Type-Directed Partial Evaluation

Article *in* Lecture Notes in Computer Science · November 1995

DOI: 10.1145/237721.237784 · Source: CiteSeer

Olivier Danvy
Yale-NUS College
213 PUBLICATIONS 5,018 CITATIONS

Type-Directed Partial Evaluation

Olivier Danvy [*]
Computer Science Department
Aarhus University [†]
(danvy@daimi.aau.dk)

Abstract

We present a strikingly simple partial evaluator, that is type-directed and reifies a compiled program into the text of a residual, specialized program. Our partial evaluator is concise (a few lines) and it handles the flagship examples of off-line monovariant partial evaluation. Its source programs are constrained in two ways: they must be closed and monomorphically typable. Thus dynamic free variables need to be factored out in a "dynamic initial environment". Type-directed partial evaluation uses no symbolic evaluation for specialization, and naturally processes static computational effects.

Our partial evaluator is the part of an offline partial evaluator that residualizes static values in dynamic contexts. Its restriction to the simply typed lambda-calculus coincides with Berger and Schwichtenberg's "inverse of the evaluation functional" (LICS'91), which is an instance of normalization in a logical setting. As such, type-directed partial evaluation essentially achieves lambda-calculus normalization. We extend it to produce specialized programs that are recursive and that use disjoint sums and computational effects. We also analyze its limitations: foremost, it does not handle inductive types.

This paper therefore bridges partial evaluation and λ-calculus normalization through higher-order abstract syntax, and touches upon parametricity, proof theory, and type theory (including subtyping and coercions), compiler optimization, and run-time code generation (including decompilation). It also offers a simple solution to denotational semantics-based compilation and compiler generation.

1 Background and Introduction

Given a source program and parts of its input, a partial evaluator reduces static expressions and reconstructs dynamic expressions, producing a residual, specialized program [15, 36]. To this end, a partial evaluator needs some method for inserting (lifting) arbitrary statically-calculated values into the residual program — *i.e.*, for residualizing them (Section 1.1). We present such a method (Section 1.2); it is type directed and we express it using Nielson and Nielson's two-level λ-calculus [44]. After a first assessment (Section 1.3), we formalize it (Section 1.4) and outline a first application: given a compiled normal form and its type, we can recover its text (Section 1.5). We implement type-directed residualization in Scheme [10] and illustrate it (Section 1.6). The restriction of type-directed residualization to the simply typed λ-calculus actually coincides with Berger and Schwichtenberg's normalization algorithm as presented in the proceedings of LICS'91 [3], and is closely related to Pfenning's normalization algorithm in Elf [46] (Section 1.7). Residualization also exhibits a normalization effect. Moving beyond the pure λ-calculus, we observe that this effect appears for constants and their operators as well (Section 1.8). We harness it to achieve partial evaluation of compiled programs (Section 2). Because this form of partial evaluation is completely directed by type information, we refer to it as *type-directed partial evaluation*.

Section 3 extends type-directed partial evaluation to disjoint sums. Section 4 analyzes the limitations of type-directed partial evaluation. Section 5 reviews related work, and Section 6 concludes.

1.1 The problem

Let us consider the example term

$$(\lambda f.g@(f@d)@f)@\lambda a.a$$

where the infix operator @ denotes function application (and associates, as usual, to the left). Both g and d are unknown, *i.e.*, dynamic. d has type b_1 and g has type $b_1 \to (b_1 \to b_1) \to b_2$, where b_1 and b_2 are dynamic base types. f occurs twice in this term: as a function in an application (where its denotation $\lambda a.a$ could be reduced) and as an argument in a dynamic application (where its denotation $\lambda a.a$ should be residualized). The question is: what are the binding times of this term?

This paper addresses closed terms. Thus let us close the term above by abstracting its two free variables. For added clarity, let us also declare its types.

$$\begin{array}{l}\lambda g : b_1 \to (b_1 \to b_1) \to b_2 .\\ \quad \lambda d : b_1 .(\lambda f : b_1 \to b_1 .g@(f@d)@f)@\lambda a : b_1 .a\end{array}$$

We want to decorate each λ-abstraction and application with static annotations (overlines) and dynamic annotations (underlines) in such a way that static reduction of the decorated

[*]Supported by BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation)

[†]Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
Home page http://www.daimi.aau.dk/~danvy

term "does not go wrong" and yields a completely dynamic term. These are the usual rules of binding-time analysis, which is otherwise abundantly described in the literature [4, 6, 11, 15, 36, 37, 41, 44]. In the rest of this paper, we use Nielson and Nielson's two-level λ-calculus, which is summarized in Appendix A.

Before considering three solutions to analyzing the term above, let us mention a non-solution and why it is a non-solution.

Non-solution:

$$\begin{array}{l} \underline{\lambda} g : b_1 \underline{\to} (b_1 \overline{\to} b_1) \underline{\to} b_2 . \\ \quad \underline{\lambda} d : b_1 . (\overline{\lambda} f : b_1 \overline{\to} b_1 . g \mathbin{\underline{@}} (f \mathbin{\overline{@}} d) \mathbin{\underline{@}} f) \mathbin{\overline{@}} \overline{\lambda} a : b_1 . a \end{array}$$

This annotation is appealing because the application of f is static (and thus will be statically reduced away), but it is incorrect because the type of g is not entirely dynamic. Thus after static reduction, the residual term is not entirely dynamic either:

$$\underline{\lambda} g : b_1 \underline{\to} (b_1 \overline{\to} b_1) \underline{\to} b_2 . \underline{\lambda} d : b_1 . g \mathbin{\underline{@}} d \mathbin{\underline{@}} \overline{\lambda} a : b_1 . a$$

In Scheme, the residual program would contain a closure:

```
> (let* ([g (gensym! "g")] [d (gensym! "d")])
    `(lambda (,g)
       (lambda (,d)
         ,((lambda (f) `((,g ,(f d))
                          ,f))
           (lambda (a) a)))))
(lambda (g15)
  (lambda (d16)
    ((g15 d16) #<procedure>)))
>
```

In summary, the annotation is incorrect because $\lambda a : b_1 . a$ can only be classified to be static (*i.e.*, of type $b_1 \overline{\to} b_1$) if f is always applied. Thus it should be classified to be dynamic (*i.e.*, of type $b_1 \underline{\to} b_1$), as done in Solution 1.

Solution 1:

$$\begin{array}{l} \underline{\lambda} g : b_1 \underline{\to} (b_1 \underline{\to} b_1) \underline{\to} b_2 . \\ \quad \underline{\lambda} d : b_1 . (\overline{\lambda} f : b_1 \underline{\to} b_1 . g \mathbin{\underline{@}} (f \mathbin{\underline{@}} d) \mathbin{\underline{@}} f) \mathbin{\overline{@}} \underline{\lambda} a : b_1 . a \end{array}$$

This solution is correct, but it does not give a satisfactory result: static reduction unfolds the outer call, duplicates the denotation of f, and creates an inner β-redex:

$$\begin{array}{l} \underline{\lambda} g : b_1 \underline{\to} (b_1 \underline{\to} b_1) \underline{\to} b_2 . \\ \quad \underline{\lambda} d : b_1 . g \mathbin{\underline{@}} ((\underline{\lambda} a : b_1 . a) \mathbin{\underline{@}} d) \mathbin{\underline{@}} \underline{\lambda} a : b_1 . a \end{array}$$

To remedy this shortcoming, the source program needs a *binding-time improvement* [36, Chapter 12], *i.e.*, a modification of the source program to make the binding-time analysis yield better results. The particular binding-time improvement needed here is eta-expansion, as done in Solution 2.

Solution 2: Eta-expanding the second occurrence of f makes it always occur in position of application. Therefore $\lambda a : b_1 . a$ can be classified to be static.

$$\begin{array}{l} \underline{\lambda} g : b_1 \underline{\to} (b_1 \underline{\to} b_1) \underline{\to} b_2 . \\ \quad \underline{\lambda} d : b_1 . (\overline{\lambda} f : b_1 \overline{\to} b_1 . g \mathbin{\underline{@}} (f \mathbin{\overline{@}} d) \mathbin{\underline{@}} \boxed{\underline{\lambda} x . f \mathbin{\overline{@}} x}) \mathbin{\overline{@}} \overline{\lambda} a : b_1 . a \end{array}$$

This annotation is correct. Static reduction yields the following term:

$$\underline{\lambda} g : b_1 \underline{\to} (b_1 \underline{\to} b_1) \underline{\to} b_2 . \underline{\lambda} d : b_1 . g \mathbin{\underline{@}} d \mathbin{\underline{@}} \underline{\lambda} a : b_1 . a$$

The result is optimal. It required, however, a binding-time improvement, *i.e.*, human intervention on the source term.

Recent work by Malmkjær, Palsberg, and the author [18, 19] shows that binding-time improvements compensate for the lack of binding-time coercions in existing binding-time analyses. Source eta-expansion, for example, provides a syntactic representation for a binding-time coercion between a higher-order static value and a dynamic context, or conversely between a dynamic value and a static higher-order context.

Binding-time analyses therefore should produce annotated terms that include binding-time coercions, as done in Solution 3. We use a down arrow to represent the coercion of a static (overlined) value into a dynamic (underlined) expression.

Solution 3: In this solution, the coercion of f from $b_1 \overline{\to} b_1$ to $b_1 \underline{\to} b_1$ is written $\downarrow^{b_1 \to b_1} f$:

$$\begin{array}{l} \underline{\lambda} g : b_1 \underline{\to} (b_1 \underline{\to} b_1) \underline{\to} b_2 . \\ \quad \underline{\lambda} d : b_1 . (\overline{\lambda} f : b_1 \overline{\to} b_1 . g \mathbin{\underline{@}} (f \mathbin{\overline{@}} d) \mathbin{\underline{@}} \boxed{\downarrow^{b_1 \to b_1} f}) \mathbin{\overline{@}} \overline{\lambda} a : b_1 . a \end{array}$$

One possibility is to represent the coercion directly with a two-level eta-redex, to make the type structure of the term syntactically apparent [18, 19]. The result of binding-time analysis is then the same as for Solution 2.

Another possibility is to produce the binding-time coercion as such, without committing to its representation, and to leave it to the static reducer to treat this coercion appropriately. *This treatment is the topic of the present paper.*

In Scheme:

```
> (let* ([g (gensym! "g")] [d (gensym! "d")])
    `(lambda (,g)
       (lambda (,d)
         ,((lambda (f) `((,g ,(f d))
                          ,(residualize f '(a -> a))))
           (lambda (a) a)))))
(lambda (g23)
  (lambda (d24)
    ((g23 d24) (lambda (x25) x25))))
>
```

Specifically, this paper is concerned with the residualization of static values in dynamic contexts, in a type-directed fashion. The solution to this seemingly insignificant problem turns out to be widely applicable.

1.2 Type-directed residualization

We want to map a static value into its dynamic counterpart, given its type.

$$\begin{array}{rcl} t \in \text{Type} & ::= & b \mid t_1 \overline{\to} t_2 \mid t_1 \overline{\times} t_2 \\ & & \mid t_1 \underline{\to} t_2 \mid t_1 \underline{\times} t_2 \\ b \in \text{Base-Type} & & \end{array}$$

The leaves of a type are dynamic and ground. In the rest of the paper, types where all constructors are static (resp. dynamic) are said to be "completely static" (resp. "completely dynamic").

At base type, residualization acts as the identity function:

$$\downarrow^{b} v \quad = \quad v$$

Residualizing a value of product type amounts to residualizing each of its components and then reconstructing the product:

$$\downarrow^{t_1 \times t_2} v \quad = \quad \underline{\mathrm{pair}}(\downarrow^{t_1} \overline{\mathrm{fst}}\ v, \downarrow^{t_2} \overline{\mathrm{snd}}\ v)$$

One can infer from Solution 3 that

$$\downarrow^{b_1 \to b_2} v \quad = \quad \underline{\lambda} x_1 . v \mathbin{\overline{@}} x_1$$

(where x_1 is fresh) and more generally that

$$\downarrow^{b \to t} v \quad = \quad \underline{\lambda} x . \downarrow^{t} (v \mathbin{\overline{@}} x).$$

At higher types, the fresh variable needs to be coerced from dynamic to static. A bit of practice with two-level eta-expansion [16, 17, 18] makes it clear that, for example:

$$\downarrow^{(b_3 \to b_4) \to b_2} v \quad = \quad \underline{\lambda} x_1 . v \mathbin{\overline{@}} (\overline{\lambda} x_3 . x_1 \mathbin{\underline{@}} x_3)$$

It is therefore natural to define a function $\uparrow$ that is symmetric to $\downarrow$, *i.e.*, that coerces its argument from dynamic to static, and to define the residualization of functions as follows.

$$\downarrow^{t_1 \to t_2} v \quad = \quad \underline{\lambda} x_1 . \downarrow^{t_2} (v \mathbin{\overline{@}} (\uparrow_{t_1} x_1))$$

The functions $\downarrow$ and $\uparrow$ essentially match the insertion of two-level eta-redexes for binding-time improvements [18, 19]. Figure 1 displays the complete definition of residualization.

Type-directed residualization maps a completely static two-level λ-term into a completely dynamic one. First, reify ($\downarrow$) and reflect ($\uparrow$) fully eta-expand a static two-level λ-term with two-level eta-redexes. Then, static reduction evaluates all the static parts and reconstructs all the dynamic parts, yielding the residual term.

1.3 Assessment

So far, we have seen (1) the need for binding-time coercions at higher types in a partial evaluator; (2) the fact that these binding-time coercions can be represented as two-level eta-redexes; and (3) an interpreter for these binding-time coercions — *i.e.*, type-directed residualization.

Let us formalize type-directed residualization, and then describe a first application.

1.4 Formalization

Proposition 1 *In Figure 1, reify maps a simply typed completely static λ-term into a well-typed two-level λ-term.*

Proof: by structural induction on the types (see Appendix A for the notion of well-typing).

Property 1 *In the simply typed case, static reduction in the two-level λ-calculus enjoys both strong normalization and subject reduction [44].*

Corollary 1 *Static reduction after reification (see Figure 1) does not go wrong and yields completely dynamic terms.*

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

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

$$\begin{array}{rcl}
\mathrm{reflect} & = & \lambda t . \lambda e : t . \uparrow_{t} e \\
 & : & \mathrm{Type} \to \mathrm{Expr} \to \mathrm{TLT} \\
\uparrow_{b} e & = & e \\
\uparrow_{t_1 \to t_2} e & = & \overline{\lambda} v_1 . \uparrow_{t_2} (e \mathbin{\underline{@}} (\downarrow^{t_1} v_1)) \\
\uparrow_{t_1 \times t_2} e & = & \overline{\mathrm{pair}}(\uparrow_{t_1} \underline{\mathrm{fst}}\, e, \uparrow_{t_2} \underline{\mathrm{snd}}\, e)
\end{array}$$

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

Since the definition of type-directed residualization is based solely on the structure of types, we have omitted type annotations.

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. 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.

N.B. One could be tempted to define

$$\text{“eval”} \ = \ \text{dynamically-reduce} \circ \mathrm{reflect}$$

by symmetry, but the result is not an evaluation functional because base types are dynamic.

Figure 1: Type-directed residualization

Corollary 2 *The type of a source term and the type of a residual term have the same shape (i.e., erasing their annotations yields the same simple type).*

This last property extends to terms that are in long $\beta\eta$-normal form [30], *i.e.*, to normal forms that are completely eta-expanded. By Proposition 1, we already know that static reduction of a completely static term in long $\beta\eta$-normal form yields a completely dynamic term in long $\beta\eta$-normal form We have independently proven the following proposition.

```
(define-record (Base base-type))
(define-record (Func domain range))
(define-record (Prod type type))

(define residualize
  (lambda (v t)
    (letrec ([reify (lambda (t v)
                      (case-record t
                        [(Base -)
                         v]
                        [(Func t1 t2)
                         (let ([x1 (gensym!)])
                           `(lambda (,x1) ,(reify t2 (v (reflect t1 x1)))))]
                        [(Prod t1 t2)
                         `(cons ,(reify t1 (car v)) ,(reify t2 (cdr v)))]))]
             [reflect (lambda (t e)
                        (case-record t
                          [(Base -)
                           e]
                          [(Func t1 t2)
                           (lambda (v1) (reflect t2 `(,e ,(reify t1 v1))))]
                          [(Prod t1 t2)
                           (cons (reflect t1 `(car ,e)) (reflect t2 `(cdr ,e)))]))])
      (begin
        (reset-gensym!)
        (reify (parse-type t) v)))))
```

Figure 2: Type-directed partial evaluation in Scheme

Proposition 2 *Residualizing a completely static term in long $\beta\eta$-normal form yields a term with the same shape (i.e., erasing the annotations of both terms yields the same simply typed λ-term, modulo α-renaming).*

In other words, residualization preserves both the shape of types and the shape of expressions that are in long $\beta\eta$-normal form.

1.5 Application

We can represent a completely static expression with a compiled representation of this expression, and a completely dynamic expression with a (compiled) program constructing the textual representation of this expression.[1] In Consel's partial evaluator Schism, for example, this representation is used to optimize program specialization [13]. Since (1) reification amounts to mapping this static expression into a two-level term and (2) static reduction amounts to running both the static and the dynamic components of this two-level term, type-directed residualization constructs the textual representation of the original static expression. Therefore, in principle, we can map a compiled program back into its text — under the restrictions that (1) this program terminates, and (2) it has a type.

The following section illustrates this application.

1.6 Type-directed residualization in Scheme

Figure 2 displays an implementation of type-directed residualization in Scheme, using a syntactic extension for declaring and using records [10]. Procedure `parse-type` maps the concrete syntactic representation of a type (an S-expression) into the corresponding abstract syntactic representation (a nested record structure). For example, `'((A * B) -> C)` is mapped into `(make-Func (make-Prod (make-Base 'A) (make-Base 'B)) (make-Base 'C))`.

The following Scheme session illustrates this implementation:

```
> (define S (lambda (f)
              (lambda (g)
                (lambda (x)
                  ((f x) (g x))))))
> S
#<procedure S>
> (residualize S
               '((A -> B -> C) -> (A -> B) -> A -> C))
(lambda (x0)
  (lambda (x1)
    (lambda (x2)
      ((x0 x2) (x1 x2)))))
> (define I*K (cons (lambda (x) x)
                    (lambda (y) (lambda (z) y))))
> I*K
(#<procedure> . #<procedure>)
> (residualize I*K '((A -> A) * (B -> C -> B)))
(cons (lambda (x0) x0)
      (lambda (x1) (lambda (x2) x1)))
>
```

`S` and `I*K` denote values that we wish to residualize. We know their type. Procedure `residualize` maps these (static, compiled) values and a representation of their type into the corresponding (dynamic, textual) representation of these values.

At this point, a legitimate question arises: *how does this really work?* Let us consider the completely static expression

$$\overline{\lambda} x.x$$

together with the type $b \to b$. This expression is mapped into the following eta-expanded term:

$$\underline{\lambda} z.(\overline{\lambda} x.x)\,\overline{@}\,z.$$

[1]The same situation occurs with interpreted instead of compiled representations, i.e., if one uses an interpreter instead of a compiler

Static β-reduction yields the completely dynamic residual term

$$\underline{\lambda} z.z$$

which constructs the text of the static expression we started with.

Similarly, the S combinator is mapped into the term

$$\underline{\lambda} a.\underline{\lambda} b.\underline{\lambda} c.S \,\overline{@}\, (\overline{\lambda} d.\overline{\lambda} e.(a \,\underline{@}\, d) \,\underline{@}\, e) \,\overline{@}\, (\overline{\lambda} f.b \,\underline{@}\, f) \,\overline{@}\, c$$

which statically β-reduces to the completely dynamic residual term

$$\underline{\lambda} a.\underline{\lambda} b.\underline{\lambda} c.(a \,\underline{@}\, c) \,\underline{@}\, (b \,\underline{@}\, c).$$

Let us conclude with a remark: because residual terms are eta-expanded, refining the type parameter yields different residual programs, as in the following examples.

```
> (residualize (lambda (x) x) '((a * b) -> a * b))
(lambda (x0) (cons (car x0) (cdr x0)))
> (residualize (lambda (x) x)
               '(((A -> B) -> C) -> (A -> B) -> C))
(lambda (x0) (lambda (x1) (x0 (lambda (x2) (x1 x2)))))
>
```

1.7 Strong normalization

The algorithm of type-directed residualization is actually well known.

In their paper "An Inverse of the Evaluation Functional for Typed λ-Calculus" [3], Berger and Schwichtenberg present a normalization algorithm for the simply typed λ-calculus. It is used for normalizing proofs as programs.

Berger and Schwichtenberg's algorithm coincides with the restriction of type-directed residualization to the simply typed λ-calculus. Reify maps a (semantic, meta-level) value and its type into a (syntactic, object-level) representation of this value ("syntactic" in the sense of "abstract-syntax tree"), and conversely, reflect maps a syntactic representation into the corresponding semantic value. Disregarding the dynamic base types, reflect thus acts as an evaluation functional, and reify acts as its inverse — hence probably the title of Berger and Schwichtenberg's paper [3].

In the implementation of his Elf logic programming language [46], Pfenning uses a similar normalization algorithm to test extensional equality, though with no static/dynamic notion and also with the following difference. When processing an arrow type whose co-domain is itself an arrow type, the function is processed *en bloc* with all its arguments:

$$\downarrow^{t_1 \to t_2 \to \quad \to t_{n+1}} v =$$
$$\lambda x_1.\lambda x_2.\ldots\lambda x_n.\downarrow^{t_{n+1}}(v@(\uparrow_{t_1} x_1)@(\uparrow_{t_2} x_2)@\ldots@(\uparrow_{t_n} x_n))$$

where t_{n+1} is not an arrow type and x_1, ..., x_n are fresh. (The algorithm actually postpones reflection until reification reaches a base type.)

Residualization also exhibits a normalization effect, as illustrated below: we residualize the result of applying a procedure to an argument. This program contains an application and *this application is performed at residualization time.*[2]

```
> (define foo (lambda (f) (lambda (x) (f x))))
> (residualize (foo (lambda (z) z)) '(A -> A))
(lambda (x0) x0)
>
```

[2]Or, viewing residualization as a form of decompilation (an analogy due to Goldberg [27]), "at decompile time"

The same legitimate question as before arises: *how does this really work?* Let us consider the completely static expression

$$(\overline{\lambda} f.\overline{\lambda} x.f \,\overline{@}\, x) \,\overline{@}\, (\overline{\lambda} z.z).$$

This expression is mapped into the following eta-expanded term:

$$\underline{\lambda} y.((\overline{\lambda} f.\overline{\lambda} x.f \,\overline{@}\, x) \,\overline{@}\, (\overline{\lambda} z.z)) \,\overline{@}\, y.$$

Static β-reductions yield the completely dynamic residual term

$$\underline{\lambda} y.y.$$

As an exercise, the curious reader might want to run the residualizer on the term $S \,\overline{@}\, K \,\overline{@}\, K$ with respect to the type $b \to b$. The combinators S and K are defined as above [2, Definition 5.1.8, Item (i)].

1.8 Beyond the pure λ-calculus

Moving beyond the pure λ-calculus, let us reiterate this last experiment: we residualize the result of applying a procedure to an argument, and *a multiplication is computed at residualization time.*

```
> (define bar (lambda (x) (lambda (k) (k (* x 5)))))
> (residualize (bar 100) '((Int -> Ans) -> Ans))
(lambda (x0) (x0 500))
>
```

The usual legitimate question arises: *how does this really work?* Let us consider the completely static expression

$$(\overline{\lambda} x.\overline{\lambda} k.k \,\overline{@}\, (x \,\overline{\times}\, 5)) \,\overline{@}\, 100.$$

This expression is mapped it into the following eta-expanded term:

$$\underline{\lambda} a.((\overline{\lambda} x.\overline{\lambda} k.k \,\overline{@}\, (x \,\overline{\times}\, 5)) \,\overline{@}\, 100) \,\overline{@}\, (\overline{\lambda} n.a \,\underline{@}\, n)$$

Static β-reduction leads to

$$\underline{\lambda} a.a \,\underline{@}\, (100 \,\overline{\times}\, 5)$$

which is statically δ-reduced to the residual term $\underline{\lambda} a.a \,\underline{@}\, 500$.

Remark: Introducing a static fixed-point operator does not compromise the subject-reduction property, so the second part of Corollary 1 in Section 1.4 can be rephrased with the proviso "if static reduction terminates".

1.9 This paper

The fact that arbitrary static reductions can occur at residualization time suggests that residualization can be used as a full-fledged partial evaluator for closed compiled programs, given their type. In the following section, we apply it to various examples that have been presented as typical or even significant achievements of partial evaluation, in the literature [15, 33, 36]. These examples include the power and the format source programs, and interpreters for Paulson's imperative language Tiny and for the λ-calculus.

The presentation of each example is structured as follows:

- we consider interpreter-like programs, *i.e.*, programs where one argument determines a part of the control flow (Abelson, [24, Foreword]);

```
> (define power
    (lambda (x n)
      (letrec ([loop (lambda (n)
                       (cond [(zero? n) 1]
                             [(odd? n) (* x (loop (1- n)))]
                             [else (sqr (loop (/ n 2)))]))])
        (loop n))))
> (define sqr (lambda (x) (* x x)))
> (power 2 10)
1024
> (define power-abstracted     ;;; Int -> (Int -> Int) * (Int * Int => Int) => Int -> Int
    (lambda (n)
      (lambda (sqr *)
        (lambda (x)
          (letrec ([loop (lambda (n)
                           (cond [(zero? n) 1]
                                 [(odd? n) (* x (loop (1- n)))]
                                 [else (sqr (loop (/ n 2)))]))])
            (loop n))))))
> (((power-abstracted 10) sqr *) 2)
1024
> (residualize (power-abstracted 10) '((Int -> Int) * (Int * Int => Int) => Int -> Int))
(lambda (x0 x1) (lambda (x2) (x0 (x1 x2 (x0 (x0 (x1 x2 1)))))))
> (((lambda (x0 x1) (lambda (x2) (x0 (x1 x2 (x0 (x0 (x1 x2 1))))))) sqr *) 2)
1024
>
```

The residualized code reads better after α-renaming. It is the specialized version of `power` when `n` is set to 10:

```
(lambda (sqr *)
  (lambda (x)
    (sqr (* x (sqr (sqr (* x 1)))))))
```

N.B. For convenience, our implementation of `residualize`, unlike the simpler version shown in Figure 2, handles Scheme-style uncurried n-ary procedures. Their types are indicated in type expressions by "=>" preceded by the n-ary product of the argument types.

Figure 3: Type-directed partial evaluation of power (an interactive session with Scheme)

- we residualize the result of applying these (separately compiled) programs to the corresponding argument.

Because residualization is type-directed, we need to know the type of the free variables in the residual program. We will routinely abstract them in the source program, as a form of "initial run-time environment", hence making the residual program a closed λ-term.

2 Type-Directed Partial Evaluation

The following examples illustrate that residualization yields specialized programs, under the condition that the residual program is a simply typed combinator — *i.e.*, with no free variables and with a simple type. The static parts of the source program, however, are less constrained than when using a partial evaluator: they can be untyped and impure. In that sense it is symmetric to a partial evaluator such as Gomard and Jones's λ-Mix [35, 36] that allows dynamic computations to be untyped but requires static computations to be typed.[3] In any case, residualization produces the same result as conventional partial evaluation (*i.e.*, a specialized program) but is naturally more efficient since no program analysis other than type inference and no symbolic interpretation take place.

[3] λ-Mix and type-directed partial evaluation both consider closed source programs They work alike for typed source programs whose binding times have been improved by source eta-expansion

2.1 Power

Figure 3 displays the usual definition of the `power` procedure in Scheme, and its abstracted counterpart where we have factored out the residual operators `sqr` and `*`. The figure illustrates that residualizing the partial application of `power` to an exponent yields the specialized version of `power` with respect to this exponent.

2.2 Format

For lack of space, we omit the classical example of partial evaluation: formatting strings. Its source code can be found in Figure 1 of Consel and Danvy's tutorial notes on partial evaluation at POPL'93 [15]. Type-directed partial evaluation yields *the same residual code* as the one presented in the tutorial notes (modulo of course the factorization of the residual operators `write-string`, `write-number`, `write-newline`, and `list-ref`).

2.3 Definitional interpreter for Paulson's Tiny language

Recursive procedures can be defined with fixed-point operators. This makes it simple to residualize recursive procedures — by abstracting their (typed) fixed-point operator.

As an example, let us consider Paulson's Tiny language [45], which is a classical example in partial evaluation [6, 8,

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

Figure 4: Source factorial program

```
(lambda (add sub mul eq gt read fix true? lookup update)
  (lambda (k8)
    (lambda (s9)
      (read (lambda (v10)
      (update 1 v10 s9 (lambda (s11)
      (update 2 1 s11 (lambda (s12)
      ((fix (lambda (while)
              (lambda (s14)
                (lookup 1 s14 (lambda (v15)
                (gt v15 0 (lambda (v16)
                (true? v16
                       (lambda (s17)
                         (lookup 2 s17 (lambda (v18)
                         (lookup 1 s17 (lambda (v19)
                         (mul v18 v19 (lambda (v20)
                         (update 2 v20 s17 (lambda (s21)
                         (lookup 1 s21 (lambda (v22)
                         (sub v22 1 (lambda (v23)
                         (update 1 v23 s21 (lambda (s24)
                         (while s24)))))))))))))))))
                       (lambda (s25)
                         (lookup 2 s25 (lambda (v26)
                         (update 0 v26 s25 (lambda (s27)
                         (k8 s27))))))
                       s14)))))))
       s12))))))))))
```

This residual program is a specialized version of the Tiny interpreter (Figures 9 and 10) with respect to the source program of Figure 4. As can be observed, it is a continuation-passing Scheme program threading the store throughout. The while loop of Figure 4 has been mapped into a fixed-point declaration. All the location offsets have been computed at partial-evaluation time.

Figure 5: Residual factorial program (after α-renaming and "pretty" printing)

```
(define instantiate-type
  (lambda (t)
    `((((() => Exp) -> Exp) *  ;;; reset-gensym-c
      ((Str -> Exp) -> Exp) *  ;;; gensym-c
      (Exp -> Exp) *           ;;; unparse-expression
      (Str -> Var) *           ;;; make-Var
      (Str * Exp => Exp) *     ;;; make-Lam
      (Exp * Exp => Exp) *     ;;; make-App
      (Exp * Exp => Exp) *     ;;; make-Pair
      (Exp -> Exp) *           ;;; make-Fst
      (Exp -> Exp)             ;;; make-Snd
      => ,t
      -> Exp)))
```

Figure 6: Type construction for self-application

9, 11, 14, 35, 36, 37, 41, 48]:

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

It is a simple exercise (see Figures 9 and 10 in appendix) to write the corresponding definitional interpreter, to apply it to, *e.g.*, the factorial program (Figure 4), and to residualize the result (Figure 5).

Essentially, type-directed partial evaluation of the Tiny interpreter acts as a front-end compiler that maps the abstract syntax of a source program into a λ-expression representing the dynamic semantics of this program [14]. This λ-expression is in continuation-passing style [49], *i.e.*, in three-address code.

We have extended the definitional λ-interpreter described in this section to richer languages, including typed higher-order procedures, block structure, and subtyping, *à la* Reynolds [47]. Thus this technique of "type-directed compilation" scales up in practice. In that sense, type-directed partial evaluation provides a simple and effective solution to (denotational) semantics-directed compilation in the λ-calculus [32, 43].

2.4 Residualizing the residualizer

To visualize the effect of residualization, one can residualize the residualizer with respect to a type. As a first approximation, given a type t, we want to evaluate

```
(residualize
  (lambda (v) (residualize v t))
  t)
```

To this end, we first need to define an abstracted version of the residualizer (with no free variables). We need to factor out all the abstract-syntax constructors, the unparser,[4] and the gensym paraphernalia, which we make continuation-passing to ensure that new symbols will be generated correctly at run time. To be precise:

```
(define abstract-residualize
  (lambda (t)
    (lambda (reset-gensym-c gensym-c
             unparse-expression
             make-Var make-Lam make-App
             make-Pair make-Fst make-Snd)
      (lambda (v)
        (letrec ([reify ...]       ;;; as in
                 [reflect ...])    ;;; Figure 2
          (reset-gensym-c
            (lambda ()
              (unparse-expression
                (reify (parse-type t) v)))))))))
```

The type of `abstract-residualize` is a dependent type in that the value of `t` denotes a representation of the type of `v`. Applying `abstract-residualize` to a represention of a type, however, yields a simply typed value. We can then write a

[4]Figure 2 uses quasiquote and unquote for readability, thus avoiding the need for an unparser

```
> (define meaning-expr-cps-cbv
    (lambda (e)
      (letrec ([meaning (lambda (e r)
                          (lambda (k)
                            (case-record e
                              [(Var i)
                               (k (r i))]
                              [(Lam i e)
                               (k (lambda (v)
                                    (meaning e (lambda (i v r) (lambda (j) (if (equal? i j) v (r j)))))))))]
                              [(App e0 e1)
                               ((meaning e0 r) (lambda (v0)
                                                 ((meaning e1 r) (lambda (v1)
                                                                   ((v0 v1) k)))))])))])
        (meaning (parse-expression e) (lambda (i) (error 'init-env "undeclared identifier: ~s" i))))))
> (define meaning-type-cps-cbv
    (lambda (t)
      (letrec ([computation (lambda (t)
                              (make-Func (make-Func (value t) (make-Base 'Ans)) (make-Base 'Ans)))]
               [value (lambda (t)
                        (case-record t
                          [(Base -)
                           t]
                          [(Func t1 t2)
                           (make-Func (value t1) (computation t2))]))])
        (unparse-type (computation (parse-type t))))))
> (residualize (meaning-expr-cps-cbv '(lambda (x) x)) (meaning-type-cps-cbv '(a -> a)))
(lambda (x0) (x0 (lambda (x1) (lambda (x2) (x2 x1)))))
>
```

N.B. The interpreter is untyped and thus we can only residualize interpreted terms that are closed and simply typed. Untyped or polymorphically typed terms, for example, are out of reach.

Figure 7: Type-directed partial evaluation of a call-by-value CPS λ-interpreter

procedure `instantiate-type` that maps the representation of the input type to a representation of the type of that simply typed value (see Figure 6).

We are now ready for self-application with respect to a type `t`:

```
(residualize
  (abstract-residualize (instantiate-type t))
  t)
```

The result is the text of a Scheme procedure. Applying this procedure to the initial environment of the residualizer (*i.e.*, the abstract-syntax constructors, *etc.*) and then to a compiled version of an expression of type `t` yields the text of that expression.

Self-application eliminates the overhead of interpreting the type of a source program.

For example, let us consider the S combinator of Section 1.6. Residualizing the residualizer with respect to its type essentially yields the eta-expanded two-level version we wrote in Section 1.6 to visualize the residualization of S.

For another example, we can consider the Tiny interpreter of Section 2.3. Residualizing the residualizer with respect to its type (see Figure 9) yields the text of a Tiny compiler (whose run-time support includes the Tiny interpreter).

2.5 The art of the λ-interpreter

We consider various λ-interpreters and residualize their application to a λ-term. The running question is as follows: which type should drive residualization?

Direct style: For a direct-style interpreter, the type is the same as the type of the interpreted λ-term and the residual term is structurally equivalent to the interpreted λ-term [36, Section 7.4].

Continuation-passing style: For a continuation-style interpreter, the type is the CPS counterpart of the type of the interpreted λ-term and the residual term is the CPS counterpart of the interpreted λ-term — for each possible continuation-passing style [28]. Figure 7 illustrates the point for left-to-right call-by-value.

Other passing styles: The same technique applies for store-passing. *etc.* interpreters, be they direct or continuation-passing, and in particular for interpreters that simulate lazy evaluation with thunks.

"Monadic" style: We cannot, however, specialize a "monadic" interpreter with respect to a source program because the residual program is parameterized with polymorphic functions [42, 50] and these polymorphic functions do not have a simple type. Thus monadic interpreters provide an example where traditional partial evaluation wins over type-directed partial evaluation.

2.6 Static computational effects

It is simple to construct a program that uses computational effects (assignments, I/O, or `call/cc`) statically, and that type-directed partial evaluation specializes successfully — something that comes for free here but that (for better or for worse) no previous partial evaluator does. We come back to this point in Section 4.4.

3 Disjoint Sums

Let us extend the language of Figure 1 with disjoint sums and booleans. (Booleans are included for pedagogical value.)

Reifying a disjoint-sum value is trivial:

$$\begin{array}{lcl} \downarrow^{t_1+t_2} v & = & \overline{\text{case}}\; v\; \overline{\text{of}}\; \underline{\text{inleft}}(v_1) \Rightarrow \underline{\text{inleft}}(\downarrow^{t_1} v_1) \\ & & \qquad\quad [] \; \underline{\text{inright}}(v_2) \Rightarrow \underline{\text{inright}}(\downarrow^{t_2} v_2) \\ & & \overline{\text{end}} \end{array}$$

Reflecting upon a disjoint-sum expression is more challenging. By symmetry, we would like to write

$$\begin{array}{lcl} \uparrow_{t_1+t_2} e & = & \underline{\text{case}}\; e\; \underline{\text{of}}\; \underline{\text{inleft}}(x_1) \Rightarrow \overline{\text{inleft}}(\uparrow_{t_1} x_1) \\ & & \qquad\quad [] \; \underline{\text{inright}}(x_2) \Rightarrow \overline{\text{inright}}(\uparrow_{t_2} x_2) \\ & & \underline{\text{end}} \end{array}$$

(where x_1 and x_2 are fresh) but this would yield ill-typed two-level λ-terms, as in the non-solution of Section 1.1. Static values would occur in conditional branches and dynamic conditional expressions would occur in static contexts — a clash at higher types.

The symmetric definition requires us to supply the context of reflection (which is expecting a static value) both with an appropriate left value and an appropriate right value, and then to construct the corresponding residual case expression. Unless the source term is tail-recursive, we thus need to abstract and to relocate this context.

Context abstraction is achieved with a control operator. This context, however, needs to be delimited, which rules out `call/cc` [10] but invites one to use `shift` and `reset` [16, 17] (though of course any other delimited control operator could do as well [21]).[5] The extended residualizer is displayed in Figure 8.

The following Scheme session illustrates this extension.

```
> (residualize (lambda (x) x) '((A + B) -> (A + B)))
(lambda (x0) (case-record x0
               [(Left x1) (make-Left x1)]
               [(Right x2) (make-Right x2)]))
> (residualize (lambda (x) 42) '(Bool -> Int))
(lambda (x0) (if x0 42 42))
> (residualize
    (lambda (call/cc fix null? zero? * car cdr)
      (lambda (xs)
        (call/cc
          (lambda (k)
            ((fix (lambda (m)
                    (lambda (xs)
                      (if (null? xs)
                          1
                          (if (zero? (car xs))
                              (k 0)
                              (* (car xs)
                                 (m (cdr xs))))))))
             xs)))))
```

[5] An overview of shift and reset can be found in Appendix B

$$\begin{array}{rcl} t \in \text{Type} & ::= & b \mid t_1 \to t_2 \mid t_1 \times t_2 \mid t_1 + t_2 \mid \text{Bool} \\ v \in \text{Value} & ::= & c \mid x \mid \overline{\lambda} x : t.v \mid v_0 \,\overline{@}\, v_1 \mid \\ & & \overline{\text{pair}}(v_1, v_2) \mid \overline{\text{fst}}\; v \mid \overline{\text{snd}}\; v \mid \\ & & \overline{\text{inleft}}(v) \mid \overline{\text{inright}}(v) \mid \\ & & \overline{\text{case}}\; v\; \overline{\text{of}}\; \underline{\text{inleft}}(x_1) \Rightarrow v_1 \\ & & \qquad\quad [] \; \underline{\text{inright}}(x_2) \Rightarrow v_2 \\ & & \overline{\text{end}} \\ e \in \text{Expr} & ::= & c \mid x \mid \underline{\lambda} x : t.e \mid e_0 \,\underline{@}\, e_1 \mid \\ & & \underline{\text{pair}}(e_1, e_2) \mid \underline{\text{fst}}\, e \mid \underline{\text{snd}}\, e \mid \\ & & \underline{\text{inleft}}(e) \mid \underline{\text{inright}}(e) \mid \\ & & \underline{\text{case}}\; e\; \underline{\text{of}}\; \underline{\text{inleft}}(x_1) \Rightarrow e_1 \\ & & \qquad\quad [] \; \underline{\text{inright}}(x_2) \Rightarrow e_2 \\ & & \underline{\text{end}} \end{array}$$

$$\begin{array}{rcl} \text{reify} & = & \lambda t.\lambda v : t.\downarrow^{t} v \\ & : & \text{Type} \to \text{Value} \to \text{TLT} \\ \downarrow^{b} v & = & v \\ \downarrow^{t_1 \to t_2} v & = & \underline{\lambda} x_1.\overline{\text{reset}}_{t_2} \downarrow^{t_2} (v \,\overline{@} \uparrow^{t_2}_{t_1} x_1) \\ & & \text{where } x_1 \text{ is fresh.} \\ \downarrow^{t_1 \times t_2} v & = & \underline{\text{pair}}(\downarrow^{t_1} \overline{\text{fst}}\; v, \downarrow^{t_2} \overline{\text{snd}}\; v) \\ \downarrow^{t_1+t_2} v & = & \overline{\text{case}}\; v\; \overline{\text{of}} \\ & & \quad \overline{\text{inleft}}(v_1) \Rightarrow \underline{\text{inleft}}(\downarrow^{t_1} v_1) \\ & & [] \; \overline{\text{inright}}(v_2) \Rightarrow \underline{\text{inright}}(\downarrow^{t_2} v_2) \\ & & \overline{\text{end}} \\ \downarrow^{\text{Bool}} v & = & \overline{\text{if}}\; v\; \overline{\text{then}}\; \text{true}\; \overline{\text{else}}\; \text{false} \end{array}$$

$$\begin{array}{rcl} \text{reflect} & = & \lambda t'.\lambda t.\lambda e : t.\uparrow^{t'}_{t} e \\ & : & \text{Type} \to \text{Type} \to \text{Expr} \to \text{TLT} \\ \uparrow^{t}_{b} e & = & e \\ \uparrow^{t}_{t_1 \to t_2} e & = & \overline{\lambda} v_1.\uparrow^{t}_{t_2} (e \,\underline{@} \downarrow^{t_1} v_1) \\ \uparrow^{t}_{t_1 \times t_2} e & = & \overline{\text{pair}}(\uparrow^{t}_{t_1} \underline{\text{fst}}\, e, \uparrow^{t}_{t_2} \underline{\text{snd}}\, e) \\ \uparrow^{t}_{t_1+t_2} e & = & \overline{\text{shift}}\; \kappa : t_1 + t_2 \to t \\ & & \overline{\text{in}}\; \underline{\text{case}}\; e\; \underline{\text{of}} \\ & & \qquad \underline{\text{inleft}}(x_1) \Rightarrow \overline{\text{reset}}_t\; (\kappa \,\overline{@}\, \overline{\text{inleft}}(\uparrow^{t}_{t_1} x_1)) \\ & & \quad [] \; \underline{\text{inright}}(x_2) \Rightarrow \overline{\text{reset}}_t\; (\kappa \,\overline{@}\, \overline{\text{inright}}(\uparrow^{t}_{t_2} x_2)) \\ & & \quad \underline{\text{end}} \\ & & \text{where } x_1 \text{ and } x_2 \text{ are fresh.} \\ \uparrow^{t}_{\text{Bool}} e & = & \overline{\text{shift}}\; \kappa : \text{Bool} \to t \\ & & \overline{\text{in}}\; \underline{\text{if}}\; e \\ & & \quad \underline{\text{then}}\; \overline{\text{reset}}_t\; (\kappa \,\overline{@}\, \text{true}) \\ & & \quad \underline{\text{else}}\; \overline{\text{reset}}_t\; (\kappa \,\overline{@}\, \text{false}) \end{array}$$

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 8: Type-directed residualization

```
     '((((Num -> Num) -> Num) -> Num) *
       (((LNum -> Num) -> LNum -> Num) -> LNum -> Num) *
       (LNum -> Bool) * (Num -> Bool) *
       (Num * Num => Num) *
       (LNum -> Num) * (LNum -> LNum) => LNum -> Num))
(lambda (x0 x1 x2 x3 x4 x5 x6)
  (lambda (x7)
    (x0 (lambda (x8)
          ((x1 (lambda (x9)
                 (lambda (x10)
                   (if (x2 x10)
                       1
                       (if (x3 (x5 x10))
                           (x8 0)
                           (x4 (x5 x10)
                               (x9 (x6 x10))))))))
           x7)))))
>
```

In the first interaction, the identity procedure over a disjoint sum is residualized. In the second interaction, a constant procedure is residualized. The third interaction features a standard example in the continuations community: a procedure that multiplies numbers in a list, and escapes if it encounters zero. Residualization requires both the type of `fix` (to traverse the list) and of `call/cc` (to escape).

The same legitimate question as in Section 1 arises: *how does this really work?* Let us residualize the static application $f \overline{@} g$ with respect to the type Bool $\rightarrow$ Int, where

$$\begin{array}{rcl} f & = & \overline{\lambda} h.\overline{\lambda} x.1 + h \overline{@} x \\ g & = & \overline{\lambda} y.\overline{\text{if}}\ y\ \overline{\text{then}}\ 2\ \overline{\text{else}}\ 3 \end{array}$$

We want to perform the addition in f statically. This requires us to reduce the conditional expression in g, even though g's argument is unknown. During residualization, the delimited context $[f \overline{@} g \overline{@} [\cdot]]$ is abstracted and relocated in both branches of a dynamic conditional expression:

$$\downarrow^{\text{Bool}\rightarrow\text{Int}}(f \overline{@} g) =$$

$$\underline{\lambda} b.\overline{\text{reset}}_{\text{Int}}(\downarrow^{\text{Int}}(f \overline{@} g \overline{@} (\uparrow^{\text{Int}}_{\text{Bool}}\ b))) =$$

$$\underline{\lambda} b.\underline{\text{if}}\ b\ \underline{\text{then}}\ \overline{\text{reset}}_{\text{Int}}(f \overline{@} g \overline{@} \text{true})\ \underline{\text{else}}\ \overline{\text{reset}}_{\text{Int}}(f \overline{@} g \overline{@} \text{false})$$

which statically reduces to $\underline{\lambda} b.\underline{\text{if}}\ b\ \underline{\text{then}}\ 3\ \underline{\text{else}}\ 4$.

4 Limitations

Our full type-directed partial evaluator is not formally proven. Only its restriction to the simply typed λ-calculus has been proven correct, because it coincides with Berger and Schwichtenberg's algorithm [3]. (The two-level λ-calculus, though, provides a more convenient format for proving, *e.g.*, that static reduction does not go wrong and yields a completely dynamic term.)

This section addresses the practical limitations of type-directed partial evaluation.

4.1 Static errors and non-termination may occur

As soon as we move beyond the simply-typed λ-calculus, nothing *a priori* guarantees that type-directed partial evaluation yields no static errors or even terminates. (As usual in partial evaluation, one cannot solve the halting problem.) For example, given the looping thunk `loop`, the expression

```
(residualize (lambda (dummy) ((loop) (/ 1 0)))
             '(Dummy -> Whatever))
```

may either diverge or yield a "division by zero" error, depending on the Scheme processor at hand, since the order in which sub-expressions are evaluated, in an application, is undetermined [10].

4.2 Residual programs must have a type

We must know the type of every residual program, since it is this type that directs residualization. (The static parts of a source program, however, need not be typed.)

Residual programs can be polymorphically typed at base type (names of base types do not matter), but they must be monomorphically typed at higher types. Overcoming this limitation would require one to pass type tags to polymorphic functions, and to enumerate possible occurrences of type tags at the application site of polymorphic functions (an F_2 analogue of control-flow analysis / closure analysis for higher-order programs).

Examples include definitional interpreters for programming languages with recursive definitions that depend on the type of the source program. Unless one can enumerate all possible instances of such recursive definitions (and thus abstract all the corresponding fixpoint operators in the definitional interpreter), these interpreters cannot be residualized.

Inductive types are out of reach as well because eta-expanding a term of type t that includes an inductive type t' does not terminate if t' occurs in negative position within t. For example, we can consider lists.

$$\begin{array}{rcl} \downarrow^{\text{List}(t)}\ w & = & \overline{\text{case}}\ w\ \overline{\text{of}} \\ & & \quad \overline{\text{nil}} \Rightarrow \underline{\text{nil}} \\ & & \quad [\!]\ \overline{\text{cons}}(v,\ w) \Rightarrow \underline{\text{cons}}(\downarrow^{t}\ v,\ \downarrow^{\text{List}(t)}\ w) \\ \uparrow^{t'}_{\text{List}(t)}\ w & = & \overline{\text{shift}}\ \kappa : \text{List}(t) \rightarrow t' \\ & & \overline{\text{in}}\ \underline{\text{case}}\ w\ \underline{\text{of}} \\ & & \quad\quad \underline{\text{nil}} \Rightarrow \overline{\text{reset}}_{t'}\ (\kappa \overline{@} \overline{\text{nil}}) \\ & & \quad\ \ [\!]\ \underline{\text{cons}}(x,\ y) \Rightarrow \\ & & \quad\quad\quad \overline{\text{reset}}_{t'}\ (\kappa \overline{@} \overline{\text{cons}}(\uparrow^{t'}_{t}\ x,\ \uparrow^{t'}_{\text{List}(t)}\ y)) \end{array}$$

where x and y are fresh. Reflecting upon a list-typed expression diverges.

The problem here is closely related to coding fixed-point operators in a call-by-value language. A similar solution can be devised and gives rise to a notion of lazy insertion of coercions.

4.3 Single-threading and computation duplication must be addressed

Type-directed partial evaluation does not escape the problem of computation duplication: a program such as

```
(lambda (f g x) ((lambda (y) (f y y)) (g x)))
```

is residualized as

```
(lambda (f g x) (f (g x) (g x)))
```

Fortunately, the Similix solution applies: a residual let expression should be inserted [8]. The residual term above then reads:

```
(lambda (f g x) (let ([y (g x)]) (f y y)))
```

We have implemented type-directed partial evaluation in such a way. This makes it possible to specialize a *direct-style* version of the Tiny interpreter in Section 2.3. The corresponding residual programs (see Figure 5) are in direct style as well. Essentially they use let expressions "let $v = f@x$ in e" instead of CPS "$f@x@\lambda v.e$".

4.4 Side effects

Unless side-effecting procedures can be performed statically, they need to be factored out and, in the absence of let insertion, be made continuation-passing.

At first, this can be seen as a shortcoming, until one considers the contemporary treatment of side effects in partial evaluators. Since Similix [8], all I/O-like side effects are residualized, which on the one hand is safe but on the other hand prevents, *e.g.*, the non-trivial specialization of an interpreter which finds its source program in a file. Ditto for specializing an interpreter that uses I/O to issue compile-time messages — they all are delayed until run time. Similar heuristics can be devised for other kinds of computational effects.

In summary, the treatment of side effects in partial evaluators is not clear cut. Type-directed partial evaluation at least offers a simple testbed for experiments.

4.5 Primitive procedures must be either static or dynamic

The following problem appears as soon as we move beyond the pure λ-calculus.

During residualization, a primitive procedure cannot be used both statically and dynamically. Thus for purposes of residualization, in a source expression such as

```
((lambda (x) (lambda (y) (+ (+ x 10) y))) 100)
```

the two instances of + must be segregated. The outer occurrence must be declared in the initial run-time environment:

```
(lambda (add)
  ((lambda (x) (lambda (y) (add (+ x 10) y))) 100))
```

This limitation may remind one of the need for binding-time separation in some partial evaluators [36, 41].

A simple solution, however, exists, that prevents segregation. Rather than binding a factorized primitive operator such as + to the offline procedure

```
(lambda (<fresh-name>)
  (lambda (a1 a2)
    `(,<fresh-name> ,a1 ,a2)))
```

one could bind it to an online procedure that probes its arguments for static-reduction opportunities.

```
(lambda (<fresh-name>)
  (lambda (a1 a2)
    (if (number? a1)
        (if (number? a2)
            (+ a1 a2)
            (if (zero? a1)
                a2
                `(,<fresh-name> ,a1 ,a2)))
        (if (and (number? a2) (zero? a2))
            a1
            `(,<fresh-name> ,a1 ,a2)))))
```

4.6 Type-directed partial evaluation is monovariant

Partial evaluation derives much power from polyvariance (the generation of mutually recursive specialized versions of source program points). Polyvariance makes it possible, *e.g.*, to derive linear string matchers out of a quadratic one and to compile pattern matching and regular expressions efficiently [15, 36]. We are currently working on making type-directed partial evaluation polyvariant.

4.7 Residual programs are hard to decipher

A pretty-printer proves very useful to read residual programs. We are currently experimenting with the ability to attach residual-name stubs to type declarations, as in Elf. This mechanism would liberate us from renaming by hand, as in the residual program of Figure 5.

5 Related Work

5.1 λ-calculus normalization and Gödelization

Normalization is traditionally understood as rewriting until a normal form is reached. In that context, (one-level) type-directed eta-expansion is a necessary step towards long $\beta\eta$-normal forms [31]. A recent trend, embodied by partial evaluation, amounts to staging normalization in two steps: a translation into an annotated language, followed by a symbolic evaluation. This technique of normalization by translation appears to be spreading [39]. Follow-up work on Berger and Schwichtenberg's algorithm includes Altenkirch, Hofmann, and Streicher's categorical reconstruction of this algorithm [1].[6] This reconstruction formalizes the environment of fresh identifiers generated by the reification of λ-abstractions as a categorical fibration. Berger and Schwichtenberg also dedicate a significant part of their paper to formalizing the generation of fresh identifiers (they represent abstract-syntax trees as base types).

In the presence of disjoint sums, the existence of a normalization algorithm in the simply typed lambda-calculus is not known. Therefore our naïve extension in Section 3 needs to be studied more closely. The call-by-value nature of our implementation, for example, makes it possible to distinguish terms that are undistinguishable under call-by-name.

```
> (residualize
    (lambda (f) (lambda (x) 42))
    '((a -> (b + c)) -> a -> Int))
(lambda (x0) (lambda (x1) 42))
> (residualize
    (lambda (f) (lambda (x) ((lambda (y) 42) (f x))))
    '((a -> (b + c)) -> a -> Int))
(lambda (x0) (lambda (x1) (case-record (x0 x1)
                            [(Left x2) 42]
                            [(Right x3) 42])))
>
```

In his PhD thesis [27], Goldberg investigates Gödelization, *i.e.*, the encoding of a value from one language into another. He identifies Berger and Schwichtenberg's algorithm as one instance of Gödelization, and presents a Gödelizer for proper combinators in the untyped λ-calculus.

An implementation of Berger and Schwichtenberg's algorithm in Standard ML can be found in Filinski's PhD

[6] In that work, reify is "quote" and reflect is "unquote"

thesis [23]. This implementation handles most of the examples displayed in the present paper, in ML. It is ingenious because as expressed in Figures 1 and 2, type-directed partial evaluation requires dependent types. It can be easily translated into Haskell (excluding disjoint sums, of course, for lack of computational effects).

5.2 Binding-time analysis

All of Nielson and Nielson's binding-time analyses dynamize functions in dynamic contexts because of the difficulties of handling contravariance in program analysis [44]. So do all other binding-time analyses [36], with the exception of Consel's [12] and Heintze's [40]. These analyses are polyvariant and thus they spawn another variant instead of dynamizing.

In practice, type-directed partial evaluation needs a simple form of binding-time analysis: a type inferencer where all base types are duplicated into a static version and a dynamic version. Whenever a primitive operation is applied to an expression which is not entirely static, it is factored out into the initial run-time environment. The other occurrences of this primitive operation do not need to be factored out, however (thus enabling a small form of polyvariance).

To use this binding-time analysis and more generally type-directed partial evaluation, the simplest is to define source programs as closed λ-terms, by abstracting all free occurrences of variables. (To enable the small form of polyvariance mentioned in the last paragraph, each occurrence of primitive operator can be abstracted separately.) One can then curry this program with the static variables first, and then residualize the application of this curried program to the static values, with respect to the type of the result. This simple use matches the statement of Kleene's S^m_n-theorem.

5.3 Partial evaluation

Type-directed partial evaluation radically departs from all other partial evaluators (and optimizing compilers) because it has no interpretive dimension whatsoever: its source programs are compiled. If anything, it is closest to run-time code generation (the output syntax need not be Scheme's).

The last ten years have seen two flavors of partial evaluation emerge: online and offline. Offline partial evaluation is staged into two components: program analysis and program specialization. Online partial evaluation is more monolithic. Extensive work on both sides [6, 11, 15, 34, 36, 41, 48, 51] has led to the conclusions of both the usefulness of program analysis and the need for online partial evaluation in a program specializer (as illustrated in Section 4.5). Because it relies on one piece of static information — the type of the residual program — type-directed partial evaluation appears as an extreme form of offline partial evaluation.

In the spring of 1989, higher-order partial evaluation was blooming at DIKU [34]. In parallel with Bondorf (then visiting Dortmund), the author developed a version of Similix [8] that did not dynamize higher-order values in dynamic contexts. In this unreleased version, instead, the specializer kept track of the arity of static closures and eta-expanded them when they occurred in dynamic contexts. Type-directed partial evaluation stems from this unpublished work. The idea, however, did not take. Despite the analogy with call unfolding, which is central to partial evaluation but unsafe without let insertion (under call by value), "Similix [...] refuses to lift higher-order values into residual expressions: lifting higher-order values and data structures is in general unsafe since it may lead to residual code that exponentially duplicates data structure and closure allocations" [9, page 327]. All the later higher-order partial evaluators developed at DIKU have adopted the same conservative strategy — a choice that Henglein questions from a type-theoretical standpoint [29]. In practice, this decision *created* the need for source binding-time improvements in offline partial evaluation [36, Chapter 12]. In contrast, binding-time coercions "improve binding times without explicit eta-conversion", to paraphrase the title of Bondorf's LFP'92 paper [7] — a property which should prove crucial for multi-level binding-time analyses since it eliminates the need for (unfathomed) multi-level binding-time improvements [26].

Thus Mix-like partial evaluation [36] and type-directed partial evaluation fundamentally contrast when it comes to dynamic computations: Mix-like partial evaluators do not associate any structure to the binding time "dynamic", whereas we rely on the type structure of dynamic computations in an essential way. As a consequence, and modulo the abstraction of the initial run-time environment (which must be defined in a partial evaluator anyway), type-directed partial evaluation needs a restricted form of binding-time analysis (see Section 5.2) but it needs no specialization by symbolic interpretation. It is, however, monovariant.

We are currently integrating the residualization algorithm in Pell-Mell, our partial evaluator for ML [40]. This algorithm fulfills our need for binding-time coercions at higher types. Type-directed partial evaluation also formalizes and clarifies a number of earlier pragmatic decisions in the system. For example, our treatment of inductive data structures can be seen as lazy insertion of coercions (Section 4.2).

5.4 Self-application

Self-application is the best-known way to optimize partial evaluation [6, 11, 15, 25, 36, 37, 41, 48]: rather than running the partial evaluator on a source program, with all the symbolic interpretive overhead this entails, one could instead

1. generate a specializer dedicated to this program (a.k.a. "a generating extension"), and
2. run this dedicated specializer on the available input.

To be efficient, self-application requires a good binding-time analysis and a good binding-time division in the partial evaluator [36, Section 7.3].

Type-directed partial evaluation is based on type inference, needs no particular binding-time division, and as illustrated in Section 2.4, is self-applicable as well.

5.5 Partial evaluation of definitional interpreters

In the particular cases where the source program p is a definitional interpreter, or where the source program is the partial evaluator PE itself and the static input is a definitional interpreter or PE itself, the partial-evaluation equations

$$\left\{\begin{array}{rcl} \text{run PE}\,\langle \mathsf{p}, \langle \mathsf{s}, \_\rangle\rangle & = & \mathsf{p}_{\langle \mathsf{s}, \_\rangle} \\ \text{run p}\,\langle \mathsf{s}, \mathsf{d}\rangle & = & \text{run}\ \mathsf{p}_{\langle \mathsf{s}, \_\rangle}\,\langle \_, \mathsf{d}\rangle \end{array}\right.$$

are known as the "Futamura Projections" [15, 25, 36]. As illustrated in Section 2.3 and (to a lesser extent) in Section 2.4, type-directed partial evaluation enables their implementation in a strikingly simple way. (The third Futamura projection, however, is out of reach because of the polymorphic type of $\mathsf{PE}_{\langle \mathsf{PE}, \_\rangle}$.)

6 Conclusion

To produce a residual program, a partial evaluator needs to residualize static values in dynamic contexts. Considering higher-order values introduces a new challenge for residualization. Most partial evaluators dodge this challenge by disallowing static higher-order values to occur in dynamic contexts — *i.e.*, in practice, by dynamizing both, and more generally by restricting residualized values to be of base type [4, 6, 9, 36, 37]. Only recently, some light has been shed on the residualization of values at higher types, given information about these types [18, 19].

We have presented an algorithm that residualizes a closed typed static value in a dynamic context, by eta-expanding the value with two-level eta-redexes and then reducing all static redexes. For the simply typed λ-calculus, the algorithm coincides with Berger and Schwichtenberg's "inverse of the evaluation functional" [3]. It is also interesting in its own right in that it can be used as a partial evaluator for closed compiled programs, given their type. This partial evaluator, in several respects, outperforms all previous partial evaluators, *e.g.*, in simplicity and in efficiency. It also provides a simple and effective solution to (denotational) semantics-directed compilation in the λ-calculus.

Future work includes formalizing type-directed partial evaluation,[7] extending it to a richer type language (*e.g.*, polymorphic or linear), making it more user-friendly, programming more substantial examples, and obtaining polyvariance.

An implementation is available on the author's home page.

Acknowledgements

Various parts of this work have benefited from the kind attention and encouragement of Thorsten Altenkirch, David Basin, Luca Cardelli, Craig Chambers, Charles Consel, Jim des Rivières, Dirk Dussart, John Hatcliff, Martin Hofmann, Thomas Jensen, Peter Mosses, Flemming Nielson, Dino Oliva, Bob Paige, Frank Pfenning, Jon Riecke, Amr Sabry, Michael Schwartzbach, Helmut Schwichtenberg, Thomas Streicher, Tommy Thorn, and Glynn Winskel. Thanks are due to the referees for insightful comments, and to Guy L. Steele Jr., for sheperding this paper. Particular thanks to Rowan Davies, Mayer Goldberg, Julia Lawall, Nevin Heintze, Karoline Malmkjær, Jens Palsberg, and René Vestergaard for discussions and comments. Hearty thanks to Andrzej Filinski for substantial e-mail interaction and comments.

All of the examples were run with R. Kent Dybvig's Chez Scheme system and with Aubrey Jaffer's SCM system. Most were type-checked using Standard ML of New Jersey.

The author gratefully acknowledges the support of the DART project (Design, Analysis and Reasoning about Tools) of the Danish Research Councils during 1995.

References

[1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a *reduction free* reduction proof. In Peter Dybjer and Randy Pollack, editors, *Informal Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory*, Göteborg, Sweden, May 1995. Report 85, Programming Methodology Group, Chalmers University and the University of Göteborg.

[2] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.

[3] 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.

[4] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1993. DIKU report 93/22.

[5] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[6] Anders Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. DIKU Report 90-17.

[7] Anders Bondorf. Improving binding times without explicit cps-conversion. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 1–10, San Francisco, California, June 1992. ACM Press.

[8] 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.

[9] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

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

[11] Charles Consel. *Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs*. PhD thesis, Université Pierre et Marie Curie (Paris VI), Paris, France, June 1989.

[12] Charles Consel. Polyvariant binding-time analysis for applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.

[13] Charles Consel and Olivier Danvy. From interpreting to compiling binding times. In Neil D. Jones, editor, *Proceedings of the Third European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 88–105, Copenhagen, Denmark, May 1990.

[7] Davies and Pfenning's logical formalization of binding-time analysis and offline partial evaluation opens a promising avenue [20]

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. 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.

[21] 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.

[22] Andrzej Filinski. Representing monads. In Boehm [5], pages 446–457.

[23] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995.

[24] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.

[25] Yoshihito Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2, 5*, pages 45–50, 1971.

[26] Robert Glück and Jesper Jørgensen. Efficient multilevel generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Seventh International Symposium on Programming Language Implementation and Logic Programming*, number 982 in Lecture Notes in Computer Science, pages 259–278, Utrecht, The Netherlands, September 1995.

[27] Mayer Goldberg. *Recursive Application Survival in the λ-Calculus*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, 1996. Forthcoming.

[28] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [5], pages 458–471.

[29] 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.

[30] Gérard Huet. Résolution d'équations dans les langages d'ordre 1, 2, ..., ω. Thèse d'État, Université de Paris VII, Paris, France, 1976.

[31] C. Barry Jay and Neil Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(3):135–154, 1995.

[32] Neil D. Jones, editor. *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, Aarhus, Denmark, 1980.

[33] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In *Partial Evaluation and Mixed Computation*, pages 1–14. North-Holland, 1988.

[34] Neil D. Jones. Mix ten years after. In William L. Scherlis, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 24–38, La Jolla, California, June 1995.

[35] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Æ. Mogensen. A self-applicable partial evaluator for the lambda calculus. In K. C. Tai and Alexander L. Wolf, editors, *Proceedings of the 1990 IEEE International Conference on Computer Languages*, pages 49–58, New Orleans, Louisiana, March 1990.

[36] 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.

[37] John Launchbury. *Projection Factorisations in Partial Evaluation*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1991.

[38] 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.

[39] Ralph Loader. Normalisation by translation. Technical report, Computing Laboratory, Oxford University, April 1995.

[40] 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.

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

[42] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[43] Peter D. Mosses. SIS — semantics implementation system, reference manual and user guide. Technical Report MD-30, DAIMI, Computer Science Department, Aarhus University, Aarhus, Denmark, 1979.

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

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

[46] 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.

[47] John C. Reynolds. The essence of Algol. In van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, 1982. North-Holland.

[48] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, California, February 1993. Technical report CSL-TR-93-563.

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

[50] Philip Wadler. The essence of functional programming (tutorial). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[51] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 165–191, Cambridge, Massachusetts, August 1991.

A Nielson and Nielson's Two-Level λ-Calculus

In its most concise form, the two-level simply typed λ-calculus [44] duplicates all the constructs of the simply typed λ-calculus (λ, application (hereby noted @), pair, fst, snd) into static constructs ($\overline{\lambda}$, $\overline{@}$, $\overline{\text{pair}}$, $\overline{\text{fst}}$, $\overline{\text{snd}}$) and dynamic constructs ($\underline{\lambda}$, $\underline{@}$, $\underline{\text{pair}}$, $\underline{\text{fst}}$, $\underline{\text{snd}}$).

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

(define-record (Skip))
(define-record (Sequence command command))
(define-record (Assign identifier 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))

(define meaning-type
  `((Int * Int * (Int -> Ans) => Ans) *    ;;; add
    (Int * Int * (Int -> Ans) => Ans) *    ;;; sub
    (Int * Int * (Int -> Ans) => Ans) *    ;;; mul
    (Int * Int * (Int -> Ans) => Ans) *    ;;; eq
    (Int * Int * (Int -> Ans) => Ans) *    ;;; gt
    ((Int -> Ans) -> Ans) *                ;;; read
    (((Sto -> Ans) -> Sto -> Ans) -> Sto -> Ans) *
    (Int * (Sto -> Ans) * (Sto -> Ans) * Sto => Ans) *
    (Nat * Sto * (Int -> Ans) => Ans) *    ;;; lookup
    (Nat * Int * Sto * (Sto -> Ans) => Ans) =>
    (Sto -> Ans) *                         ;;; continuation
    Sto =>                                 ;;; store
    Ans))
```

Figure 9: Scheme interpreter for Tiny (abstract syntax and semantic algebras)

A simply typed λ-term is mapped into a two-level λ-term by a binding-time analysis. The intention is to formalize the following idea:

> Statically reducing a two-level λ-term, erasing the annotations of the residual term, and reducing this unannotated term should yield the same result (normal form) as reducing the original term.

The two-level λ-calculus thus provides an ideal medium for staged evaluation with more than one binding time. To this end, it makes use of three properties that are captured in its typing discipline:

- static reduction preserves well-typing;
- static reduction strongly normalizes;
- static reduction yields normal forms that are completely dynamic.

Static reduction can be implemented directly in a functional language: overlined constructs are treated as syntax constructs and dynamic constructs as syntax-building functions. It can also be implemented with quasiquote and unquote in Scheme (as done in Section 1) and thus can be seen as macro-expansion in a simply typed setting.

B Abstracting Control with Shift and Reset

This section provides some intuition about the effect of shift and reset.

Shift and reset were introduced to capture composition and identity over continuations [16, 17]. Reset delimits a context, and is identical to Felleisen's prompt; shift abstracts a delimited context, and is similar (though not in general

```
(define meaning
  (lambda (p)
    (lambda (add sub mul eq gt read fix true? lookup update)
      (lambda (k s)
        (letrec ([meaning-program
                  (lambda (p k s)
                    (case-record p
                      [(Program vs c) (meaning-declaration vs 0 (lambda (r) (meaning-command c r k s)))]))]
                 [meaning-declaration
                  (lambda (d offset k)
                    (if (null? d)
                        (k (lambda (i) (error 'lookup "undeclared identifier ~s" i)))
                        (meaning-declaration (cdr d)
                                             (add1 offset)
                                             (lambda (r) (k (lambda (i) (if (eq? (car d) i) offset (r i))))))))]
                 [meaning-command
                  (lambda (c r k s)
                    (case-record c
                      [(Skip) (k s)]
                      [(Sequence c1 c2) (meaning-command c1 r (lambda (s) (meaning-command c2 r k s)) s)]
                      [(Assign i e) (meaning-expression e r (lambda (v) (update (r i) v s k)) s)]
                      [(Conditional e c-then c-else)
                       (meaning-expression e r (lambda (v)
                                                 (true? v
                                                        (lambda (s) (meaning-command c-then r k s))
                                                        (lambda (s) (meaning-command c-else r k s))
                                                        s)) s)]
                      [(While e c)
                       ((fix (lambda (while)
                               (lambda (s)
                                 (meaning-expression e r (lambda (v)
                                                           (true? v
                                                                  (lambda (s) (meaning-command c r while s))
                                                                  k
                                                                  s)) s)))) s)]))]
                 [meaning-expression
                  (lambda (e r k s)
                    (case-record e
                      [(Literal l) (k l)]
                      [(Identifier i) (lookup (r i) s k)]
                      [(Primop op e1 e2)
                       (meaning-expression e1 r (lambda (v1)
                                                  (meaning-expression e2 r (lambda (v2)
                                                                             ((meaning-primop op) v1 v2 k)) s)) s)]
                      [(Read) (read k)]))]
                 [meaning-primop
                  (lambda (op)
                    (case op [(+) add] [(-) sub] [(*) mul] [(=) eq] [(>) gt]))])
          (meaning-program p k s)))))
```

Figure 10: Scheme interpreter for Tiny (valuation functions)

equivalent) to Felleisen's control [21]. Since contexts are delimited, their abstraction can be composed.

Let us consider some examples.

$1 + \text{reset}\ (2 \times \text{shift}\ k\ \text{in}\ 3 \times ((k\,4) + (k\,5)))$
$= 1 + \text{let}\ k = \lambda v.2 \times v\ \text{in}\ 3 \times ((k\,4) + (k\,5)) = 55$

$1 + \text{reset}\ (2 \times \text{shift}\ k\ \text{in}\ 3 \times (k\,4))$
$= 1 + \text{let}\ k = \lambda v.2 \times v\ \text{in}\ 3 \times (k\,4) = 25$

$1 + \text{reset}\ (2 \times \text{shift}\ k\ \text{in}\ k\,10)$
$= 1 + \text{let}\ k = \lambda v.2 \times v\ \text{in}\ k\,10 = 21$

$1 + \text{reset}\ (2 \times \text{shift}\ k\ \text{in}\ 10)$
$= 1 + \text{let}\ k = \lambda v.2 \times v\ \text{in}\ 10 = 11$

In the three terms, k is bound to an abstraction of the delimited context $[2 \times [\,]]$. This abstraction reads $\lambda v.2 \times v$. In the first term, it is used twice; in the second and in the third, once; and in the last, it is not used.

Shift and reset can be eliminated by CPS transformation [49]. A shift expression is CPS-transformed by abstracting the current (delimited) continuation into a procedure. When this procedure is applied, the abstracted continuation is composed with the new current continuation. Finally, a reset expression is CPS-transformed by supplying the identity procedure as a continuation [16, 17].

Filinski's direct implementation of shift and reset can be found in the literature, both in Standard ML of New Jersey [22] and in Scheme [38].