid \times \mid = \mid \geq
\end{array}
$$

It is a simple exercise to write the corresponding definitional interpreter in direct style (see Figure 3) or in continuation style (see Figure 4). One can then apply it to, *e.g.*, the factorial program

```
block res, val, aux
in val := read ; aux := 1 ;
   while val > 0 do
     aux := aux * val ; val := val - 1
   end ;
   res := aux
end
```

and residualize the result with either of

```
(residualize (meaning-d fac) 'Type-d)
(residualize (meaning-c fac) 'Type-c)
```

where `meaning-d` and `type-d` are defined in Figures 3 and 9, `meaning-c` and `type-c` are defined in Figures 4 and 10, and `fac` denotes the parsed factorial program. Figures 5 and 6 display the corresponding residual programs.

The residual program of Figure 5 is a direct-style Scheme program in A-normal form, threading the store throughout. The residual program of Figure 6 is a continuation-passing Scheme program, also threading the store throughout. In both programs, the while loop has been mapped into a fixed-point declaration (reflecting the semantics of while loops in both Tiny interpreters). All the location offsets have been computed at partial-evaluation time.

The following four facts are worth noting.

1. These residual programs have been generated straight out of the two interpreters of Figures 3 and 4, *i.e.*, with no post-processing.

```
(define meaning-d
  (lambda (p)
    (lambda (add sub mul equ gt read fix true? lookup update)
      (lambda (s)
        (letrec ([meaning-prog ...]
                 [meaning-decl ...]
                 [meaning-comm
                  (lambda (c r s)
                    (case-record c
                      [(Skip) s]
                      [(Sequence c1 c2)
                       (meaning-comm c2 r (meaning-comm c1 r s))]
                      [(Assign i e)
                       (update (r i) (meaning-expr e r s) s)]
                      [(Conditional e c-then c-else) ...]
                      [(While e c)
                       ((fix (lambda (while)
                               (lambda (s)
                                 (true? (meaning-expr e r s)
                                        (lambda (s)
                                          (while
                                            (meaning-comm c r s)))
                                        (lambda (s) s)
                                        s)))) s)]))]
                 [meaning-expr
                  (lambda (e r s)
                    (case-record e
                      [(Literal l) l]
                      [(Boolean b) b]
                      [(Identifier i) (lookup (r i) s)]
                      [(Primop op e1 e2) ((meaning-prim op)
                                          (meaning-expr e1 r s)
                                          (meaning-expr e2 r s))]
                      [(Read) (read)]))]
                 [meaning-prim
                  (lambda (op)
                    (case op
                      [(+) add] [(-) sub] [(*) mul] ...))])
          (meaning-progr p s))))))
```

Figure 3: Direct-style Scheme interpreter for Tiny (valuation functions)

```
(define meaning-c
  (lambda (p)
    (lambda (add sub mul equ gt read fix true? lookup update)
      (lambda (s k)
        (letrec ([meaning-prog ...]
                 [meaning-decl ...]
                 [meaning-comm
                  (lambda (c r s k)
                    (case-record c
                      [(Skip) (k s)]
                      [(Sequence c1 c2)
                       (meaning-comm c1 r s (lambda (s)
                       (meaning-comm c2 r s k)))]
                      [(Assign i e)
                       (meaning-expr e r s (lambda (v)
                       (update (r i) v s k)))]
                      [(Conditional e c-then c-else) ...]
                      [(While e c)
                       ((fix (lambda (while)
                               (lambda (s k)
                                 (meaning-expr e r s (lambda (v)
                                 (true? v
                                        (lambda (s k)
                                          (meaning-comm c r s
                                            (lambda (s)
                                              (while s k))))
                                        (lambda (s k) (k s))
                                       s k))))))) s k)])))]
                 [meaning-expr
                  (lambda (e r s k)
                    (case-record e
                      [(Literal l) (k l)]
                      [(Boolean b) (k b)]
                      [(Identifier i) (lookup (r i) s k)]
                      [(Primop op e1 e2) ...]
                      [(Read) (read k)]))]
                 [meaning-prim ...])
          (meaning-prog p s k))))))
```

Figure 4: Continuation-style Scheme interpreter for Tiny (valuation functions)

```
(lambda (add sub mul equ gt read fix true? lookup update)
  (lambda (s)
    (let* ([n0 (read)]
           [s (update 1 n0 s)]
           [s (update 2 1 s)]
           [s ((fix (lambda (while1)
                      (lambda (s)
                        (let* ([n2 (lookup 1 s)]
                               [n3 (gt n2 0)])
                          (true? n3
                                 (lambda (s)
                                   (let* ([n4 (lookup 2 s)]
                                          [n5 (lookup 1 s)]
                                          [n6 (mul n4 n5)]
                                          [s (update 2 n6 s)]
                                          [n7 (lookup 1 s)]
                                          [n8 (sub n7 1)]
                                          [s (update 1 n8 s)])
                                     (while1 s)))
                                 (lambda (s) s)
                                 s))))) s)]
           [n9 (lookup 2 s)])
      (update 0 n9 s))))
```

This residual program is a specialized version of the Tiny interpreter of Figure 3 with respect to the factorial source program. It is also the textual direct-style version of the residual program of Figure 6.

Figure 5: Direct-style residual factorial program

2. The two interpreters were compiled with an ordinary Scheme compiler, and the residual programs thus were generated without the usual symbolic interpretation of a partial evaluator (generating extensions nonwithstanding).

3. Thanks to the naming scheme of Section 2, both residual programs are also straightforward to read. Specifically, in Figures 9 and 10,

   - the type of expressible values is declared with the name stub `n`, to reflect that the corresponding variables are of integer type;

```
(lambda (add sub mul equ gt read fix true? lookup update)
  (lambda (s k)
    (read (lambda (n0)
    (update 1 n0 s (lambda (s)
    (update 2 1 s (lambda (s)
    ((fix (lambda (while1)
            (lambda (s k)
              (lookup 1 s (lambda (n2)
              (gt n2 0 (lambda (n3)
              (true? n3
                     (lambda (s k)
                       (lookup 2 s (lambda (n4)
                       (lookup 1 s (lambda (n5)
                       (mul n4 n5 (lambda (n6)
                       (update 2 n6 s (lambda (s)
                       (lookup 1 s (lambda (n7)
                       (sub n7 1 (lambda (n8)
                       (update 1 n8 s (lambda (s)
                       (while1 s (lambda (s) (k s))))))))))))))))))
                     (lambda (s k) (k s))
                     s
                     (lambda (s) (k s))))))))) s (lambda (s)
    (lookup 2 s (lambda (n9)
    (update 0 n9 s (lambda (s) (k s))))))))))))))))
```

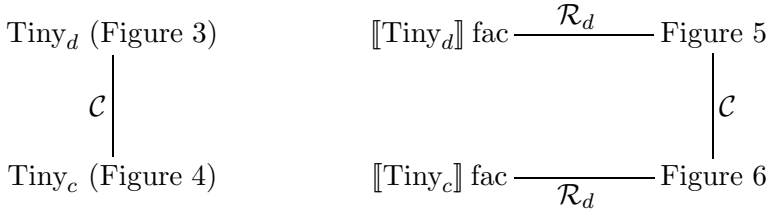
This residual program is a specialized version of the Tiny interpreter of Figure 4 with respect to the factorial source program. It is also the textual CPS version of the residual program of Figure 5.

Figure 6: Continuation-style residual factorial program

- the domain of the semantic operator `fix` is declared with the name stub `while`, to single out the denotation of source while loops;
- the type of the semantic operator `lookup` is declared with an alias, since it is declared globally to the definitional interpreter;
- the types of the store and of the continuation are declared with an alias, since both are single-threaded in the definitional interpreter.

4. Matching the fact that Figure 4 is the CPS counterpart of Figure 3 [16], Figure 6 is the textual CPS counterpart of Figure 5. This property usually holds modulo renaming, using *e.g.*, Schism or Similix [5, 9].

The following diagram summarizes the situation. $\mathcal{R}_d$ denotes the residualizing function of Figure 1. $\mathcal{C}$ denotes the CPS transformation. Tiny_d and Tiny_c denote the text of the direct-style and of the continuation-style Tiny interpreters, respectively. $[\![\text{Tiny}_d]\!]$ and $[\![\text{Tiny}_c]\!]$ denote their meaning (*i.e.*, compiled code). Finally, fac denotes the source factorial program.



6 Corollary: CPS transformation of compiled programs

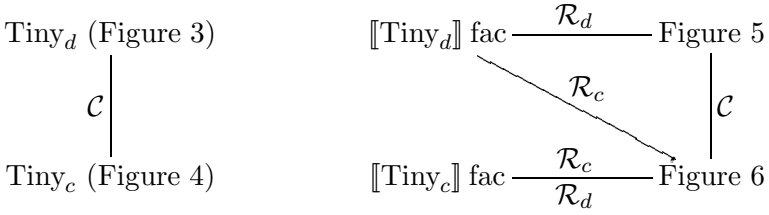
It is very simple to translate nqCPS terms into CPS [12, 23, 28]. Let expressions, for example, in the context of a continuation k, are essentially desugared as follows:

$$\langle\!|\text{let } v = f@x \text{ in } e|\!\rangle\ k \quad = \quad f@x@(\lambda v.\langle\!|e|\!\rangle\ k)$$

This makes it simple to adapt Figure 1 to produce CPS terms. The corresponding program is available through the author's home page.[1] It can be used to perform the following experiment: residualizing the direct-style Tiny interpreter of Figure 3 with respect to the factorial program now yields a continuation-style residual program. This continuation-style residual program textually coincides with the ordinary residualization of the continuation-style Tiny interpreter of Figure 4, provided we relax the alias definition of the compound type `CCont` in Figure 10.

The following diagram extends the diagram of Section 5 and summarizes the situation. $\mathcal{R}_c$ denotes the new residualizing function.

[1] `http://www.brics.dk/~danvy`



In particular, since all types in the continuation-style Tiny interpreter are pure, residualizing it with either $\mathcal{R}_d$ or $\mathcal{R}_c$ yields the same result.

$\mathcal{R}_c$, however, is not the CPS counterpart of $\mathcal{R}_d$. Furthermore, it does not make sense to CPS transform $\mathcal{R}_d$ as defined in Figure 1 and 2 because source programs in general are in direct style.[2] This makes it a true necessity here to abstract delimited control.

7 Related work

7.1 Partial evaluation

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 algorithm for higher-order coercions [25, 26], and also with Berger and Schwichtenberg's normalization algorithm for the simply typed λ-calculus [1]. 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. In particular, and this is the whole point of type-directed partial evaluation, 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.[3]

The two-level λ-calculus has appeared ideal to express the residualization algorithm. Other unexplored developments include subtyping in the two-level λ-calculus [33].

[2] CPS-transforming higher-order programs assumes that their higher-order arguments are also CPS-transformed.

[3] One of the referees 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.

7.2 Logical frameworks

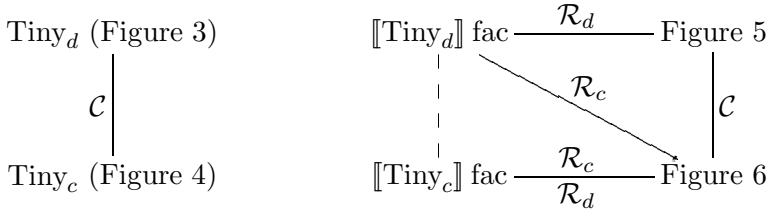
Users of Frank Pfenning's Elf system [35] are also provided with the ability to associate name stubs to types. The reason is the same as here: readability of generated code in the presence of higher-order abstract syntax.

7.3 Out of control: let insertion vs. disjoint sums

In the POPL'96 proceedings, shift and reset are used to handle disjoint sums [13, 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 supersedes the treatment for let insertion. This is thus a case for shift_2 and reset_2 [14]. We leave this aspect for future work.

7.4 An extensional CPS transformation

In his PhD thesis [21], 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 extensional transformation with residualization appears to yield the same effect as the CPS transformation of Section 6. The extensional CPS transformation is dashed in the following diagram, which extends the diagram of Section 6.



8 Conclusion and issues

We have extended type-directed partial evaluation with two pragmatic features: the abilities to have a say in residual identifiers and to insert residual let expressions. These make it possible to improve the readability of residual

programs, to ensure sound call unfolding, and to specialize direct-style programs containing dynamic computational effects. A simple variant makes it possible to generate residual code in CPS.

These simple steps should contribute to make type-directed partial evaluation more practical. Much work remains to formalize it and make it fit with partial evaluation at large.

Acknowledgements

To Andrzej Filinski, Karoline Malmkjær, and René Vestergaard for their interaction and criticism, and to Julia Lawall, Peter Thiemann, and the anonymous referees for perceptive comments.

The diagrams were drawn with Kristoffer Rose's XY-pic package and the Scheme sessions were obtained with R. Kent Dybvig's Chez Scheme system.

A Abstracting control with shift and reset

In the following expression, the special form `Reset` denotes a "prompt" [19], *i.e.*, it delimits the control of its body by supplying it with the identity continuation. The special form `Shift` abstracts this delimited control into a procedure.

```
(+ 1 (Reset (* 10 (Shift k ...))))
```

The abstracted continuation reads `(lambda (v) (* 10 v))`. So for example, the expression

```
(+ 1 (Reset (* 10 (Shift k (+ (k 6) (k 4))))))
```

can also be read as

```
(+ 1 (let ([k (lambda (v) (* 10 v))])
       (+ (k 6) (k 4))))
```

and evaluates to `101`.

In contrast with call/cc, applying an abstracted continuation here does not "jump out" to yield a final answer. It returns a result at its point of application. This functional behavior makes it possible to compose abstracted continuations [14, 20].

```
(define-record (Leaf x))
(define-record (Node left-tree right-tree))

(define flatten
  (lambda (t)     ;;; Binary-Tree(X) -> List(X)
    (letrec ([help (lambda (t)
                     (case-record t
                       [(Leaf x)
                        (Shift k
                               (cons x (Reset (k 'dummy))))]
                       [(Node left right)
                        (begin
                          (help left)
                          (help right))]))])
      (Reset (begin
               (help t)
               '())))))

(define flatten-c
  (lambda (t k)   ;;; (Binary-Tree(X) (List(X) -> Answer)) => Answer
    (letrec ([help (lambda (t k)
                     (case-record t
                       [(Leaf x)
                        (cons x (k 'dummy))]
                       [(Node left right)
                        (help left (lambda (dummy)
                                     (help right k)))]))])
      (k (help t (lambda (dummy) '()))))))
```

Figure 7: Flattening a binary tree

The programming technique used to insert let expressions in Section 4 can be used, for example, to flatten binary trees, as illustrated in Figure 7. The binary tree is traversed depth-first and from left to right, in a delimited context. At every leaf, the traversal is abstracted and the contents of the leaf are cons'ed to the result of the traversal. Procedure `flatten-c` is the CPS counterpart of Procedure `flatten-d`. Notice that even though `flatten-c` is seemingly in "continuation-passing style", it is not tail-recursive. This is the trademark of abstracting delimited control [14, 15, 19].

```
(define-record (Program names command))

(define-record (Skip))
(define-record (Sequence command command))
(define-record (Assign name expression))
(define-record (Conditional expression command command))
(define-record (While expression command))

(define-record (Literal constant))
(define-record (Boolean constant))
(define-record (Identifier name))
(define-record (Primop op expression expression))
(define-record (Read))
```

Figure 8: Abstract syntax for Tiny

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*

```
(define-base-type Int n)
(define-base-type Nat n)
(define-base-type Sto s alias)
(define-compound-type CCont (Sto -!> Sto) k alias)
(define-compound-type Add ((Int Int) =!> Int) add alias)
(define-compound-type Sub ((Int Int) =!> Int) sub alias)
(define-compound-type Mul ((Int Int) =!> Int) mul alias)
(define-compound-type Equ ((Int Int) =!> Int) equ alias)
(define-compound-type Gt ((Int Int) =!> Int) gt alias)
(define-compound-type Read (() =!> Int) read alias)
(define-compound-type While (Sto -!> Sto) while)
(define-compound-type Fix
                      ((While -> Sto -!> Sto) -> Sto -!> Sto)
                      fix alias)
(define-compound-type True?
                      ((Int CCont CCont Sto) =!> Sto) true? alias)
(define-compound-type Lookup ((Nat Sto) =!> Int) lookup alias)
(define-compound-type Update ((Nat Int Sto) =!> Sto) update alias)
(define Type-d
  ((Add Sub Mul Equ Gt Read Fix True? Lookup Update) => Sto -!> Sto))
```

Figure 9: Direct-style Scheme interpreter for Tiny (semantic algebras)

ACM Conference on Lisp and 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.

```
(define-base-type Int n)
(define-base-type Nat n)
(define-base-type Ans)
(define-base-type Sto s alias)
(define-compound-type ECont (Int -> Ans) c alias)
(define-compound-type CCont (Sto -> Ans) k alias)
(define-compound-type Add ((Int Int ECont) => Ans) add alias)
(define-compound-type Sub ((Int Int ECont) => Ans) sub alias)
(define-compound-type Mul ((Int Int ECont) => Ans) mul alias)
(define-compound-type Equ ((Int Int ECont) => Ans) equ alias)
(define-compound-type Gt ((Int Int ECont) => Ans) gt alias)
(define-compound-type Read (ECont -> Ans) read alias)
(define-compound-type While ((Sto CCont) => Ans) while)
(define-compound-type Fix
               ((While -> (Sto CCont) => Ans) -> (Sto CCont) => Ans)
                     fix alias)
(define-compound-type True?
   ((Int ((Sto CCont) => Ans) ((Sto CCont) => Ans) Sto CCont) => Ans)
                     true? alias)
(define-compound-type Lookup ((Nat Sto ECont) => Ans) lookup alias)
(define-compound-type Update
                     ((Nat Int Sto CCont) => Ans)
                     update alias)
(define-compound-type Type-c
  ((Add Sub Mul Equ Gt Read Fix True? Lookup Update) =>
   (Sto CCont) => Ans))
```

Figure 10: Continuation-style Scheme interpreter for Tiny (semantic algebras)

[10] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[11] 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.

[12] 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.

[13] 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*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[14] 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.

[15] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[16] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993.

[17] 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.

[18] 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. To appear in TOPLAS.

[19] 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.

[20] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling

full functional jumps. In Robert (Corky) Cartwright, editor, *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, Snowbird, Utah, July 1988.

[21] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996.

[22] 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.

[23] John Hatcliff. *The Structure of Continuation-Passing Styles*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, June 1994.

[24] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [3], pages 458–471.

[25] 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.

[26] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In Boehm [3], pages 213–226.

[27] 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.

[28] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.

[29] 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.

[30] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In John Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Rapport de recherche N° 2265, INRIA*, pages 112–119, Orlando, Florida, June 1994. Also appears as Technical report CMU-CS-94-129.

[31] Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Bjørner, Ershov, and Jones [2], pages 325–347.

[32] Torben Æ. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, March 1989.

[33] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[34] Larry Paulson. Compiler generation from denotational semantics. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.

[35] 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.

[36] David A. Schmidt. Detecting global variables in denotational definitions. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[37] Peter Sestoft. Automatic call unfolding in a partial evaluator. In Bjørner, Ershov, and Jones [2], pages 485–506.

Recent Publications in the BRICS Report Series

RS-96-15 **Olivier Danvy.** ***Pragmatic Aspects of Type-Directed Partial Evaluation*****. May 1996. 27 pp.**

RS-96-14 **Olivier Danvy and Karoline Malmkjær.** ***On the Idempotence of the CPS Transformation*****. May 1996. 15 pp.**

RS-96-13 **Olivier Danvy and René Vestergaard.** ***Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation*****. May 1996. 28 pp. To appear in** ***8th International Symposium on Programming Languages, Implementations, Logics, and Programs*****, PLILP '96 Proceedings, LNCS, 1996.**

RS-96-12 **Lars Arge, Darren E. Vengroff, and Jeffrey S. Vitter.** ***External-Memory Algorithms for Processing Line Segments in Geographic Information Systems*****. May 1996. 34 pp. A shorter version of this paper appears in Spirakis, editor,** ***Algorithms - ESA '95: Third Annual European Symposium Proceedings*****, LNCS 979, 1995, pages 295–310.**

RS-96-11 **Devdatt Dubhashi, David A. Grable, and Alessandro Panconesi.** ***Near-Optimal, Distributed Edge Colouring via the Nibble Method*****. May 1996. 17 pp. Appears in Spirakis, editor,** ***Algorithms - ESA '95: Third Annual European Symposium Proceedings*****, LNCS 979, 1995, pages 448–459. Invited to be published in a special issue of** ***Theoretical Computer Science*** **devoted to the proceedings of ESA '95.**

RS-96-10 **Torben Braüner and Valeria de Paiva.** ***Cut-Elimination for Full Intuitionistic Linear Logic*****. April 1996. 27 pp. Also available as Technical Report 395, Computer Laboratory, University of Cambridge.**

RS-96-9 **Thore Husfeldt, Theis Rauhe, and Søren Skyum.** ***Lower Bounds for Dynamic Transitive Closure, Planar Point Location, and Parentheses Matching*****. April 1996. 11 pp. Appears in Karlson and Lingas, editors,** ***Algorithm Theory: 5th Scandinavian Workshop*****, SWAT '96 Proceedings, LNCS 1097, 1996, pages 198–211.**