BRICS

Basic Research in Computer Science

BRICS RS-97-53 O. Danvy: Online Type-Directed Partial Evaluation

Online Type-Directed Partial Evaluation

Olivier Danvy

BRICS Report Series RS-97-53

ISSN 0909-0878 December 1997

**See back inner page for a list of recent BRICS Report Series publications.
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 the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/97/53/`

Online Type-Directed Partial Evaluation *

Olivier Danvy

BRICS †

Department of Computer Science

University of Aarhus ‡

December 1997

Abstract

In this experimental work, we extend type-directed partial evaluation (a.k.a. "reduction-free normalization" and "normalization by evaluation") to make it online, by enriching it with primitive operations (δ-rules). Each call to a primitive operator is either unfolded or residualized, depending on the operands and either with a default policy or with a user-supplied filter. The user can also specify how to residualize an operation, by pattern-matching over the operands. Operators may be pure or have a computational effect.

We report a complete implementation of online type-directed partial evaluation in Scheme, extending our earlier offline implementation. Our partial evaluator is native in that it runs compiled code instead of using the usual meta-level technique of symbolic evaluation.

*Extended version of an article to appear in the proceedings of the Third Fuji International Symposium on Functional and Logic Programming (FLOPS'98), Kyoto, Japan, April 2-4, 1998.

†Basic Research in Computer Science,
Centre of the Danish National Research Foundation.
Home page: `http://www.brics.dk`

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

1 Introduction and Motivation

Type-directed partial evaluation [17] is a practical outgrowth of an intriguing normalization property lying at the interface between the object level and the meta-level of a simply typed two-level λ-calculus. Namely: Let us consider a λ-term e which is closed and lives in the meta-level. We can coerce it into a two-level λ-term by the obvious two-level η-expansion (noted with a type-indexed downarrow in Figure 1). Then reducing all the meta-level redices yields an object λ-term which corresponds to the long $\beta\eta$-normal form of e.

For example, if we let

$$\begin{array}{rcl} S & = & \overline{\lambda} f.\overline{\lambda} g.\overline{\lambda} x.(f \mathbin{\overline{@}} x) \mathbin{\overline{@}} (g \mathbin{\overline{@}} x) \\ K & = & \overline{\lambda} a.\overline{\lambda} b.a \end{array}$$

then reducing the meta-level redices of $\downarrow^{\alpha \to \alpha} (S \mathbin{\overline{@}} K \mathbin{\overline{@}} K)$ yields $\underline{\lambda} x.x$.

This property was first noticed by Berger and Schwichtenberg [5], who exploited it to normalize programs extracted from proofs. They implemented the two-level λ-calculus directly in Scheme by letting meta-level terms be Scheme values and object-level terms be S-expressions (i.e., lists). The corresponding two-level η-expander is displayed in Figure 2. This Scheme procedure is passed a representation of a type and a (closed) Scheme value of that type, constructs (first-order) S-expressions using Scheme's quasiquote and unquote [11], and returns a S-expression representing the long $\beta\eta$-normal form of the Scheme value. Later, Coquand machine-checked Berger and Schwichtenberg's algorithm [15] and Berger presented an alternative version by extracting it from a normalization proof [4]. In his PhD thesis [25], Goldberg investigates other encodings of a value from one language into another, which he calls "Gödelization."

Both the foundations and the applications of two-level η-expansion are being explored today. Altenkirch, Hofmann and Streicher, and Čubrić, Dybjer and Scott are conducting a mathematical investigation [1, 2, 16]. Danvy and his students, and Sheard and his students are conducting a more experimental investigation [17, 18, 21, 22, 35, 36]. The present work continues our practical investigation.

Overview: The rest of this article is organized as follows. In Section 2, we briefly review the state of the art of offline type-directed partial evaluation in Scheme. In Section 3, we describe a very simple online extension of

$$
\begin{array}{rcl}
t & ::= & \alpha \mid t_1 \rightarrow t_2 \mid t_1 \times t_2 \\
e & ::= & x \mid \overline{\lambda} x.e \mid e_0 \mathbin{\overline{@}} e_1 \mid \overline{\mathrm{pair}}(e_1, e_2) \mid \overline{\pi_1}(e) \mid \overline{\pi_2}(e) \\
 & & \mid \underline{\lambda} x.e \mid e_0 \mathbin{\underline{@}} e_1 \mid \underline{\mathrm{pair}}(e_1, e_2) \mid \underline{\pi_1}(e) \mid \underline{\pi_2}(e)
\end{array}
$$

$$
\begin{array}{rrcll}
 & \downarrow^{\alpha} e & = & e & \\
 & \downarrow^{t_1 \rightarrow t_2} e & = & \underline{\lambda} x_1.\downarrow^{t_2} (e \mathbin{\overline{@}} (\uparrow_{t_1} x_1)) & \text{where } x_1 \text{ is fresh.} \\
 & \downarrow^{t_1 \times t_2} e & = & \underline{\mathrm{pair}}(\downarrow^{t_1} \overline{\pi_1}(e), \downarrow^{t_2} \overline{\pi_2}(e)) & \\
\text{where} & \uparrow_{\alpha} e & = & e & \\
 & \uparrow_{t_1 \rightarrow t_2} e & = & \overline{\lambda} x_1.\uparrow_{t_2} (e \mathbin{\underline{@}} (\downarrow^{t_1} x_1)) & \text{where } x_1 \text{ is fresh.} \\
 & \uparrow_{t_1 \times t_2} e & = & \overline{\mathrm{pair}}(\uparrow_{t_1} \underline{\pi_1}(e), \uparrow_{t_2} \underline{\pi_2}(e)) &
\end{array}
$$

Applications are noted with an infix "@". Meta-level constructs are overlined, and object-level constructs are underlined.

N.B. $\downarrow$ (resp. $\uparrow$) is indexed by types occurring positively (resp. negatively) in the source type.

Figure 1: Simply typed two-level η-expansion

type-directed partial evaluation with primitive operators, and we refine it as non-intrusively as we can to make it practically useful. Section 4 reviews related work and Section 5 concludes.

Prerequisites: We assume some rudimentary knowledge about partial evaluation [14, 17, 28] and a reasonable familiarity with (typed) functional programming in general and Scheme in particular [11]. We use Scheme because of its syntactic flexibility (little need for parsing and unparsing due to S-expressions, pretty-printing facilities, syntactic extensions), its semantic versatility (dynamic typing, overloading), and ultimately (and subjectively) its elegance.

2 Offline Type-Directed Partial Evaluation

The challenge of implementing type-directed partial evaluation lies in the fact that it is *compiled code* that is being run. The usual partial-evaluation techniques [14, 28] are of very little help here, since they fundamentally rely

```
(define tdpe
  (lambda (v t)
    (letrec ([reify
              (lambda (t v)
                (cond
                  [(base-type? t)
                   v]
                  [(function-type? t)
                   (let ([t1 (function-type->domain t)]
                         [t2 (function-type->range t)])
                     (let ([x1 (gensym! (type->name-stub t1))])
                       `(lambda (,x1)
                          ,(reify t2 (v (reflect t1 x1))))))]
                  [(product-type? t)
                   (let ([t1 (product-type->first t)]
                         [t2 (product-type->second t)])
                     `(cons ,(reify t1 (car v))
                            ,(reify t2 (cdr v))))]))]
             [reflect
              (lambda (t e)
                (cond
                  [(base-type? t)
                   e]
                  [(function-type? t)
                   (let ([t1 (function-type->domain t)]
                         [t2 (function-type->range t)])
                     (lambda (v1)
                       (reflect t2 `(,e ,(reify t1 v1)))))]
                  [(product-type? t)
                   (let ([t1 (product-type->first t)]
                         [t2 (product-type->second t)])
                     (cons (reflect t1 `(car ,e))
                           (reflect t2 `(cdr ,e))))]))])
      (begin
        (reset-gensym!)
        (reify t v)))))
```

Figure 2: Two-level η-expansion in Scheme
(a.k.a. type-directed partial evaluation)

$$
\begin{array}{rcl}
\langle\text{type}\rangle & ::= & \texttt{Bool} \\
 & | & \langle\text{base-type}\rangle \\
 & | & \texttt{(}\langle\text{compound-type}\rangle\texttt{)} \\[1ex]
\langle\text{compound-type}\rangle & ::= & \langle\text{type}\rangle\ \texttt{*}\ \langle\text{type}\rangle \\
 & | & \langle\text{type}\rangle\ \texttt{+}\ \langle\text{type}\rangle \\
 & | & \langle\text{single-domain}\rangle\ \{\texttt{->}\ |\ \texttt{-!>}\}\ \langle\text{single-range}\rangle \\
 & | & \langle\text{multiple-domain}\rangle\ \{\texttt{=>}\ |\ \texttt{=!>}\}\ \langle\text{single-range}\rangle \\[1ex]
\langle\text{single-domain}\rangle & ::= & \langle\text{type}\rangle \\[1ex]
\langle\text{multiple-domain}\rangle & ::= & \texttt{(}\langle\text{type}\rangle^{*}\texttt{)} \\[1ex]
\langle\text{single-range}\rangle & ::= & \langle\text{type}\rangle
\end{array}
$$

N.B. The arrows `->`, `-!>`, `=>`, and `=!>` associate to the right.

Figure 3: Abstract syntax of types

on having access to *source code* for analysis and transformation. That is not the case here, since specialization is performed by running the program. Therefore there is only one partial-evaluation policy, which a fortiori is fixed prior to program specialization. Type-directed partial evaluation is thus an extreme form of *offline* partial evaluation [14, 28].

As documented earlier [17, 18], we have already extended type-directed partial evaluation to make it reasonably applicable to Scheme. The extension handles literals, uncurried functions, functions with computational effects, booleans and sums, multiple results, and a simple record facility. The user also has a say in the generation of residual names, to make residual programs readable.

For simplicity, in the rest of this article, we only consider values of base type, uncurried functions, functions with computational effects, products, booleans, and sums. Let us illustrate each of these points in turn. First of all, the syntax of types is displayed in Figure 3. A base type is a type variable (noted α in Figure 1). A compound type is a product, a sum, or a function. An uncurried function has a multiple domain.[1] Function types can be annotated with a computational effect (noted "`!`" in Figure 3).

[1]The types "`((a) => a)`" and "`(a -> a)`" are synonymous.

Base types and compound types are declared as follows:

```
(define-base-type <identifier> {<string>})
(define-compound-type <identifier> <type> {<string>})
```

The string parameter is optional. It serves as a "name stub" for generating residual names, alleviating the need to rename residual programs by hand to make them readable. This naming feature is also available in the constraint logic-programming language Elf [33].

Because type-directed partial evaluation only handles type schemes, base types do not matter. However, because we are human, their name does to us. Therefore, in the rest of this section, we will assume that we have already defined the base types `a`, `b`, and `c`, as in the following interactive Scheme session:

```
> (define-base-type a)
> (define-base-type b)
> (define-base-type c)
>
```

By default, the name stub for residual variables of type `a` will be `"a"`, etc.

Road map: The rest of this section is organized as follows. We first illustrate pure type-directed partial evaluation with examples from the pure λ-calculus: the combinator example of Section 1 and Church numbers (Section 2.1). We then illustrate applied type-directed partial evaluation with examples involving uncurried functions, functions with computational effects, and literals (Section 2.2). Literals beg for a context-sensitive partial-evaluation policy, which we achieve by making type-directed partial evaluation online (Section 3).

2.1 Pure type-directed partial evaluation

In this section, we illustrate pure type-directed partial evaluation as specified in Figure 1 and implemented in Figure 2. Overlined λ-abstractions and applications are represented as Scheme's λ-abstractions and applications. Underlined λ-abstractions and applications are represented as Scheme lists.

2.1.1 The Hilbert combinators

Let us reproduce the combinator example of Section 1.

```
> (define S
    (lambda (f)
      (lambda (g)
        (lambda (x)
          ((f x) (g x))))))
> (define K
    (lambda (a)
      (lambda (b)
        a)))
> (define I ((S K) K))
> (I 42)
42
> (tdpe I '(a -> a))
(lambda (a0) a0)
>
```

We define the Hilbert combinators S and K as the Scheme procedures `S` and `K`, and then the identity combinator I as usual. `I` denotes a Scheme procedure that we can apply, e.g., to 42. We can also residualize it into the text of its normal form by type-directed partial evaluation, using the Scheme procedure `tdpe` of Figure 2. `tdpe` is passed a Scheme value and a representation of its type (as a constant Scheme list), and yields a representation of the normal form of this value (as a Scheme list).

In summary, and as illustrated here, type-directed partial evaluation constructs the text of the long $\beta\eta$-normal form of a closed higher-order value obtained, e.g., by combining other higher-order values. This construction is achieved by two-level η-expansion, as specified in Figure 1 and as directly implemented in Figure 2.

2.1.2 Church numbers

We define the Church number `c0` (representing zero) and the Church successor function `cs`, and then the Church number representing three. This number is a Scheme procedure that we can apply, e.g., to the Scheme successor function and to the Scheme representation of zero, to obtain the Scheme representation of three. We can also residualize it into the text of its normal form by type-directed partial evaluation.

```
> (define c0
    (lambda (s)
      (lambda (z)
        z)))
> (define cs
    (lambda (n)
      (lambda (s)
        (lambda (z)
          (s ((n s) z))))))
> (((cs (cs (cs c0))) 1+) 0)
3
> (tdpe (cs (cs (cs c0))) '((a -> a) -> a -> a))
(lambda (x0)
  (lambda (a1)
    (x0 (x0 (x0 a1)))))
>
```

To improve the readability of this residual code, we can declare the types of c0 and of cz:

```
> (define-compound-type z a)
> (define-compound-type s (a -> a))
> (tdpe (cs (cs (cs c0))) '(s -> z -> a))
(lambda (s0)
  (lambda (z1)
    (s0 (s0 (s0 z1)))))
>
```

2.2 Applied type-directed partial evaluation

In this section, we illustrate uncurried functions, functions with computational effects, and literals.

2.2.1 Uncurried functions

Type-directed partial evaluation handles Scheme's uncurried functions. For example, here is the uncurried S combinator:

```
> (define uS (lambda (f g x) (f x (g x))))
> (tdpe uS '((((a b) => c) (a -> b) a) => c))
(lambda (x0 x1 a2) (x0 a2 (x1 a2)))
>
```

2.2.2 Functions with computational effects

Functions with computational effects, or whose calls we do not want to duplicate, are treated by inserting a residual let expression [18]:

```
> (tdpe uS '((((a b) =!> c) (a -!> b) a) =!> c))
(lambda (x0 x1 a2)
  (let ([b3 (x1 a2)])
    (x0 a2 b3)))
>
```

In this example, we have specified that both the first and the second parameters of `uS` have an effect. In the residual code, the application of the first one does not need to be named since it is a tail-call. The application of the second one, however, is named.

Let insertion is very useful in practice. For example, it makes it possible to specialize definitional interpreters expressed in direct style [22]. Usually, definitional interpreters need to be written in continuation-passing style to specialize well [13, 14, 28].

2.2.3 Literals

Handling literals requires some initiative from the user, in the sense that because we are running compiled code, a distinction needs to be made *at the source level* between the static occurrences of operations over these literals and the dynamic ones. For example suppose we want to residualize the application of the function (call it `foo`)

```
(lambda (x) (lambda (y) (lambda (f) (f (+ x 1) (+ y 1)))))
```

to, e.g., the literal 10. Scheme will raise an error: the right-most occurrence of `+` expects two numbers, not the residual identifier denoted by `y`.

The source program therefore needs to be *factorized*, in the sense that all primitive functions over dynamic literals need to be abstracted out, e.g., as follows.

```
> (define abstracted-foo
    (lambda (x)
      (lambda (y add)
        (lambda (f)
          (f (add x 1) (add y 1))))))
```

```
> (define-base-type Int "n")
> (define-compound-type Add ((Int Int) => Int) "add")
> (define-compound-type F ((Int Int) => Int) "f")
> (tdpe (abstracted-foo 10) '((Int Add) => F -> Int))
(lambda (n0 add1)
  (lambda (f2)
    (f2 (add1 10 1) (add1 n0 1))))
>
```

But the residual program is unsatisfactory in that the addition of 10 to 1 did not happen at partial-evaluation time. More discernment is needed in the factorization: only the dynamic occurrence of addition should be factorized.

```
> (define discerning-abstracted-foo
    (lambda (x)
      (lambda (y add)
        (lambda (f)
          (f (+ x 1) (add y 1))))))
> (tdpe (discerning-abstracted-foo 10)
        '((Int Add) => F -> Int))
(lambda (n0 add1)
  (lambda (f2)
    (f2 11 (add1 n0 1))))
>
```

The addition of 10 to 1 happened at partial-evaluation time — but at the cost of much effort. Online type-directed partial evaluation precisely stems from the desire to get rid of this kind of gymnastics.

3 Online Type-Directed Partial Evaluation

We thus introduce a facility to declare primitive functions — functions that will be used atomically during type-directed partial evaluation. In the rest of this section, we refer to them as *primitive operators*, or again just as *operators*.

```
(define-primitive <identifier> <type> <static-version>)
```

The policy of an operator is blissfully simple: if all operands are static (i.e., none is a piece of residual syntax), the static version is invoked; otherwise, a residual call [to this operator] is constructed, based on its type. The static version is also invoked at run time (where no operand is ever a piece of residual syntax).

Primitive operators are thus fundamentally overloaded, and overloaded in a way that cannot be resolved at compile time.

Road map: The rest of this section is organized as follows. We first illustrate pure primitive operators with the example of Section 2.2.3 (Section 3.1). We then describe impure primitive operators, i.e., primitive operators whose type is annotated with an effect (Section 3.2). Both kinds of operators are definable over base types. But what about compound types (products and functions)? They require more flexibility than the default partial-evaluation policy. We therefore parameterize primitive operators with user-defined filters (Sections 3.3 and 3.4). We then illustrate filters with a standard example in partial evaluation: the exponentiation function (Section 3.5). Turning to the residualization of pure primitive operators, we also make it user-definable (Section 3.6). To this end, we introduce a domain-specific language over residual terms (Section 3.7). We illustrate it (Section 3.8) and, finally, we revisit impure primitive operators and make their residualization user-definable as well (Section 3.9).

3.1 Pure primitive operators

Getting back to the example of Section 2.2.3, defining addition as a primitive operation relieves us from having to selectively abstract the dynamic occurrences of free variables in `foo`. As a side benefit, `foo` naturally becomes discerning:

```
> (define-primitive add ((Int Int) => Int) +)
> (define foo
    (lambda (x)
      (lambda (y)
        (lambda (f)
          (f (add x 1) (add y 1))))))
> (tdpe (foo 10) '(Int -> F -> Int))
(lambda (n0)
  (lambda (f1)
    (f1 11 (add n0 1))))
>
```

We have declared `add` to be a primitive operator. During partial evaluation, this operator is invoked twice: once on completely static operands (10 and

1) and once on incompletely static operands (a residual variable and 1). In the former case, a static addition takes place, yielding 11. In the latter case, the call to `add` is residualized.

As this first example illustrates, primitive operations fit in type-directed partial evaluation smoothly. In particular, by construction, they are context-sensitive and thus their binding times are polyvariant.

Overall, source programs are still abstracted with all the primitive operators. Defining primitive operators, however, is considerably less intrusive than having to lambda-abstract their dynamic instances.

3.2 Impure primitive operators

What about impure, i.e., effectful primitive operators? We declare them as primitive operators and we annotate their type with an effect. An impure primitive operator is then given the same treatment as any other effectful function, i.e., the result of each of its calls is named with a let expression. Impure primitive operators thus fit in type-directed partial evaluation just as smoothly as pure ones do.

We can illustrate them by annotating the type of `add`, and going through the same steps as above:

```
> (define-primitive add! ((Int Int) =!> Int) +)
> (define foo!
    (lambda (x)
      (lambda (y)
        (lambda (f)
          (f (add! x 1) (add! y 1))))))
> (tdpe (foo! 10) '(Int -> F -!> Int))
(lambda (n0)
  (lambda (f1)
    (let* ([n2 (add! 10 1)]
           [n3 (add! n0 1)])
      (f1 n2 n3))))
>
```

Both calls to `add!` have been unconditionally residualized and sequentialized.

This raises a new problem: if impure primitive operators are unconditionally residualized, how can we ever *run* a program using impure primitive operators? The problem hinges on the fact that type-directed partial evaluation happens *at run time*.

We solve this problem by adding one global switch in our implementation. This switch controls the partial-evaluation mode. If it is on, primitive operators work as described in this section. If it is off, only their static version is accessible. In effect, the switch only makes a difference for impure primitive operators.

3.3 Compound types

So far, we only have considered primitive operators from base type(s) to base type, taking advantage of the fact that at base type, it is trivial to test the "staticness" of any operand: just check whether it is a piece of residual syntax. Compound-type values such as higher-order functions, however, are represented as such — i.e., as higher-order functions. They are not as easily recognizable as base-type values.

We could grope for a mechanism. Curried operators over base-type domains, for example, are simple to handle: just wait until they are completely applied, and then test whether all their operands are static. A more general solution, however, is necessary, and we describe it in the following section.

3.4 Controlling unfolding

We thus introduce a facility to parameterize the partial-evaluation policy of an operator, *filters*, which are user-supplied predicates over the operands. (Filters were inspired by Schism [12], as addressed in Section 4.)

```
(define-primitive-with-filter <identifier> <type>
  <filter>
  <static-version>)
```

A filter guards an operator and determines its partial-evaluation policy: it is applied to all the operands, as directed by the type, and returns a boolean value indicating whether to invoke the static version or to residualize the call. It typically use the predicates `static?` and `dynamic?` over base-type values. If the type of an operator is annotated with an effect, its filter is ignored; this is consistent with Section 3.2.

3.5 An example: the exponentiation function

Figure 4 displays a complete program for computing the power function, using the Russian-peasant algorithm. This program makes use of everything

```
(define-base-type Int "n")
(define-compound-type Loop (Int -> Int) "loop")

(define-primitive is-zero? (Int -> Bool) zero?)
(define-primitive is-odd? (Int -> Bool) odd?)
(define-primitive dec (Int -> Int) 1-)
(define-primitive mul ((Int Int) => Int) *)
(define-primitive sqr (Int -> Int) (lambda (x) (* x x)))
(define-primitive div ((Int Int) => Int) /)

(define-primitive-with-filter fix
  ((Loop -> Loop) -> Loop)
  (lambda (f)
    (lambda (n)
      (static? n)))
  (lambda (f)   ;;; Curry's applicative-order fixed-point operator
    ((lambda (x) (f (lambda (a) ((x x) a))))
     (lambda (x) (f (lambda (a) ((x x) a)))))))

(define power
  (lambda (x n)
    ((fix (lambda (loop)
            (lambda (n)
              (cond
                [(is-zero? n)
                 1]
                [(is-odd? n)
                 (mul x (loop (dec n)))]
                [else
                 (sqr (loop (div n 2)))])))) n)))
```

Figure 4: The exponentiation function

we have described so far: definitions of base type (`Int`) and of compound type (`Loop`), including the specification of name stubs for residual variables; definition of primitive operators with the default partial-evaluation policy; and definition of a primitive operator with a user-supplied filter. The filter here is very simple: the fixed-point operator should proceed whenever the exponent is static. The main function is `power`. In effect it is closed, since all its free variables are explicitly declared as primitive operators.

Running power:

```
> (power 10 10)
10000000000
>
```

Applying power to two static arguments yields a static result.

Specializing power with respect to a given exponent:

```
> (tdpe (lambda (x) (power x 10)) '(Int -> Int))
(lambda (n0)
  (sqr (mul n0 (sqr (sqr (mul n0 1))))))
> ((lambda (n0)
     (sqr (mul n0 (sqr (sqr (mul n0 1))))))
   10)
10000000000
>
```

Applying power to a dynamic base and the static exponent 10 yields a residual program, which is the specialized version of power with respect to the exponent 10. As usual in partial evaluation, running the specialized program on the remaining input (10) yields the same result as running the source program on the complete input (10 and 10).

Analysis: because the exponent is static, the filter yields true and all the recursive calls in power are unfolded.

Specializing power with respect to a given base:

```
> (tdpe (lambda (n) (power 10 n)) '(Int -> Int))
(lambda (n0)
  ((fix1 (lambda (loop1)
           (lambda (n2)
             (cond [(is-zero? n2)
                    1]
                   [(is-odd? n2)
                    (mul 10 (loop1 (dec n2)))]
                   [else
                    (sqr (loop1 (div n2 2)))]))))
   n0))
>
```

Applying power to the static base 10 and a dynamic exponent yields a residual program, which is the specialized version of power with respect to the base 10. Again, running the specialized program on the remaining input (10) would yield the same result as running the source program on the complete input (10 and 10).

Analysis: because the exponent is dynamic, the filter yields false and none of the recursive calls in `power` are unfolded. The static base 10, however, is inlined in the residual program.

Specializing power with respect to no argument:

```
> (tdpe power '((Int Int) => Int))
(lambda (n0 n1)
  ((fix1 (lambda (loop2)
           (lambda (n3)
             (cond
               [(is-zero? n3)
                1]
               [(is-odd? n3)
                (mul n0 (loop2 (dec n3)))]
               [else
                (sqr (loop2 (div n3 2)))]))))
   n1))
>
```

We can also residualize `power` with respect to `((Int Int) => Int)`, i.e., with respect to no static input. The result is its text.

3.6 Controlling residualization

We have made operators increasingly versatile, but one monolithic action remains: how they are residualized. Again, it makes sense to leave this policy up to the user. For example, in Figure 4, `mul` would be better defined to do something special if one of its operands is dynamic but the other is 0 or 1.

We therefore enrich the declaration of operators with an optional dynamic version specifying how to residualize their calls.

```
(define-primitive <identifier> <type>
  <static-version>
  {<dynamic-version>})

(define-primitive-with-filter <identifier> <type> <filter>
  <static-version>
  {<dynamic-version>})
```

We want, however, to preserve the user from having to deal directly with our representation of residual syntax. To this end, we introduce the following domain-specific language.

3.7 A domain-specific language over residual terms

For over ten years now, the programming language Scheme has been the theater of an intensive exploration of macros, viewed not as an inherently risky business, but as a reasonable way to extend the syntax of one's programming language. The line of research initiated in Kohlbecker's PhD thesis [29] and continued in Chez Scheme [24] is of particular interest to us. *Syntactic extensions*, as they are called, are hygienic macros[2] where abstract syntax is accessed by pattern matching.

In this section, we briefly present a domain-specific language over residual terms, designed jointly with Morten Rhiger [34]. This language follows Kohlbecker and Dybvig's lead, with a special form `case-syntax` (see Figure 5) providing access to residual terms by pattern matching. Dynamic versions of primitive operators are thus defined as a rewriting system, and therefore can be formalized using standard rewriting techniques.

- Some identifiers are listed first. They denote values that should be residualized and pattern-matched upon. Implicitly these identifiers are of base type; otherwise they are specified together with their type.
- The identifiers that should be considered as constants during pattern matching are listed next.
- Each clause specifies:
  - a list of patterns against which are matched the residualized values; pattern matching either fails or yields an environment binding the identifiers of the pattern to the corresponding residual term; the rest of the clause is evaluated in the extended environment;
  - an optional fender, which is a Scheme predicate evaluated in the extended environment and guarding the template;
  - a template specifying how to construct the corresponding residual term.

[2]Hygienic in the sense that no names are captured during macro-expansion [30].

```
(case-syntax (⟨head⟩ ...) (⟨constant-identifier⟩ ...)
    ⟨clause⟩
    ...
    [else ⟨template⟩])

where      ⟨head⟩ ::= ⟨identifier⟩ | [⟨identifier⟩ ⟨type⟩]
         ⟨clause⟩ ::= [(⟨pattern⟩ ...) ⟨template⟩]
                    | [(⟨pattern⟩ ...) ⟨fender⟩ ⟨template⟩]
        ⟨pattern⟩ ::= ⟨literal⟩
                    | ⟨constant-identifier⟩
                    | ⟨identifier⟩
                    | (⟨pattern⟩ ...)
       ⟨template⟩ ::= ⟨literal⟩
                    | ⟨constant-identifier⟩
                    | ⟨identifier⟩
                    | (⟨template⟩ ...)
                    | (with ([⟨identifier⟩ ⟨Scheme-expression⟩] ...)
                        ⟨template⟩)
         ⟨fender⟩ ::= ⟨Scheme-expression⟩
```

Figure 5: Abstract syntax of `case-syntax`

- A pattern can be a literal, an identifier that should be considered as a constant, an identifier that should be part of the resulting environment if pattern-matching succeeds, or a compound pattern for matching a residual application.

- Except for `with`-expressions, a template is constructed like a pattern, and is instantiated to construct a residual term. During this instantiation, static computations are enabled through `with`-expressions, which are essentially like `let`-expressions in Scheme.

3.8 A few examples

Thus equipped, we can define a binary addition that, when residualized, probes for 0 (the identity element of addition) and simplifies its result accordingly:

```
> (define-primitive add
    ((Int Int) => Int)
    +
    (lambda (x1 x2)
      (case-syntax (x1 x2) (add)
        [(0 x2)
         x2]
        [(x1 0)
         x1]
        [else
         (add x1 x2)])))

> (define bar
    (lambda (x z)
      (lambda (y)
        (add (add x z) (add y z)))))
> (tdpe (bar 0 0) '(Int -> Int))
(lambda (n0)
  (add n0 n0))
>
```

Primitive operators can also synergize:

- For example, Figure 6 displays a ternary operator adding its operands and deferring to `add` as we have just defined it if two of its operands are static.
- For another example, we have defined a multiplication operator `mul` probing its operands for other occurrences of `mul`, reassociating them and performing static multiplications whenever possible. With this operator, we were able to specialize the folding of a static function [multiplying its static argument with something dynamic] over a static tree of static numbers, and to perform *all* the multiplications of static numbers at specialization time.

Finally, primitive operators can upset the binding-time balance and make dynamic operations return static results. For example, multiplying a dynamic number by 0 (which is static) yields 0. In this example, the binding-time balance is upset because even though one of the multiplicands is dynamic, the result is static.

Probing for 0 and returning 0, and probing for 1 and returning the other dynamic argument are expressed as follows:

```
(define-primitive add3
  ((Int Int Int) => Int)
  +
  (lambda (x y z)
    (case-syntax (x y z) (add3 add)
      [(x y z)
       (and (static? x) (static? y) (zero? (+ x y)))
       z]
      [(x y z)
       (and (static? x) (static? y))
       (with ([v (+ x y)])
         (add v z))]
      [(x y z)
       (and (static? x) (static? z) (zero? (+ x z)))
       y]
      [(x y z)
       (and (static? x) (static? z))
       (with ([v (+ x z)])
         (add v y))]
      [(x y z)
       (and (static? y) (static? z) (zero? (+ y z)))
       x]
      [(x y z)
       (and (static? y) (static? z))
       (with ([v (+ y z)])
         (add x v))]
      [(0 y z) (add y z)]
      [(x 0 z) (add x z)]
      [(x y 0) (add x y)]
      [else    (add3 x y z)]))))
```

Figure 6: Ternary addition

```
> (define-primitive mul ((Int Int) => Int)
    *
    (lambda (x1 x2)
      (case-syntax (x1 x2) (mul)
        [(0 x2) 0]
        [(x1 0) 0]
        [(1 x2) x2]
        [(x1 1) x1]
        [else (mul x1 x2)])))
```

```
> (define baz
    (lambda (x y)
      (lambda (z f)
        (add x (mul y (f z))))))
> (tdpe (baz 10 0)
        '((Int (Int -> Int)) => Int))
(lambda (n0 x1)
  10)
>
```

In this session, we have made `mul` aware that 0 is absorbant. This awareness pays off when residualizing `(baz 10 0)` since it makes the multiplication yield 0 even though the second multiplicand is dynamic, which enables the addition to be performed statically.

But is it correct? The residual application `(f z)` has just been discarded. What if it had a computational effect?

Fortunately (and one more time), types save the day: if the function denoted by `f` was effectful, it would have been signaled in its type, and a residual let expression would have been inserted, as illustrated below:

```
> (tdpe (baz 10 0)
        '((Int (Int -!> Int)) =!> Int))
(lambda (n0 x1)
  (let ([n2 (x1 n0)])
    10))
>
```

In this session, the application of the effectful function is residualized and at the same time the multiplication by 0 yields 0, which enables the addition to be performed statically.

3.9 Impure primitive operators, revisited

We are currently experimenting with another residualization policy for impure primitive operators. The idea is that since the call to an impure primitive operator is residualized using a let expression anyway, we have access to the residual identifier naming the residual call. We thus pass it to the dynamic version of the operator, in addition to the operands. The dynamic version can then probe for simplifications and either return a static value if a simplification applies or return the residual identifier otherwise.

For example, an impure version of `mul` can be defined as follows:

```
> (define-primitive! mul!
    ((Int Int) =!> Int)
    *
    (lambda (x1 x2)
      (lambda (r)       ;;; <---***---
        (case-syntax (x1 x2) ()
          [(0 x2)
           0]
          [(x1 0)
           0]
          [(1 x2)
           x2]
          [(x1 1)
           x1]
          [else
           r]))))  ;;; <---***---
> (define baz!
    (lambda (x y)
      (lambda (z f)
        (add x (mul! y (f z))))))
> (tdpe (baz! 10 0) '((Int (Int -!> Int)) =!> Int))
(lambda (n0 x1)
  (let* ([n2 (x1 n0)]
         [n3 (mul! 0 n2)])
    10))
>
```

In this example, the call to `mul!` is residualized but it still yields a static result, which is exploited statically.

4 Related Work

Our work combines a number of ideas, each of which is relatively known by itself. We list them below.

Probing operands: The idea of making functions probe their actual parameters to decide whether to reduce their call or to residualize it is partial-evaluation folklore. Virtually every person who wrote a partial evaluator toyed with it to some degree. It is mentioned, for example, in Section 4.5 of the first publication on type-directed partial evaluation [17]. The present work originates in that section.

Running code with instrumented primitive operators: This idea is as old as Lisp and is used, e.g., for "tracing" function calls and returns. For an other example, in Sussman's Scheme-programming environment today, all primitive operators are instrumented to attempt algebraic simplifications over their operands.[3]

To the best of our knowledge, Berlin was the first to apply this idea — i.e., redefining primitive operators to perform symbolic computation and running a source program directly to specialize it — to partial evaluation. To the best of our knowledge also, this is not explicitly stated in his published work [6, 7, 8].[4]

Berlin's residual programs are essentially first-order, and both his source and residual programs are untyped. Because of that, the user must annotate function calls and conditional expressions in source programs. Also, the partial evaluator avoids code duplication by constructing a graph, which is then unparsed into a Scheme program, using a traditional compiler analysis for eliminating common subexpressions and constructing residual let expressions. In comparison, type-directed partial evaluation is higher-order, typed, and directly inserts let expressions at residualization time.

Guards: The idea is as old as programming. In the area of partial evaluation, Consel was the first to put them to use with filters, to control whether to unfold or to residualize user-defined function calls in his partial evaluator Schism [12]. As for primitive operators in Schism, they are pure and use the standard strategy of only being executed if all their operands are static. In comparison, type-directed partial evaluation unconditionally unfolds all functions (inserting let expressions if their type is annotated with a computational effect) and uses filters only for primitive operators, which may be impure. Both Schism's binding-time analysis and specializer are polyvariant. Type-directed partial evaluation does not have a binding-time analysis, though its binding times are polyvariant. Its specialization strategy, however, is monovariant.

[3] Gerald J. Sussman, personal communication at POPL'96.

[4] Thanks are due to Daniel Weise for the information (personal communication at FCRC'96), and also for pointing out how strongly Berlin's work influenced modern online partial evaluators: indeed, Berlin's strategy is ideal to generate basic blocks, but control structures require careful source annotations, which was deemed impractical. According to Weise, the Fuse project shifted from symbolic execution to symbolic interpretation to automate the annotation process of online partial evaluation [8, 37].

Abstract-syntax rewriting (macros): The idea is as old as programming languages. Macros were fraught with name-capture peril until the advent of hygienic macro-expansion [30]. Our domain-specific language over residual abstract syntax follows the lead of Chez Scheme's syntactic extensions [24], namely, it uses pattern matching, fenders (i.e., guards), and `with`-expressions.

Let insertion: Mogensen suggested to insert residual let expressions to handle partially static values [31]. Bondorf and Danvy put let insertion at the core of the Similix partial evaluator to handle call unfolding in the presence of dynamic actual parameters [10].[5] Let insertion is a cornerstone of call-by-value type-directed partial evaluation [18].

Two-level η-expansion: The idea has been used both in the CPS transformation and in partial evaluation [19, 20].

Partial evaluation: With two exceptions, all other partial evaluators operate over the text of their source programs [14, 28]. The first exception is Berlin's [6, 7]. As described above, it operates by running the source program with instrumented primitive operators. The second exception is "generating extensions," which means "dedicated specializers as obtained by self-application" in the partial-evaluation jargon [14, 28]. In contrast to a general-purpose specializer, a generating extension incurs no interpretive overhead. In our experience, though, at least with Similix, generating extensions are still less efficient than type-directed partial evaluation in practice [9, 21, 34].

Type specialization: Hughes has recently presented a radically new way of looking at and implementing partial evaluation for functional programs [27]. This new approach departs from traditional partial evaluation in that the partial evaluator does not proceed by symbolic interpretation. Instead, it piggy-backs on type inference, as directed by the control and data flows of the source program, and thus does not follow the same steps as a traditional partial evaluator. Type specialization differs from type-directed partial evaluation in that it still operates on the text of source programs,

[5] "Ensuring that dynamic side-effects do not disappear and are not duplicated, and keeping them in the same order as in the source program" is one of the oldest mantras of Similix. Type-directed partial evaluation shares the same mantra.

albeit in an unspecified order. In contrast, a type-directed partial evaluator follows the same steps as a traditional partial evaluator performing symbolic interpretation *and in the same order* — it just does so without interpretive overhead.

Other implementations of type-directed partial evaluation: We distinguish between *meta-level* and *native* implementations.

- A meta-level implementation consists of an interpreter enriched with two-level η-expansion. Altenkirch, Hofmann, and Streicher [1, 2] and Sheard [35] have written such interpreters, the former for pure call by name and the latter for applied call by value. Both also handle polymorphism, and in addition, Sheard's treats inductive data types through a fixed-point operator and "lazy reflection." Lazy reflection amounts to delaying η-expansion for contravariant types, which is possible in a meta-level implementation. Otherwise, Sheard's implementation is online: it offers pure primitive operators that probe their operands with the usual fixed policy. (We also considered that approach in Section 4.5 of our earlier work [17].)

- A native implementation directly processes compiled code. Berger and Schwichtenberg have such an implementation, in Scheme [4, 5]. Filinski does too, in Standard ML,[6] and so do Zhe Yang, also in Standard ML,[7] and Rhiger, in Gofer [34]. In the summer of 1997, Balat and Danvy have combined a native implementation with run-time code generation, in Caml [3]. An ML native implementation cannot offer probing primitive operators since they are fundamentally overloaded. Our (offline) Scheme implementation also handles polymorphism.

According to published numbers for comparable examples (including inductive data types), native implementations perform between 1000 and 10000 times faster than meta-level implementations.

The author's earlier implementations have already been used at other institutions [23, 26]. The present online implementation so far is only used by the author and his students, e.g., for Action Semantics [21, 32, 34]. We find it more flexible and about as efficient, despite the extra processing activity of primitive operators.

[6] Personal communication, spring 1995.

[7] Personal communication, spring 1996.

5 Conclusion and Issues

Because offline type-directed partial evaluation only handles closed terms, it requires the user to close every source program by lambda-abstracting all its free variables, which is awkward in practice. We have extended our type-directed partial evaluator with typed primitive operations (δ-rules), whose default policy is to proceed if all the operands are static and to residualize the operation otherwise. The user can specify the partial-evaluation policy of an operator in two ways: (1) by specifying a filter deciding whether to perform the operation or to residualize it; and (2) by specifying how to residualize the operation. This extension makes type-directed partial evaluation more modular (the user can write or use libraries of primitive operators) and more flexible (the partial-evaluation behaviour of primitive operators is context-sensitive, i.e., their binding times are polyvariant). Online type-directed partial evaluation is also naturally incremental in that while residual programs can be compiled and run, they can also be compiled and further specialized should the opportunity arise.

Contribution: Foremost, we report an online extension of a native implementation of type-directed partial evaluation for Scheme. This extension handles both pure and impure primitive operators, and both their unfolding and residualization strategies can be specified by the user. The former is achieved with filters and the latter through a domain-specific language for residual terms, designed jointly with Morten Rhiger [34]. The resulting implementation meshes smoothly with our earlier implementation of type-directed partial evaluation. It is available from the author on request.[8] In this article, we also have attempted to provide a comprehensive practical overview of type-directed partial evaluation.

Limitations: They are three-fold. Practical: the specialization strategy of type-directed partial evaluation is monovariant [14, 28], and the use of our type-directed partial evaluator does require some skill, since specialization can diverge or yield huge residual programs. Fundamental: only the call-by-name version of type-directed partial evaluation has been formalized. And conceptual: inductive types are still out of reach in all their generality.

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

Acknowledgements

Grateful thanks to Morten Rhiger for many pleasant discussions about `case-syntax`, and to Julia Lawall and Karoline Malmkjær for valuable and timely comments on an earlier version of this article. Thanks are also due to the anonymous reviewers.

References

[1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt and David E. Rydeheard, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199. Springer-Verlag, 1995.

[2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[3] Vincent Balat and Olivier Danvy. Strong normalization by run-time code generation. Technical Report BRICS RS-97-43, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1997.

[4] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993.

[5] 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.

[6] Andrew A. Berlin. A compilation strategy for numerical programs based on partial evaluation. Master's thesis, MIT Artificial Intelligence Laboratory, July 1989. Technical report 1144.

[7] Andrew A. Berlin. Partial evaluation applied to numerical computation. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on*

Lisp and Functional Programming, pages 139–150, Nice, France, June 1990. ACM Press.

[8] Andrew A. Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[9] Anders Bondorf. Similix manual, system version 3.0. Technical Report 91/9, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1991.

[10] 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.

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

[12] Charles Consel. New insights into partial evaluation: the Schism experiment. In Harald Ganzinger, editor, *Proceedings of the Second European Symposium on Programming*, number 300 in Lecture Notes in Computer Science, pages 236–246, Nancy, France, March 1988.

[13] 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.

[14] 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.

[15] Catarina Coquand. From semantics to rules: A machine assisted analysis. In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *Proceedings of CSL'93*, number 832 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[16] Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 1997. To appear.

[17] 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.

[18] Olivier Danvy. Pragmatics of type-directed partial evaluation. Technical Report BRICS RS-96-15, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 1996.

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

[20] 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.

[21] Olivier Danvy and Morten Rhiger. Compiling actions by type-directed partial evaluation. In *Proceedings of the 9th Nordic Workshop on Programming Theory*, Tallinn, Estonia, October 1997.

[22] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Springer-Verlag. Extended version available as the technical report BRICS-RS-96-13.

[23] Scott Draves. *Automatic Program Specialization for Interactive Media.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1997. Technical Report CMU-CS-97-159.

[24] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

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

[26] William L. Harrison and Samuel N. Kamin. Compilation as partial evaluation of functor category semantics. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1997.

[27] John Hughes. Type specialisation for the lambda calculus. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996. Springer-Verlag.

[28] 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.

[29] Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, Bloomington, Indiana, 1986.

[30] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, Cambridge, Massachusetts, August 1986.

[31] Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[32] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[33] 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.

[34] Morten Rhiger. A study in higher-order programming languages. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1997.

[35] Tim Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In Charles Consel, editor, *Proceedings of the ACM*

SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 22–35, Amsterdam, The Netherlands, June 1997. ACM Press.

[36] René Vestergaard. From proof normalization to compiler generation and type-directed change-of-representation. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 1997.

[37] 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. Springer-Verlag.

Recent BRICS Report Series Publications

RS-97-53 **Olivier Danvy.** ***Online Type-Directed Partial Evaluation*****. December 1997. 31 pp. Extended version of an article to appear in** ***Third Fuji International Symposium on Functional and Logic Programming*****, FLOPS '98 Proceedings (Kyoto, Japan, April 2–4, 1998).**

RS-97-52 **Paola Quaglia.** ***On the Finitary Characterization of π-Congruences*****. December 1997. 59 pp.**

RS-97-51 **James McKinna and Robert Pollack.** ***Some Lambda Calculus and Type Theory Formalized*****. December 1997. 43 pp.**

RS-97-50 **Ivan B. Damgård and Birgit Pfitzmann.** ***Sequential Iteration of Interactive Arguments and an Efficient Zero-Knowledge Argument for NP*****. December 1997. 19 pp.**

RS-97-49 **Peter D. Mosses.** ***CASL for ASF+SDF Users*****. December 1997. 22 pp. Appears in** ***ASF+SDF'97, Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshops in Computing*****, http://www.springer.co.uk/ewic/workshops/ASFSDF97. Springer-Verlag, 1997.**

RS-97-48 **Peter D. Mosses.** ***CoFI: The Common Framework Initiative for Algebraic Specification and Development*****. December 1997. 24 pp. Appears in Bidoit and Dauchet, editors,** ***Theory and Practice of Software Development. 7th International Joint Conference CAAP/FASE*****, TAPSOFT '97 Proceedings, LNCS 1214, 1997, pages 115–137.**

RS-97-47 **Anders B. Sandholm and Michael I. Schwartzbach.** ***Distributed Safety Controllers for Web Services*****. December 1997. 20 pp. To appear in** ***European Theory and Practice of Software. 1st Joint Conference FoSSaCS/FASE/ESOP/CC/TACAS*****, ETAPS '97 Proceedings, LNCS, 1998.**

RS-97-46 **Olivier Danvy and Kristoffer H. Rose.** ***Higher-Order Rewriting and Partial Evaluation*****. December 1997. 20 pp. Extended version of paper to appear in** ***Rewriting Techniques and Applications: 9th International Conference*****, RTA '98 Proceedings, LNCS, 1998.**