Type-Directed Partial Evaluation

Olivier Danvy

BRICS *
Department of Computer Science
University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
E-mail: {danvy,tdpe}@brics.dk
Home pages: http://www.brics.dk/~{danvy,tdpe}

Abstract. Type-directed partial evaluation uses a normalization function to achieve partial evaluation. These lecture notes review its background, foundations, practice, and applications. Of specific interest is the modular technique of offline and online type-directed partial evaluation in Standard ML of New Jersey.

1 Background and introduction

1.1 Partial evaluation by normalization

Partial evaluation is traditionally presented as follows [11, 40]. Given a program processor 'run', and a source program p with some input $\langle \mathsf{s},\ \mathsf{d} \rangle$ such that running p on this input yields some output a,

$$\text{run } \mathsf{p}\ \langle \mathsf{s},\ \mathsf{d} \rangle = \mathsf{a}$$

specializing p with respect to $\langle \mathsf{s},\ \_ \rangle$ with a partial evaluator 'PE' yields a residual program $\mathsf{p}_{\langle \mathsf{s},\ \_ \rangle}$ such that running $\mathsf{p}_{\langle \mathsf{s},\ \_ \rangle}$ on the remaining input $\langle \_,\ \mathsf{d} \rangle$ yields the same output a, provided that the source program, the partial evaluator, and the specialized program all terminate. Equationally:

$$\begin{cases} \text{run PE } \langle \mathsf{p},\ \langle \mathsf{s},\ \_ \rangle \rangle = \mathsf{p}_{\langle \mathsf{s},\ \_ \rangle} \\ \text{run } \mathsf{p}_{\langle \mathsf{s},\ \_ \rangle}\ \langle \_,\ \mathsf{d} \rangle = \mathsf{a} \end{cases}$$

The challenge of partial evaluation lies in writing a non-trivial partial evaluator, i.e., one performing the operations in p that depend on s and yielding the corresponding simplified residual program.

* Basic Research in Computer Science (http://www.brics.dk),
Centre of the Danish National Research Foundation.

This requirement reminds one of the concept of normalization in the lambda-calculus [4] and in rewriting systems [26]. Given three terms $e_0 : t_1 \to t_2 \to t_3$, $e_1 : t_1$, and $e_2 : t_2$ such that applying e_0 to e_1 and e_2 yields some result a,

$$e_0\ e_1\ e_2 = a$$

normalizing the result of applying e_0 to e_1 yields a residual term $r : t_2 \to t_3$, such that by construction, applying r to e_2 yields the same result a, provided that applying e_0, normalization, and applying r all converge. Equationally:

$$\begin{cases} e_0\ e_1 = r \\ \quad r\ e_2 = a \end{cases}$$

In these lecture notes, we show how to achieve partial evaluation using normalization in the lambda-calculus. More precisely, we use a *normalization function*, as developed in Section 2.

1.2 Prerequisites and notation

We assume a basic familiarity with partial evaluation, such as that which can be gathered in the present volume. More specifically, we assume that the reader knows that an offline partial evaluator is a two-stage processor with

1. a binding-time analysis that decorates a source program with static and dynamic annotations, and
2. a static reducer that reduces all the static constructs away, yielding a residual program.

We also assume that it is clear to the reader that a binding-time analysis should produce a well-annotated "two-level" term, and that a two-level term is well-annotated if static reduction "does not go wrong" and yields a completely dynamic term.

The canonical example of the power function: The power function, of type

```
int * int -> int
```

maps a integer x (the base parameter) and a natural number n (the exponent parameter) into x^n. It can be programmed in various ways. The version we consider throughout these notes is written in ML [44] as follows.

```
fun power (x, 0) = 1
  | power (x, n) = x * (power (x, n-1))
```

If we want to specialize it with respect to a static value for n, the recursive calls, the conditional expression, and the decrement are classified as static, and the multiplication is classified as dynamic. As a result, the power function is completely unfolded at specialization time. (The multiplication by `1` may or may not be simplified away.) Specializing the power function with respect to $n = 3$, for example, yields the following residual program:

```
fun power_d3 x = x * x * x
```

where the multiplication by `1` was simplified away.

If we want to specialize the power function with respect to a static value for x, the recursive calls, the conditional expression, the decrement, and the multiplication all are classified as dynamic. As a result, the power function is essentially reconstructed at specialization time. (The static value is inlined.) Specializing the power function with respect to $x = 8$, for example, yields the following residual program:

```
fun power_8d 0 = 1
  | power_8d n = 8 * (power_8d (n-1))
```

Lambda-calculus, two-level lambda-calculus: The rest of Section 1 assumes the following grammar for the pure simply typed lambda-calculus:

$$t ::= \alpha \mid t_1 \rightarrow t_2 \mid t_1 \times t_2$$
$$e ::= x \mid \lambda x.e \mid e_0@e_1 \mid \mathrm{pair}(e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e$$

Applications are noted with an infix "@", pairs are constructed with a prefix operator "pair" and projections are noted "π". As for α, it stands for an unspecified atomic type.

The corresponding two-level lambda-calculus is obtained by overlining static syntax constructors and underlining dynamic syntax constructors:

$$\begin{aligned} e ::= x &\mid \overline{\lambda} x.e \mid e_0\overline{@}e_1 \mid \overline{\mathrm{pair}}(e_1, e_2) \mid \overline{\pi_1}\ e \mid \overline{\pi_2}\ e \\ &\mid \underline{\lambda} x.e \mid e_0\underline{@}e_1 \mid \underline{\mathrm{pair}}(e_1, e_2) \mid \underline{\pi_1}\ e \mid \underline{\pi_2}\ e \end{aligned}$$

A "completely static expression" (resp. "completely dynamic expression") is a two-level lambda-term where all syntax constructors are overlined (resp. underlined).

1.3 Two-level programming in ML

We consider the two-level lambda-calculus implemented in ML by representing overlines with ordinary syntax constructs and underlines with constructors of a data type representing residual terms. Let us illustrate this implementation in ML using the data type `Exp.exp` of Figure 1.

For example, the ML expressions

```
fn x => x
Exp.LAM ("x", Exp.VAR "x")
```

respectively represent the completely static expression $\overline{\lambda} x.x$ and the completely dynamic expression $\underline{\lambda} x.x$.

```
structure Exp
= struct
    datatype exp = VAR of string
                 | LAM of string * exp | APP of exp * exp
                 | PAIR of exp * exp | FST of exp | SND of exp
  end
```

Fig. 1. Abstract syntax of residual expressions

Run time: Static reduction is achieved by ML evaluation. For example, the two-level expression $\underline{\lambda} x.(\overline{\lambda} v.v)\overline{@}x$ is represented as

```
Exp.LAM ("x", (fn v => v) (Exp.VAR "x"))
```

This ML expression evaluates to

```
Exp.LAM ("x", Exp.VAR "x")
```

which represents the completely dynamic expression $\underline{\lambda} x.x$.

Compile time: What it means for a two-level expression to be "well-annotated" can be non-trivial [40, 46, 48, 57]. These considerations reduce to the ML typing discipline here. As already mentioned, well-annotatedness boils down to two points:

1. static reduction should not go wrong, and
2. the result should be completely dynamic.

Each of these points is trivially satisfied here:

1. the ML type system ensures that evaluation will not go wrong, and
2. the result is completely dynamic if it has type `Exp.exp`.

Assessment: Implementing the two-level lambda-calculus in ML simplifies it radically. Conceptually, well-annotatedness is reduced to ML typeability and static reduction to ML evaluation. And practically, this implementation directly benefits from existing programming-language technology rather than requiring one to duplicate this technology with a two-level-language processor. It provides, however, no guarantees that the residual program is well typed in any sense.

1.4 Binding-time coercions

The topic of binding-time coercions is already documented in Jens Palsberg's contribution to this volume [47]. Briefly put, a binding-time coercion maps an expression into a new expression to ensure well-annotatedness between expressions and their contexts during static reduction. In that, binding-time coercions fulfill the same task as, e.g., subtype coercions [37].

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

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

Fig. 2. Type-directed binding-time coercions

We are only interested in one thing here: how to coerce a closed, completely static expression into the corresponding dynamic expression. This coercion is achieved using the type-directed translation displayed in Figure 2, which can be seen to operate by "two-level eta expansion" [22, 23]. Given a closed, completely static expression e of type t,

$$\downarrow^{t} e$$

coerces it into its dynamic counterpart. Notationally, the down arrow converts overlines into underlines. We refer to it as "reification."

To process the left-hand side of an arrow, reification uses an auxiliary type-directed translation, which we refer to as "reflection." We write it with an up arrow, to express the fact that it converts underlines into overlines.

In turn, to process the left-hand side of an arrow, reflection uses reification. Reification and reflection are thus mutually recursive. They operate in a type-directed way, independently of their argument.

Examples (of reifying a static expression):

$$
\begin{aligned}
\downarrow^{\alpha \to \alpha} e &= \underline{\lambda} x_1 . e \overline{@} x_1 \\
\downarrow^{((\alpha \times \alpha \to \alpha) \to \alpha) \to \alpha} e &= \underline{\lambda} x_1 . e \overline{@} (\overline{\lambda} x_2 . x_1 \underline{@} (\underline{\lambda} x_3 . x_2 \overline{@}\ \overline{\mathrm{pair}}(\underline{\pi_1}\ x_3, \underline{\pi_2}\ x_3)))
\end{aligned}
$$

In ML, using the data type of Figure 1, these two type-indexed down arrows are respectively expressed as follows:

```
    fn e => LAM ("x1",e (VAR "x1"))
    (* (Exp.exp -> Exp.exp) -> Exp.exp *)

    fn e => LAM ("x1",
                 e (fn x2 => APP (VAR "x1",
                                  LAM ("x3",
                                       x2 (FST (VAR "x3"),
                                           SND (VAR "x3"))))))
(* (((Exp.exp * Exp.exp -> Exp.exp) -> Exp.exp) -> Exp.exp) -> Exp.exp *)
```

Examples (of reflecting upon a dynamic expression):

$$\uparrow_{\alpha\to\alpha} e = \overline{\lambda} x_1 . e \underline{@} x_1$$

$$\uparrow_{((\alpha\times\alpha\to\alpha)\to\alpha)\to\alpha} e = \overline{\lambda} x_1 . e \underline{@} (\underline{\lambda} x_2 . x_1 \overline{@} (\overline{\lambda} x_3 . x_2 \underline{@} \underline{\mathrm{pair}} (\overline{\pi_1}\, x_3, \overline{\pi_2}\, x_3)))$$

In ML, these two type-indexed up arrows are respectively expressed as follows:

```
fn e => fn x1 => APP (e,x1)
(* Exp.exp -> Exp.exp -> Exp.exp *)

fn e => fn x1 => APP (e,
                      LAM ("x2",
                           x1 (fn (x3_1, x3_2)
                               => APP (VAR "x2", PAIR (x3_1, x3_2)))))
(* Exp.exp -> ((Exp.exp * Exp.exp -> Exp.exp) -> Exp.exp) -> Exp.exp *)
```

1.5 Summary and conclusion

We have reviewed the basic ideas of partial evaluation and more specifically of Neil Jones's offline partial evaluation. We have settled on the particular brand of two-level language that arises when one implements "dynamic" with an ML data type representing the abstract syntax of residual programs and "static" with the corresponding ML language constructs. And we have reviewed binding-time coercions and how they are implemented in ML.

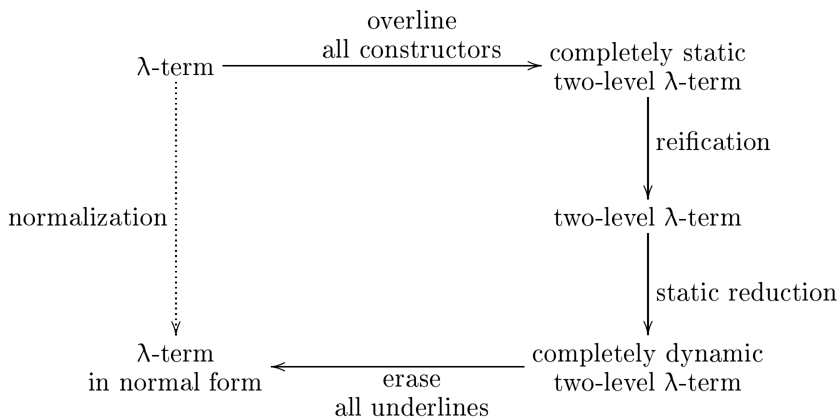
The type of each binding-time coercion matches the type of its input. Therefore, a polymorphic value corresponding to a pure lambda-term is reified into a residual expression by (1) instantiating all its type variables with `Exp.exp` and (2) plugging it in a two-level, code-generating context manufactured in a type-directed way.

2 Normalization by evaluation

In one way or another, the translation displayed in Figure 2 is a familiar sight in the offline-partial-evaluation community [8, 22, 23, 40]. It is, however, also known in other areas of computer science, and we review how in Section 2.1. In Section 2.2, we describe how reification can "decompile" ML values corresponding to pure lambda-terms in normal form. In Section 2.3, we illustrate normalization by evaluation, i.e., how reification can decompile values corresponding to pure lambda-terms into the representation of their normal form. In Section 2.4, we then turn to the implementation of reification and reflection in ML, which is not obvious, since they are type-indexed. Thus equipped, we then consider how to use normalization by evaluation to achieve partial evaluation, as outlined in Section 1.1. However, as analyzed in Section 2.5, normalization by evaluation needs to be adjusted to ML, which we do in Section 2.6. The result is type-directed partial evaluation.

2.1 A normalization function

In logic, proof theory, and category theory, the contents of Figure 2 has been discovered and studied as a *normalization function* [19]. There, the dynamic parts of the two-level lambda-calculus live in a term model, and the static parts live in a lambda-model with term constructors. The normalization function is type-indexed and maps a closed, completely static lambda-term into a closed, completely dynamic lambda-term in normal form:



Ulrich Berger and Helmut Schwichtenberg, for example, discovered this normalization function in the area of proof theory [5]. They also observed that this function provides an efficient way to normalize lambda-terms if the overlines are implemented with ordinary syntax constructs and the underlines are implemented with constructors of residual terms, similarly to what is described in Section 1.3, but in Scheme [42]. In effect, the Scheme evaluator carries out both the normalization steps and the construction of the residual program.

Berger and Schwichtenberg present the normalization function as a left inverse of the evaluation function for the simply typed lambda-calculus; the idea is that an evaluator maps a dynamic lambda-term (an abstract-syntax tree) into its static counterpart (its value), while the normalizer has the inverse functionality. The interested reader is kindly directed to the proceedings of the workshop on "Normalization by Evaluation" for further independent discoveries of this normalization function [1, 12, 13, 19].

In summary, if one implements the two-level lambda-calculus as in Section 1.3, then reifying a simply typed, closed, and completely static higher-order function into a dynamic expression automatically yields a representation of its normal form. In the rest of this section, we illustrate this phenomenon with decompilation, before turning to the implementation of a normalization function in ML.

2.2 Application to decompilation

Reification lets us "decompile" values into the text of a corresponding expression – an observation due to Mayer Goldberg [31, 32].

Analogy with a first-year Scheme exercise: To build an appreciation of programs as data, beginners in Scheme are often asked to write a function constructing a list that represents a Scheme program. One such function is `reify-int-list` that maps a list of numbers into the Scheme list that, when evaluated, will yield that list of numbers. Here is the transcript of an interactive Scheme session illustrating the exercise (">" is the interactive prompt):

```
> (reify-int-list (cons 1 (cons 2 '())))
(cons 1 (cons 2 '()))
> (cons 1 (cons 2 '()))
(1 2)
> '(1 2)
(1 2)
> (reify-int-list '(1 2))
(cons 1 (cons 2 '()))
>
```

There are two issues in this Scheme exercise:

1. algorithmically, `reify-int-list` performs a straightforward recursive descent in the input list; and
2. conceptually, we can read the output list as a Scheme program.

In ML and in Haskell, it is more natural to output an abstract-syntax tree (represented with an inductive data type) and then to unparse it into concrete syntax (represented with a string) and possibly to pretty-print it.

Let us illustrate decompilation in ML without any unparser and pretty-printer, using the data type of Figure 1. We consider several examples in turn, reproducing transcripts of an interactive ML session ("-" is the interactive prompt).

The identity combinator I *at type* $\alpha \to \alpha$

The associated reifier reads $\overline{\lambda} v.\underline{\lambda} x.v \mathbin{\overline{@}} x$.

```
- val I = fn x => x;                    (* the identity combinator *)
val I = fn : 'a -> 'a
- val reify_a2a                          (* the associated reifier *)
      = fn v => Exp.LAM ("x",v (Exp.VAR "x"));
val reify_a2a = fn : (Exp.exp -> Exp.exp) -> Exp.exp
- reify_a2a I;                                    (* decompilation *)
val it = LAM ("x",VAR "x") : Exp.exp
-
```

Compared to `reify-int-list` above, the striking thing here is that we do not decompile first-order values, but higher-order ones, i.e., functions.

The cancellation combinator K *at type* $\alpha \to \beta \to \alpha$

The associated reifier reads $\overline{\lambda}v.\underline{\lambda}x.\underline{\lambda}y.(v\overline{@}x)\overline{@}y$.

```
- fun K x y = x;                              (* the K combinator *)
val K = fn : 'a -> 'b -> 'a
- local open Exp
  in val reify_a2b2a                       (* the associated reifier *)
         = fn v => LAM ("x",LAM ("y",v (VAR "x") (VAR "y")))
  end;
val reify_a2b2a = fn : (Exp.exp -> Exp.exp -> Exp.exp) -> Exp.exp
- reify_a2b2a K;                                   (* decompilation *)
val it = LAM ("x",LAM ("y",VAR "x")) : Exp.exp
-
```

A random higher-order function at type $((\alpha \to \alpha) \to \beta) \to \beta$

The associated reifier reads $\overline{\lambda}v.\underline{\lambda}f.v\overline{@}(\overline{\lambda}v_1.f\underline{@}(\underline{\lambda}x.v_1\overline{@}x))$.

```
- val foo = fn f => f (fn x => x);
val foo = fn : (('a -> 'a) -> 'b) -> 'b
- local open Exp
  in val reify_foo                         (* the associated reifier *)
         = fn v => LAM ("f",
                        v (fn v1 => APP (VAR "f",
                                         LAM ("x",
                                              v1 (VAR "x")))))
  end;
val reify_foo
    = fn : (((Exp.exp -> Exp.exp) -> Exp.exp) -> Exp.exp) -> Exp.exp
- reify_foo foo;                                   (* decompilation *)
val it = LAM ("f",APP (VAR "f",LAM ("x",VAR "x"))) : Exp.exp
-
```

In each case we have decompiled a higher-order function corresponding to a pure lambda-term by reifying it according to its type.

As these examples have just illustrated, decompilation is the inverse of the evaluation function for normal forms [5, 14, 27].

2.3 Normalization by evaluation

Let us now illustrate normalization by evaluation. We consider two source terms that are not in normal form, and how they are reified into a representation of their normal form. The two input values have the same type and thus we normalize them with the same reifier.

In the next interaction, we consider a source term with a beta-redex in the body of a lambda-abstraction. Because of ML's evaluation strategy, the corresponding beta-reduction takes place each time this source term is applied. This reduction, however, can be performed "at normalization time."

```
- reify_a2a (fn x => (fn y => y) x);                 (* a beta-redex *)
val it = LAM ("x",VAR "x") : Exp.exp
-
```

In the next interaction, we revisit the standard definition of the identity combinator I in terms of the Hilbert combinators S and K.

```
- fun S f g x = f x (g x);                       (* the S combinator *)
val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
- reify_a2a (S K K);                 (* reification of S K K into I *)
val it = LAM ("x",VAR "x") : Exp.exp
-
```

2.4 Naive normalization by evaluation in ML

In Sections 1.4, 2.2, and and 2.3, we have written one reifier per type. Let us now turn to implementing a type-indexed normalization function `nbe`, i.e., to writing the contents of Figure 2 in ML. But how does one write a type-indexed function in ML? In Figure 2, reification and reflection very obviously are dependently typed – and ML does not provide any support for dependent types. Fortunately, Andrzej Filinski and Zhe Yang have recently devised the technique of defining reification and reflection pairwise, in a polymorphically typed way [17, 59].

Figure 3 displays an implementation of normalization by evaluation in Standard ML using the Filinski-Yang programming technique. The data type `rr` embodies each reify/reflect pair:

- `rra` denotes the "type constructor" corresponding to the atomic type α (noted `a'` in Figure 4);
- `rrf` denotes the "type constructor" corresponding to functions (infix and noted `-->` in Figure 4); and
- `rrp` denotes the "type constructor" corresponding to products (infix and noted `**` in Figure 4).

Overall, given the representation of a type and a polymorphic value of the corresponding type, normalization by evaluation boils down to reifying the value. For readability, the generator of fresh variables is also initialized in passing.

Examples: Thus equipped, we now revisit the reifiers of Section 2.2.

```
- val reify_a2a = nbe (a' --> a');
val reify_a2a = fn : (Exp.exp -> Exp.exp) -> Exp.exp
- val reify_a2b2a = nbe (a' --> a' --> a');
val reify_a2b2a = fn : (Exp.exp -> Exp.exp -> Exp.exp) -> Exp.exp
- val reify_foo = nbe (((a' --> a') --> a') --> a');
val reify_foo
    = fn : (((Exp.exp -> Exp.exp) -> Exp.exp) -> Exp.exp) -> Exp.exp
-
```

As can be noticed, we only use one atomic type: to repeat the last paragraph of Section 1, all type variables are instantiated with `Exp.exp`.

```
structure Naive_nbe
= struct
    local open Exp
    in datatype 'a rr = RR of ('a -> exp) * (exp -> 'a)

       val rra
           = RR (fn e => e, fn e => e)

       fun rrf (RR (reify1, reflect1), RR (reify2, reflect2))
           = RR (fn f => let val x = Gensym.new "x"
                         in LAM (x, reify2 (f (reflect1 (VAR x))))
                         end,
                 fn e => fn v => reflect2 (APP (e, reify1 v)))

       fun rrp (RR (reify1, reflect1), RR (reify2, reflect2))
           = RR (fn (v1, v2) => PAIR (reify1 v1, reify2 v2),
                 fn e => (reflect1 (FST e), reflect2 (SND e)))

       fun nbe (RR (reify, reflect)) v
           = (Gensym.init (); reify v)
    end
  end
```

Fig. 3. Naive normalization by evaluation in Standard ML (definition)

```
val a' = Naive_nbe.rra

infixr 5 -->
val op --> = Naive_nbe.rrf

infixr 6 **
val op ** = Naive_nbe.rrp

val nbe = Naive_nbe.nbe
```

Fig. 4. Naive normalization by evaluation in Standard ML (interface)

Refinement: We can also decompile a single polymorphic value with respect to more refined types:

```
- nbe ((a' --> a') --> a' --> a') (fn x => x);
val it = LAM ("x1",LAM ("x2",APP (VAR "x1",VAR "x2"))) : Exp.exp
- nbe (a' ** a' --> a' ** a') (fn x => x);
val it = LAM ("x1",PAIR (FST (VAR "x1"),SND (VAR "x1"))) : Exp.exp
-
```

As Guy Steele and Gerry Sussman once said about the Y combinator, "That this manages to work is truly remarkable." [55, page 70].

2.5 Towards type-directed partial evaluation

Now that we have coded normalization by evaluation in ML, we can go back to our initial goal, as stated in Section 1.1: to achieve partial evaluation by partially applying a source function to a static argument and normalizing the result. To this end, we proceed with the following swift generalizations [14]:

1. We use more of ML in our source programs than what corresponds to the pure lambda-calculus. For example, residualizing the open function

   ```
   fn x => (fn i => if i >= 0 then x else x) 42
   ```

   with `reify_a2a` also yields the residual (closed) identity function.
   Our extension, however, must stay reasonable in some sense. For example, residualizing the open function

   ```
   fn x => (print "hello world"; x)
   ```

   with `reify_a2a` also yields the residual identity function, but the string `"hello world"` is output during residualization, which may or may not be what we want.
2. Correspondingly, we can extend the residual syntax of Figure 1 with literals. The reification function at the type of each literal is then simply defined as the corresponding syntax constructor. The reflection function, however, is undefined in general. Indeed, we could only determine an integer-expecting context, for example, by feeding it with an infinite number of integers. As a consequence, we cannot residualize a function such as

   ```
   fn x => x+1
   ```

   This is in contrast with the pure simply typed lambda-calculus where a term can be totally determined by observing the result of plugging it into finitely many contexts – a property which is at the root of normalization by evaluation for the pure simply typed lambda-calculus [3, 19].
3. It is also tempting to use ML's recursion in a source program, even though this introduces the risk of non-termination at partial-evaluation time.
4. Correspondingly, we can code residual recursive functions using fixed-point operators.

5. Finally, ML follows call-by-value, whereas naive normalization by evaluation assumes call-by-name. For example, the possibly non-terminating function

```
fn f => fn x => (fn y => x) (f x)
```

is residualized into the term

```
fn f => fn x => x
```

which denotes a terminating function.
This phenomenon requires us to extend normalization by evaluation with the partial-evaluation technique of dynamic let insertion [7, 35, 43], so that the residual term reads as follows.

```
fn f => fn x => let val _ = f x
                in x
                end
```

Let insertion also solves the problem of computation duplication, which is significant in the presence of residual functions with computational effects [35]. For example, a function such as

```
fn (f, g, h, x) => (fn y => g (y, h x, y)) (f x)
```

is naively residualized into the term

```
fn (f, g, h, x) => g (f x, h x, f x)
```

where the function denoted by `f` is applied twice and out of order with respect to the function denoted by `h`. Let insertion maintains the proper sequencing in the residual program:

```
fn (f, g, h, x) => let val y = f x
                       val z = h x
                   in g (y, z, y)
                   end
```

In his study of normalization by evaluation [29, 30], Andrzej Filinski scrutinizes the generalizations above:

– Normalization by evaluation is defined in a fairly restricted setting – the pure simply typed lambda-calculus. This formal setting needs to be extended to account for base types and for the corresponding operations.
– As soon as we introduce recursion, arguments based on strong normalization cease to apply. The overall picture of partial evaluation by partial application and normalization thus needs to be adjusted.
– Normalization by evaluation is formally defined in a normal-order setting. It needs to be adjusted to work in a call-by-value language, especially in the presence of computational effects such as divergence.

In summary, normalization by evaluation was originally defined for the pure lambda-calculus. It is not immediately clear whether it can be directly transcribed in a richer functional language and still be claimed to work in some sense. Filinski, however, has proven that it can be transcribed in call-by-name PCF [29, 30].

Treating a full-fledged call-by-value functional language such as Scheme and ML thus requires one to adapt normalization by evaluation. This is the goal of Section 2.6.

```
structure Exp
= struct
    datatype exp = VAR of string
                 | LAM of string * exp
                 | APP of exp * exp
                 | PAIR of exp * exp
                 | FST of exp
                 | SND of exp
                 | LET of string * exp * exp
                 | INT of int
                 | BOOL of bool
                 | COND of exp * exp * exp

    fun pp e = ...  (* a pretty-printer *)
  end
```

Fig. 5. Abstract syntax of residual expressions (extended)

2.6 Normalization by evaluation in ML

Because of call-by-value, the standard technique of dynamic let insertion has to be adapted to avoid the computation mismatch illustrated in Item 5 of Section 2.5; this extension makes it possible to handle observationally effectful functions, as well as booleans and more generally disjoint sums [14, 15]. In the rest of these lecture notes, we consider this version of normalization by evaluation, displayed in Figure 6.

Figure 6 requires the extended residual syntax of Figure 5, i.e., let expressions because of call-by-value, integer and boolean literals, and conditional expressions. The overall structure of reification and reflection is the same as in Figure 3. Integers, in particular, are implemented as described in Item 2 of Section 2.5, page 12: the integer reification function is defined as the abstract-syntax constructor for integers, and the integer reflection function raises an uncaught exception. In the rest of the figure, the new parts involve the control operators `shift` and `reset` [21].

`Shift` and `reset` are respectively used to abstract (delimited) control and to delimit control. They are most tellingly used for booleans (`rrb` in Figure 6): the challenge there is to implement the reflection function, which must have the type `exp -> bool`. Since there are only two booleans, we successively provide them to the context, yielding two residual expressions that form the two branches of a residual conditional expression. We lay our hands on the context using the control operator `shift`, which provides us with a functional abstraction of the current context. Supplying a value to this context then reduces to applying the functional abstraction to this value, which we successively do with `true` and `false`. This programming technique is illustrated in Figure 8 and also in the literature [14, 15, 20, 21, 43].

```
structure Nbe =
struct
  local open Exp
  in structure Ctrl = Control (type ans = exp)
     datatype 'a rr = RR of ('a -> exp) * (exp -> 'a)

     val rra = RR (fn e => e, fn e => e)

     fun rrf (RR (reify1, reflect1), RR (reify2, reflect2))
         = RR (fn f => let val x = Gensym.new "x"
                       in LAM (x, Ctrl.reset
                                    (fn ()
                                     => reify2 (f (reflect1 (VAR x)))))
                       end,
               fn e => fn v => let val r = Gensym.new "r"
                               in Ctrl.shift
                                    (fn k
                                     => LET (r,
                                             APP (e, reify1 v),
                                             Ctrl.reset
                                               (fn ()
                                                => k (reflect2 (VAR r)))))
                               end)

     fun rrp (RR (reify1, reflect1), RR (reify2, reflect2))
         = RR (fn (v1, v2) => PAIR (reify1 v1, reify2 v2),
               fn e => (reflect1 (FST e), reflect2 (SND e)))

     exception NoWay
     val rri = (INT,
                fn _ => raise NoWay)

     val rrb
         = RR (BOOL,
               fn e => Ctrl.shift
                         (fn k => COND (e,
                                        Ctrl.reset (fn () => k true),
                                        Ctrl.reset (fn () => k false))))

     fun nbe (RR (reify, reflect)) v
         = (Gensym.init (); reify v)
     fun nbe' (RR (reify, reflect)) e
         = reflect e
  end
end
```

Fig. 6. Normalization by evaluation in Standard ML (definition)

```
val a' = Nbe.rra
val int' = Nbe.rra
val bool' = Nbe.rra

infixr 5 -->; val op --> = Nbe.rrf

infixr 6 **; val op ** = Nbe.rrp

val int = Nbe.rri

val bool = Nbe.rrb

val nbe = Nbe.nbe
val nbe' = Nbe.nbe'
```

Fig. 7. Normalization by evaluation in Standard ML (interface)

```
structure Example
= struct
    structure Ctrl = Control (type ans = int)

    val x1 = 10 + (Ctrl.reset
                     (fn () => 500 + Ctrl.shift
                                       (fn k => (k 0) + (k 100))))
    val x2 = 10 + let fun k v = 500 + v
                  in (k 0) + (k 100)
                  end
  end
```

The two computations above declare an outer context `10 + [ ]`. They also declare a delimited context `[500 + [ ]]`, which is abstracted as a function denoted by `k`. This function is successively applied to `0`, yielding `500`, and to `100`, yielding `600`. The two results are added, yielding `1100` which is then plugged in the outer context. The overall result is `1110`.

In the first computation, the context is delimited by `reset` and the delimited context is abstracted into a function with `shift`. The second computation is the continuation-passing counterpart of the first one [21].

It should be noted that `shift` yields a control abstraction that behaves as a function, i.e., that returns a result to the context of its invocation and thus can be composed. In contrast, the Scheme control operator `call/cc` yields a control abstraction that behaves as a goto, in the sense that invoking it does not return any result to the context of its invocation.

`Shift` and `reset` are documented further in the literature [20, 21, 28].

Fig. 8. An example of using shift and reset

```
signature ESCAPE
= sig
    type void
    val coerce : void -> 'a
    val escape : (('a -> void) -> 'a) -> 'a
  end

structure Escape : ESCAPE
= struct
    datatype void = VOID of void
    fun coerce (VOID v) = coerce v
    fun escape f
        = SMLofNJ.Cont.callcc (fn k => f (fn x => SMLofNJ.Cont.throw k x))
  end

signature CONTROL
= sig
    type ans
    val shift : (('a -> ans) -> ans) -> 'a
    val reset : (unit -> ans) -> ans
  end

functor Control (type ans) : CONTROL =
struct
  open Escape
  exception MissingReset
  val mk : (ans -> void) ref = ref (fn _ => raise MissingReset)
  fun abort x = coerce (!mk x)
  type ans = ans
  fun reset t
      = escape (fn k => let val m = !mk
                        in mk := (fn r => (mk := m; k r));
                           abort (t ())
                        end)
  fun shift h
      = escape (fn k => abort (h (fn v => reset (fn () => coerce (k v)))))
end
```

Fig. 9. Shift and reset in Standard ML of New Jersey [28]

We reproduce Filinski's implementation of `shift` and `reset` in Figure 9. This implementation relies on the Scheme-like control operator `callcc` available in Standard ML of New Jersey [28].

Let insertion: Let us consider the following simple program, where two functions are intertwined, and one is specified to be the identity function:

```
structure Let_example
= struct
    fun main f g x = g (f (g (f (g (f (g x))))))
    fun spec f x = main f (fn a => a) x
  end
```

We residualize `Let_example.spec` according to its most general type as follows:

```
nbe ((a' --> a') --> a' --> a') Let_example.spec
```

The raw result is of type `Exp.exp` and reads as follows.

```
LAM ("x1", LAM ("x2", LET ("r3",
                           APP (VAR "x1",VAR "x2"),
                                LET ("r4",
                                     APP (VAR "x1",VAR "r3"),
                                     LET ("r5",
                                          APP (VAR "x1",VAR "r4"),
                                          VAR "r5")))))
```

The static function has been eliminated statically and let expressions have been inserted to name each intermediate result.

Once unparsed and pretty-printed, the residual program reads as follows.

```
fn x1 => fn x2 => let val r3 = x1 x2
                      val r4 = x1 r3
                  in x1 r4
                  end
```

The attentive reader will have noticed that the output of the pretty-printer is properly tail-recursive, i.e., it does not name the last call.

Booleans: Let us consider function composition:

```
structure Boolean_example
= struct
    fun main f g x = f (g x)
    fun spec f x g = main f g x
  end
```

Residualizing `Boolean_example.spec` with respect to the type

```
(bool' --> bool) --> bool' --> (bool' --> bool') --> bool
```

yields the following residual program:

```
fn x1 => fn x2 => fn x3 => let val r4 = x3 x2
                               val r5 = x1 r4
                           in if r5
                              then true
                              else false
                           end
```

As above, residual let expressions have been inserted. In addition, the boolean variable `r5` has been "eta-expanded" into a conditional expression. This insertion of conditional expressions also has the effect of duplicating boolean contexts, as illustrated next.

Residualizing `Boolean_example.spec` with respect to the type

```
(bool --> bool') --> bool' --> (bool' --> bool) --> bool'
```

yields the following residual program, where the application of `x1` is duplicated in the conditional branches.

```
fn x1 => fn x2 => fn x3 => let val r4 = x3 x2
                           in if r4
                              then x1 true
                              else x1 false
                           end
```

Similarly, residualizing `Boolean_example.spec` with respect to the type

```
(bool' --> bool') --> bool --> (bool --> bool') --> bool'
```

yields the following residual program, where a function definition is cloned in the conditional branches.

```
fn x1 => fn x2 => if x2
                  then fn x3 => let val r4 = x3 true
                                in x1 r4
                                end
                  else fn x6 => let val r7 = x6 false
                                in x1 r7
                                end
```

As an exercise, the reader might want to residualize `Boolean_example.spec` with respect to the type

```
(bool --> bool) --> bool --> (bool --> bool) --> bool.
```

Duplicating boolean contexts is usually justified because of the static computations it may enable. If these are neglectable, one can avoid code duplication by generalizing the static type `bool` into the dynamic type `bool'` defined in Figure 7.

2.7 An alternative approach

Suppose one proscribes booleans and disjoint sums in residual programs. Could one then implement let insertion in a simpler way than with `shift` and `reset`? In June 1998, Eijiro Sumii answered positively to this question.[1]

And indeed what is the goal of `shift` and `reset` in the definition of `rrf` in Figure 6? Essentially to name each intermediate result and to sequentialize its computation. Let us capture this goal with the alternative data type for residual expressions displayed in Figure 10. This data type accounts for lambda-calculus terms, plus a sequential let expression.

Thus equipped, let us observe that each `shift` in `rrf` adds a let binding. But we can obtain the same effect with state instead of with control, by keeping a list of let bindings in a global hook and making the reflect component of `rrf` extend this list:

```
fn e => fn v => let val r = Gensym.new "r"
                in hook := (r, APP (e, reify1 v)) :: !hook;
                   reflect2 (VAR r)
                end
```

Conversely, in the reify component of `rrf`, we can initialize the global list when creating a residual lambda-abstraction and package the list of bindings when completing its residual body:

```
fn f => let val x = Gensym.new "x"
            val previous_hook = !hook
            val _ = hook := []
            val body = reify2 (f (reflect1 (VAR x)))
            val header = rev (!hook)
            val _ = hook := previous_hook
        in LAM (x, LET (header, body))
        end
```

The complete specification is displayed in Figure 11. In practice, it enables one to implement type-directed partial evaluation in a call-by-value functional language without control facilities such as Caml. Sumii's state-based technique also applies for let insertion in traditional syntax-directed partial evaluation, which, according to Peter Thiemann, is folklore.

2.8 Summary and conclusion

In this section, we have presented "normalization by evaluation" and we have adapted it to the call-by-value setting corresponding to our encoding of the two-level lambda-calculus in ML. Similarly, the proof techniques can be (non-trivially) adapted from call-by-name to call-by-value with monadic effects to show the correctness of this variant. Also, in the spring of 1999, Andrzej Filinski has formalized the relation between control-based and state-based let insertion.

We are now ready to use normalization to perform partial evaluation.

[1] Personal communication, Aarhus, Denmark, September 1998.

```
structure Exp_alt
= struct
    datatype exp = VAR of string
                 | LAM of string * exp
                 | APP of exp * exp
                 | LET of (string * exp) list * exp
  end
```

Fig. 10. Alternative abstract syntax of residual expressions

```
structure Nbe_alt =
struct
  local open Exp_alt
  in val hook = ref [] : (string * Exp_alt.exp) list ref
     datatype 'a rr = RR of ('a -> exp) * (exp -> 'a)

     val rra = RR (fn e => e, fn e => e)

     fun rrf (RR (reify1, reflect1), RR (reify2, reflect2))
         = RR (fn f => let val x = Gensym.new "x"
                           val previous_hook = !hook
                           val _ = hook := []
                           val body = reify2 (f (reflect1 (VAR x)))
                           val header = rev (!hook)
                           val _ = hook := previous_hook
                       in LAM (x, LET (header, body))
                       end,
               fn e => fn v => let val r = Gensym.new "r"
                               in hook := (r, APP (e, reify1 v)) :: !hook;
                                  reflect2 (VAR r)
                               end)

     fun nbe (RR (reify, reflect)) v
         = (Gensym.init (); reify v)
     fun nbe' (RR (reify, reflect)) e
         = reflect e
  end
end
```

Fig. 11. Alternative normalization by evaluation in ML

3 Offline type-directed partial evaluation

We define type-directed partial evaluation as normalization by evaluation over ML values, as defined in Figures 6 and 7, pages 15 and 16. Since normalization by evaluation operates over closed terms, we close our source programs by abstracting all their dynamic variables.

In practice, it is a simple matter to close source programs by abstracting all their dynamic free variables. This is naturally achieved by lambda-abstraction in Scheme [14, 25]. In ML, however, it is more natural to use parameterized modules, i.e., functors [44], to abstract the dynamic primitive operators from a source program.

Functors make it possible not only to parameterize a source program with its primitive operators but also with their type, while ensuring a proper binding-time division through the ML typing system.

- Running a source program is achieved by instantiating the corresponding functor with a "standard" interpretation of the domains and the operators to perform evaluation.
- Specializing a source program is achieved by instantiating the corresponding functor with a "non-standard" interpretation to perform partial evaluation.

A residual program contains free variables, namely the primitive operators. We thus unparse it and pretty-print it as a functor parameterized with these operators. We can then instantiate residual programs with the standard interpretation to run them and with the non-standard interpretation to specialize them further, incrementally.

The rest of this session illustrates the practice of offline type-directed partial evaluation. We consider the traditional example of the power function, and we proceed in two steps.

3.1 The power function, part 1/2

In this section, we specialize the power function with respect to its exponent parameter. Therefore its multiplication is dynamic and we abstract it. Figure 12 displays the signature of the abstracted types and primitive operators: integers and multiplication. For typing purposes, we also use a "quote" function to map an actual integer into an abstracted integer. Figure 13 displays the standard interpretation of this signature: it is the obvious one, and thus integers are ML's integers, quoting is the identity function, and multiplication is ML's native multiplication. Figure 14 displays a non-standard interpretation: integers are residual expressions, quoting is the integer constructor of expressions, and multiplication constructs a (named) residual application of the identifier `"mul"` to its two actual parameters, using reflection. Figure 15 displays the actual power function, which is declared in a functor parameterized with the interpretation of integers and of multiplication.

```
signature PRIMITIVE_power_ds
= sig
    type int_

    val qint : int -> int_
    val mul : int_ * int_ -> int_
  end
```

Fig. 12. Signature for abstracted components

```
structure Primitive_power_ds_e : PRIMITIVE_power_ds
= struct
    type int_ = int

    fun qint i = i
    val mul = op *
  end
```

Fig. 13. Standard interpretation: evaluation

```
structure Primitive_power_ds_pe : PRIMITIVE_power_ds
= struct
    local open Exp
    in type int_ = exp

       val qint = INT
       fun mul (e1, e2)
           = nbe' (int' ** int' --> int') (VAR "mul") (e1, e2)
    end
  end
```

Fig. 14. Non-standard interpretation: partial evaluation

```
functor mkPower_ds (structure P : PRIMITIVE_power_ds)
= struct
    local open P
    in fun power (x, n)
           = let fun loop 0 = qint 1
                   | loop n = mul (x, loop (n-1))
             in loop n
             end
    end
  end
```

Fig. 15. Source program

```
structure Power_ds_e
= mkPower_ds (structure P = Primitive_power_ds_e)
```

Fig. 16. Standard instantiation: evaluation

```
structure Power_ds_pe
= mkPower_ds (structure P = Primitive_power_ds_pe)
```

Fig. 17. Non-standard instantiation: partial evaluation

Evaluation: In Figure 16, we instantiate `mkPower_ds` with the standard interpretation of Figure 13. The result is a structure that we call `Power_ds_e`, and in which the identifier `power` denotes the usual power function.

Partial evaluation: In Figure 17, we instantiate `mkPower_ds` with the non-standard interpretation of Figure 14. The result is a structure that we call `Power_ds_pe`.

We specialize the power function with respect to the exponent 3 by partially applying its non-standard version and residualizing the result:

```
nbe (int' --> int') (fn d => Power_ds_pe.power (d, 3))
```

The residual code has the type `Exp.exp` and reads as follows.

```
LAM ("x1",
     LET ("r2",
          APP (VAR "mul",PAIR (VAR "x1",INT 1)),
          LET ("r3",
               APP (VAR "mul",PAIR (VAR "x1",VAR "r2")),
               LET ("r4",
                    APP (VAR "mul",PAIR (VAR "x1",VAR "r3")),
                    VAR "r4"))))
```

This residual code contains free variables. Pretty-printing it (providing the parameters `"mkPower_d3"`, `"PRIMITIVE_power_ds"`, `"power"`, `"qint"`, and `"mul"`) yields a residual program that is closed, more readable, and also directly usable:

```
functor mkPower_d3 (structure P : PRIMITIVE_power_ds)
= struct
    local open P
    in fun power x1
           = let val r2 = mul (x1, qint 1)
                 val r3 = mul (x1, r2)
             in mul (x1, r3)
             end
    end
  end
```

This residual program is ready to be instantiated with `Primitive_power_ds_e` for evaluation or (hypothetically here) with `Primitive_power_ds_pe` for further partial evaluation. Compared to the source program, the recursive function `loop` has been unfolded, as could be expected.

3.2 The power function, part 2/2

In this section, we specialize the power function with respect to its base parameter. All the components of the definition are dynamic and thus we abstract them. Figure 18 displays the signature of the abstracted types and primitive operators: integers, booleans, and the corresponding operations. For typing purposes, we still use a quote function for integers; we also use an "unquote" function for booleans, in order to use ML's conditional expression. Besides the usual arithmetic operators, we also use a call-by-value fixed-point operator to account for the recursive definition of the power function. Figure 19 displays the standard interpretation of this signature: it is the obvious one. Figure 20 displays a non-standard interpretation: integers and booleans are residual expressions, quoting is the integer constructor of expressions, and unquoting a boolean expression reflects upon it at boolean type. As for the primitive operators, they construct residual applications of the corresponding identifier to their actual parameters. Figure 21 displays the actual power function, which is declared in a parameterized functor.

Evaluation: In Figure 22, page 27, we instantiate `mkPower_sd` with the standard interpretation of Figure 19. The result is a structure that we call `Power_sd_e`, and in which the identifier `power` denotes the usual power function.

Partial evaluation: In Figure 23, page 27, we instantiate `mkPower_sd` with the non-standard interpretation of Figure 20. The result is a structure that we call `Power_sd_pe`.

We specialize the power function with respect to the base 8 by partially applying its non-standard version and residualizing the result:

```
nbe (int' --> int') (fn d => Power_sd_pe.power (8, d))
```

Pretty-printing the residual code yields the following residual program, which is similar to the source program of Figure 21 except that the base parameter has disappeared, `8` has been inlined in the induction case, and let expressions have been inserted.

```
signature PRIMITIVE_power_sd
= sig
    type int_
    type bool_

    val qint : int -> int_
    val ubool : bool_ -> bool
    val dec : int_ -> int_
    val mul : int_ * int_ -> int_
    val eqi : int_ * int_ -> bool_
    val fix : ((int_ -> int_) -> int_ -> int_) -> int_ -> int_
  end
```

Fig. 18. Signature for abstracted components

```
structure Primitive_power_sd_e : PRIMITIVE_power_sd
= struct
    type int_ = int
    type bool_ = bool

    fun qint i = i
    fun ubool b = b
    fun dec i = i-1
    val mul = op *
    val eqi = op =
    fun fix f x = f (fix f) x (* fix is a CBV fixed-point operator *)
  end
```

Fig. 19. Standard interpretation: evaluation

```
structure Primitive_power_sd_pe : PRIMITIVE_power_sd
= struct
    local open Exp
    in type int_ = exp
       type bool_ = exp

       val qint = INT
       val ubool = nbe' bool
       val dec = nbe' (int' --> int') (VAR "dec")
       val mul = nbe' (int' ** int' --> int') (VAR "mul")
       val eqi = nbe' (int' ** int' --> int') (VAR "eqi")
       val fix = nbe' (((int' --> int') --> int' --> int')
                       --> int' --> int')
                      (VAR "fix")
    end
  end
```

Fig. 20. Non-standard interpretation: partial evaluation

```
functor mkPower_sd (structure P : PRIMITIVE_power_sd)
= struct
    local open P
    in fun power (x, n)
           = fix (fn loop => fn n => if ubool (eqi (n, qint 0))
                                     then qint 1
                                     else mul (qint x, loop (dec n)))
                 n
    end
  end
```

Fig. 21. Source program

```
structure Power_sd_e
= mkPower_sd (structure P = Primitive_power_sd_e)
```

Fig. 22. Standard instantiation: evaluation

```
structure Power_sd_pe
= mkPower_sd (structure P = Primitive_power_sd_pe)
```

Fig. 23. Non-standard instantiation: partial evaluation

```
functor mkPower_8d (structure P : POWER)
= struct
    local open P
    in fun power x1
           = let val r2 = fix (fn x3
                                => fn x4
                                   => let val r5 = eqi (x4, qint 0)
                                      in if ubool r5
                                         then qint 1
                                         else let val r6 = dec x4
                                                  val r7 = x3 r6
                                              in mul (qint 8, r7)
                                              end)
             in r2 x1
             end
    end
  end
```

As in Section 3.1, this residual program is as could be expected.

3.3 Summary and conclusion

To use offline type-directed partial evaluation, one thus

1. specifies a signature for dynamic primitive operators and the corresponding types;
2. specifies their evaluation and their partial evaluation;
3. parameterizes a source program with these primitive operators and types;
4. instantiates the source program with the partial-evaluation operators and types, and residualizes a value at an appropriate type; and
5. pretty-prints the result into a parameterized residual program.

The first results of offline type-directed partial evaluation have been very encouraging: it handles the standard examples of the trade (i.e., mostly, the first and the second Futamura projections) with an impressive efficiency. Its functionality is otherwise essentially the same as Lambda-Mix's: higher-order monovariant specialization over closed programs [39]. Its use, however, is considerably more convenient since the binding-time separation of each source program is guided and ensured by the ML type system. There is therefore no need for expert binding-time improvements [40, Chapter 12]. In fact, we believe that this disarming ease of use is probably the main factor that has let offline type-directed partial evaluation scale up, as illustrated in the work of Vestergaard and the author [25] and of Harrison and Kamin [34].

In practice, however, offline type-directed partial evaluation imposes a restriction on its user: the binding-time signatures of primitive operators must be monovariant. This restriction forces the user to distinguish between "static" and "dynamic" occurrences of primitive operators in each source program. Against this backdrop, we have turned to the "online" flavor of partial evaluation, where one abstracts the source program completely and makes each primitive operator probe its operands for possible simplifications. This is the topic of Section 4.

4 Online type-directed partial evaluation

A partial evaluator is online if its operators probe their operands dynamically to decide whether to perform an operation at partial-evaluation time or to residualize it until run time [58]. In his PhD thesis [52], Erik Ruf described how to obtain the best of both offline and online worlds:

- on the one hand, one can trust the static information of the binding-time analysis since it is safe; and
- on the other hand, one should make dynamic operators online because a binding-time analysis is conservative.

The idea applies directly here: in Figure 14, page 14, if we define multiplication to probe its operands, we can naturally look for obvious simplifications, as in Figure 24. (NB: the simplifications by zero are safe because of let insertion.)

Specializing `Power_ds_pe.power` (in Figure 15, page 23 and in Figure 17, page 24) with respect to `3` then yields the following simpler residual program.

```
structure Primitive_power_ds_pe : PRIMITIVE_power_ds
= struct
    local open Exp
    in type int_ = exp
       val qint = INT
       fun mul (INT i1, INT i2) = INT (i1 * i2)
         | mul (INT 0, _) = INT 0
         | mul (_, INT 0) = INT 0
         | mul (INT 1, e2) = e2
         | mul (e1, INT 1) = e1
         | mul e = nbe' (int' ** int' --> int') (VAR "mul") e
    end
  end
```

Fig. 24. Online version of Figure 14, page 23

```
functor mkPower_d3 (structure P : POWER)
= struct
    local open P
    in fun power x1 = let val r2 = mul (x1, x1)
                      in mul (x1, r2)
                      end
    end
  end
```

Compared with the earlier definition of `mkPower_d3`, page 25, the vacuous multiplication of `x1` by `1` has been simplified away.

In the rest of this section, we illustrate online type-directed partial evaluation with two case studies. In Section 4.1, we consider a very simple example where the uses of a primitive operator need not be split into static and dynamic occurrences, which is more practical. And in Section 4.2, we revisit the power function: this time, we abstract all of its operators and we make them online. This makes it possible to specialize the *same* source program with respect to either the base parameter or the exponent parameter. On the way, we come across the familiar tension between unfolding and residualizing recursive function calls, as epitomized by Schism's filters [10].

4.1 Online simplification for integers

Figure 25 displays the signature of a minimal implementation of integers: a type `int_`, a quote function for integer literals, and an addition function.

Figure 26 displays the obvious standard interpretation of integers: `int_` is instantiated to be the type of integers, `qint` is defined as the identity function, and `add` is defined as addition.

Figure 27 displays a non-standard interpretation of integers where `int_` is instantiated to be the type of residual expressions, `qint` is defined as the integer

```
signature PRIMITIVE1
= sig
    type int_

    val qint : int -> int_
    val add : int_ * int_ -> int_
  end
```

Fig. 25. Signature for integers

```
structure Primitive1_e : PRIMITIVE1
= struct
    type int_ = int

    fun qint x = x
    val add = op +
  end
```

Fig. 26. Standard interpretation for integers: evaluation

```
structure Primitive1_pe : PRIMITIVE1
= struct
    local open Exp
    in type int_ = exp

       val qint = INT
       fun add (INT i1, INT i2)
           = INT (i1+i2)
         | add (INT 0, e2)
           = e2
         | add (e1, INT 0)
           = e1
         | add e
           = nbe' (int' ** int' --> int') (VAR "add") e
    end
  end
```

Fig. 27. Non-standard interpretation for integers: partial evaluation

```
functor mkEx1 (structure P : PRIMITIVE1)
= struct
    local open P
    in fun main x y = add (add (x, qint 10), y)
       val spec = main (qint 100)
    end
  end
```

Fig. 28. Sample source program

constructor, and add is defined as a mapping of two integer-typed expressions into a simplified integer-typed expression. If there is nothing to simplify, then the variable "add" is reflected upon at type int' ** int' --> int' and the result is applied to the argument of add, which is a pair of expressions. The result is an expression.

Thus equipped, let us consider the source program of Figure 28. It is parameterized by the implementation of integers specified in Figure 25. It involves two literals, 10 and 100, both of which are quoted. Our goal is to residualize the value of spec. It thus should appear clearly that the inner occurrence of add is applied to two static integers and that the outer occurrence is applied to a static integer and a dynamic one.

Evaluation: Instantiating mkEx1 with Prim1_e for P yields a structure that we call Ex1_e. Applying Ex1_e.spec to 1000 yields 1110, which has the type Prim1_e.int_.

Partial evaluation: Instantiating mkEx1 with Prim1_pe for P yields a structure that we call Ex1_pe. Residualizing Ex1_pe.spec at type int' --> int' yields

```
LAM ("x1", APP (VAR "add", PAIR (INT 110,VAR "x1")))
```

which has the type Exp.exp.

Pretty-printing this residual code (providing "mkEx1'", "PRIMITIVE1", "spec", and "qint") yields the following more readable residual program:

```
functor mkEx1' (structure P : PRIMITIVE1)
= struct
    local open P
    in fun spec x1
           = add (qint 110, x1)
    end
  end
```

Compared to the source program, the inner addition has been simplified.

4.2 The power function, revisited

We now reconsider the canonical example of the power function. To this end, we need integers, booleans, decrement, multiplication, integer equality, and a recursion facility. Again, we use a quote function for integers, an unquote function for booleans, and a call-by-value fixed-point operator over functions of type int_ -> int_ for recursion. This paraphernalia is summarized in the signature of Figure 29.

The standard interpretation for evaluation is the obvious one and thus we omit it.

Figure 30 displays the non-standard interpretation for partial evaluation. The only remarkable point is the definition of fix, which embodies our unfolding strategy: if the exponent is known, then the call to fix should be unfolded; otherwise, it should be residualized.

The (parameterized) source program is displayed in Figure 31.

```
signature POWER
= sig
    type int_
    type bool_

    val qint : int -> int_
    val ubool : bool_ -> bool

    val dec : int_ -> int_
    val mul : int_ * int_ -> int_
    val eqi : int_ * int_ -> bool_
    val fix : ((int_ -> int_) -> int_ -> int_) -> int_ -> int_
  end
```

Fig. 29. Signature of primitive operations for the power function

```
structure Primitive_power_pe : POWER
= struct
    local open Exp
    in type int_ = exp
       type bool_ = exp

       val qint = INT
       fun ubool (BOOL true) = true
         | ubool (BOOL false) = false
         | ubool e = nbe' bool e

       fun dec (INT i) = INT (i-1)
         | dec e = nbe' (int' --> int') (VAR "dec") e
       fun mul (INT i1, INT i2) = INT (i1 * i2)
         | mul (INT 0, _) = INT 0
         | mul (_, INT 0) = INT 0
         | mul (INT 1, e) = e
         | mul (e, INT 1) = e
         | mul e = nbe' (int' ** int' --> int') (VAR "mul") e
       fun eqi (INT i1, INT i2) = BOOL (i1=i2)
         | eqi e = nbe' (int' ** int' --> bool') (VAR "eqi") e
       fun fix f (x as (INT _))
           = Fix.fix f x  (* Fix.fix is a CBV fixed-point operator *)
         | fix f x
           = nbe' (((int' --> int') --> int' --> int')
                   --> int' --> int')
                  (VAR "fix") f x
    end
  end
```

Fig. 30. Non-standard interpretation for the power function: partial evaluation

```
functor mkPower (structure P : POWER)
= struct
    local open P
    in fun power (x, n)
            = fix (fn loop => fn n => if ubool (eqi (n, qint 0))
                                      then qint 1
                                      else mul (x, loop (dec n))) n
    end
  end
```

Fig. 31. Source program

Evaluation: Instantiating `mkPower` with a standard interpretation for `P` yields the usual power function.

Partial evaluation: Instantiating `mkPower` with `Primitive_power_pe` for `P` yields a structure that we call `Power_pe`.

```
structure Power_pe
= mkPower (structure P = Primitive_power_pe)
```

Let us specialize `Power_pe.power` with respect to its second parameter:

```
val power_d3 = let val power = Power_pe.power
                   val qint = Primitive_power_pe.qint
               in nbe (int' --> int') (fn x => power (x, qint 3))
               end
```

The exponent is static and thus all the calls to `loop` are unfolded. Also, to account for call-by-value, let expressions are inserted to name all intermediate function calls. Finally, in the base case, the primitive operator `mul` is given the opportunity to simplify a multiplication by `1`. The result is identical to that obtained in the introduction to Section 4, page 29.

Let us specialize `Power_pe.power` with respect to its first parameter:

```
val power_8d = let val power = Power_pe.power
                   val qint = Primitive_power_pe.qint
               in nbe (int' --> int') (fn n => power (qint 8, n))
               end
```

The exponent is dynamic and thus the calls to `loop` are residualized. The residual code is thus essentially the same as the source code, modulo the facts that (1) let expressions are inserted to name all intermediate function calls, and (2) the literal `8` is inlined. The result is identical to that obtained in Section 3.2, page 27.

From the same source program, we thus have obtained the same results as in Section 3, where we considered two distinct binding-time annotated versions of `power`.

4.3 Summary and conclusion

Online type-directed partial evaluation extends offline type-directed partial evaluation by making the abstracted operators probe their operands for possible simplifications. As we pointed out elsewhere [16], this probing idea is partial-evaluation folklore.

In this section, we have pushed the online idea to its natural limit by making source programs completely closed: all variables are either local to the source program or they are declared through its parameters. Declaring recursive functions through fixed points has forced us to address their unfolding policy by guarding the call to each fixed-point operator, in a manner reminiscent of Schism's filters. (Of course, the same could be said of all the primitive operators that pattern match their operands.) More stylistically, specifying a programming-language interpreter as a functor parameterized by structures nicely matches the format of denotational semantics, i.e., domains, semantic algebras, and valuation functions [53], making type-directed partial evaluation a convenient and effective "semantic back-end."

In practice, divergence and code duplication are the main problems one must address when using type-directed partial evaluation:

- divergence is dealt with by guarding each fixed-point operator and possibly by using several distinct instances; and
- code duplication arises from conditional expressions and is dealt with by generalizing boolean types into the dynamic type `bool'`.

Turning to performance, one might wonder how much the online overhead penalizes type-directed partial evaluation. The answer is: less than one might think, since in our experience, an online type-directed partial evaluator is noticeably more efficient than an offline syntax-directed partial evaluator such as Similix [24].

5 Incremental type-directed partial evaluation

The goal of this section is to spell out the mechanics of incremental type-directed partial evaluation. We consider the following function `super_power`.

```
fun super_power (s3, s2, s1)
    = power (s3, power (s2, s1))
```

We specialize `super_power` with respect to `s1 = 3`. Then we specialize the result with respect to `s2 = 2`, obtaining the same result as if we had directly specialized `super_power` with respect to `s1 = 3` and `s2 = 2`.

The source program: The source program is displayed in Figure 32. It uses the functor `mkPower` of Figure 31, page 33.

```
functor mkSuperPower_ddd (structure P : POWER)
= struct
    local structure Power = mkPower (structure P = P)
    in fun main (s3, s2, s1)
           = Power.power (s3, Power.power (s2, s1))
    end
  end
```

Fig. 32. Source program: mkSuperPower_ddd

```
structure SuperPower_ddd_pe
= mkSuperPower_ddd (structure P = Primitive_power_pe)
```

Fig. 33. Instantiation of mkSuperPower_ddd

```
functor mkSuperPower_dd3 (structure P : POWER)
= struct
    local open P
    in fun main (x0, x1)
           = let val r2 = mul (x1, x1)
                 val r3 = mul (x1, r2)
                 val r4 = fix (fn x5
                              => fn x6
                                 => let val r7 = eqi (x6, qint 0)
                                    in if ubool r7
                                       then qint 1
                                       else let val r8 = dec x6
                                                val r9 = x5 r8
                                            in mul (x0, r9)
                                            end
                                    end)
             in r4 r3
             end
    end
  end
```

Fig. 34. Residual program: mkSuperPower_dd3

```
structure SuperPower_dd3_pe
= mkSuperPower_dd3 (structure P = Primitive_power_pe)
```

Fig. 35. Instantiation of mkSuperPower_dd3

```
functor mkSuperPower_d23 (structure P : POWER)
= struct
    local open P
    in fun main x1
           = let val r2 = mul (x1, x1)
                 val r3 = mul (x1, r2)
                 val r4 = mul (x1, r3)
                 val r5 = mul (x1, r4)
                 val r6 = mul (x1, r5)
                 val r7 = mul (x1, r6)
             in mul (x1, r7)
             end
    end
  end
```

Fig. 36. Residual program: `mkSuperPower_d23`

First degree of specialization: In Figure 33, we instantiate `mkSuperPower_ddd` for partial evaluation. We then specialize the main function with respect to its third argument:

```
let val main = SuperPower_ddd_pe.main
    val qint = Primitive_power_pe.qint
in nbe (int' ** int' --> int') (fn (x, y) => main (x, y, qint 3))
end
```

The residual program is displayed in Figure 34. The inner occurrence of `power` has been specialized away, and the outer occurrence has been inlined in the residual program.

Second degree of specialization: In Figure 35, we instantiate `mkSuperPower_dd3` for further partial evaluation. We then specialize the main function with respect to its second argument:

```
let val main = SuperPower_dd3_pe.main
    val qint = Primitive_power_pe.qint
in nbe (int' --> int') (fn x => main (x, qint 2))
end
```

The residual program is displayed in Figure 36. The remaining occurrence of `power` has been specialized away.

Both degrees of specialization at once: We would have obtained textually the same residual program as in Figure 36 by specializing the original main function with respect to both its static arguments:

```
let val main = SuperPower_ddd_pe.main
    val qint = Primitive_power_pe.qint
in nbe (int' --> int') (fn x => main (x, qint 2, qint 3))
end
```

6 Type-directed partial evaluation and the cogen approach

The question often arises how type-directed partial evaluation and the cogen approach compare. In this section, we situate type-directed partial evaluation within the cogen approach.

6.1 On normalization by evaluation

At the core of type-directed partial evaluation, the notion of static reduction matches the notion of evaluation in a functional language. This match also holds in a simply typed system, making it possible for it to ensure that static reduction will not go wrong.

Such a correspondence, however, does not hold in general. For example, it does not in the two-level functional languages Flemming and Hanne Nielson consider in their book [46]. These two-level languages have their own notions of binding times and of static reduction. For each of them, a dedicated binding-time analysis and the corresponding static reducer need to be studied [45, 48, 57].

In contrast, type-directed partial evaluation results from a deliberate effort to make static reduction and evaluation coincide. In a type-directed partial evaluator, static expressions are thus represented as native code (see Section 1.3). Therefore, static reduction takes place at native speed, and specialization using a type-directed partial evaluator can be quite efficient.

6.2 On type-directed partial evaluation

Type-directed partial evaluation builds on normalization by evaluation by introducing primitive operations, thus staging source programs into user-defined functions and primitive operators, as in Schism and Similix. Primitive operations are then instantiated either for evaluation or for partial evaluation. Specialization is still carried out by evaluation, with a binding-time discipline that still corresponds to the simply typed λ-calculus: static or dynamic base types, and static (but no dynamic) compound type constructors.

6.3 From traditional partial evaluation to the cogen approach

In a traditional partial evaluator, static values are represented symbolically and static reduction is carried out by symbolic evaluation. Therefore, specialization takes place with a certain interpretive overhead. Against this backdrop, the "cogen approach" was developed to represent static values natively.

Let us reconsider the partial-evaluation equation of Section 1.1.

$$\text{run PE}\ \langle \mathtt{p},\ \langle \mathtt{s},\ \_\rangle\rangle = \mathtt{p}_{\langle \mathtt{s},\ \_\rangle}$$

Traditionally [40], a partial evaluator operates as an interpreter. However, both for expressiveness and for efficiency, modern partial evaluators such as ML-Mix [6], pgg [56], and Tempo [9] specialize any program p by first constructing a dedicated partial evaluator $\text{PE}_{\langle p,\,\_\rangle}$ and then running $\text{PE}_{\langle p,\,\_\rangle}$ on the static input.

$$\text{run PE}_{\langle p,\,\_\rangle}\ \langle s,\ \_\rangle = p_{\langle s,\,\_\rangle}$$

Such dedicated partial evaluators are called "generating extensions." Generating extensions are constructed by a program traditionally named "cogen," and the overall approach is thus called "the cogen approach."

The cogen approach was developed for offline partial evaluators, and thus cogen usually operates on binding-time analyzed (i.e., two-level) source programs.

Let us assume a binding-time analysis that makes it possible to implement two-level terms as we have done in Section 1.3:

- static expressions are translated into syntactic constructs ($\overline{\lambda}$ into `fn`, etc.), giving rise to native values; and
- dynamic expressions are translated into constructors of residual syntax ($\underline{\lambda}$ into `LAM`, etc.).

The resulting two-level programs can then be compiled and run at native speed, just as with type-directed partial evaluation. Furthermore, if the binding-time analysis inserts binding-time coercions [36, 47], generating extensions have the same performance and produce the same residual programs as type-directed partial evaluation. This property has been verified in practice by Morten Rhiger and the author for Action Semantics [24] and also, independently, by Simon Helsen and Peter Thiemann [36].

6.4 A first example

Let us compare the cogen approach and type-directed partial evaluation on an example that does not require any binding-time coercion. Applying the cogen approach to the (static) identity function $\overline{\lambda}x.x$ yields the following generating extension.

```
let val x = Gensym.new "x"
in Exp.LAM (x, Exp.VAR x)
end
```

In comparison, residualizing the identity function at type $\alpha \to \alpha$ amounts to plugging it into a context induced by its type, i.e., passing it to the following function (that implements $\downarrow^{\alpha\to\alpha}$).

```
fn f => let val x = Gensym.new "x"
        in Exp.LAM (x, f (Exp.VAR x))
        end
```

Modulo some administrative reductions, the two approaches work identically here.

6.5 A second example

Let us compare the cogen approach and type-directed partial evaluation on an example that does require a binding-time coercion:

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

g is dynamic, and therefore the context $g@[\cdot]$ has to be dynamic. $\lambda a.a$ flows both to the evaluation context $[\cdot]@(g@f)$ where it can be considered static, and to the context $g@[\cdot]$, where it must be considered dynamic. As illustrated in Jens Palsberg's chapter [47], the binding-time analysis can either classify $\lambda a.a$ as dynamic, since it will flow into a dynamic context, or classify it as static and coerce the dynamic context into a static one with the two-level eta-redex $\underline{\lambda} x.[\cdot]\overline{@}x$.

Below, we can see that binding-time coercion by two-level eta-expansion is immaterial in type-directed partial evaluation, whereas it is an issue in the cogen approach.

Type-directed partial evaluation: Residualizing the term above at type $((\alpha \to \alpha) \to \alpha) \to \alpha$ yields the following optimal residual term.

$$\underline{\lambda} g.g\underline{@}(\underline{\lambda} x.x)$$

Cogen without binding-time improvement: Binding-time analysis without any coercion yields the following two-level term, where $\lambda a.a$ is dynamic.

$$(\overline{\lambda} f.\underline{\lambda} g.f\underline{@}(g\underline{@}f))\overline{@}(\underline{\lambda} a.a)$$

And static reduction yields the following sub-optimal residual term.

$$\underline{\lambda} g.(\underline{\lambda} a.a)\underline{@}(g\underline{@}(\underline{\lambda} a.a))$$

Cogen with binding-time improvement: Binding-time analysis with a coercion yields the following two-level term, where $\lambda a.a$ is static. (The coercion is put in a box, as an aid to the eye.)

$$(\overline{\lambda} f.\underline{\lambda} g.f\overline{@}(g\underline{@}(\boxed{\underline{\lambda} x.f\overline{@}x})))\overline{@}(\overline{\lambda} a.a)$$

And static reduction yields the same optimal result as type-directed partial evaluation:

$$\underline{\lambda} g.g\underline{@}(\underline{\lambda} x.x)$$

Modulo binding-time coercions, the two approaches thus work identically here.

6.6 Summary and conclusion

In their independent comparison between type-directed partial evaluation and the cogen approach [36], Simon Helsen and Peter Thiemann observe that both specializers yield equivalent results in comparable time in the presence of binding-time coercions. Now unlike in the cogen approach, type-directed partial evaluation does not involve creating a textual two-level program and compiling it. Nevertheless we believe that type-directed partial evaluation can be viewed as an instance of the cogen approach, where static reduction is carried out by evaluation. This instance is very simple, using a binding-time discipline that corresponds to the simply typed λ-calculus and does not necessitate explicit binding-time coercions.

This relationship between type-directed partial evaluation and the cogen approach is not accidental, as type-directed partial evaluation grew out of binding-time improvements [22, 23]. And indeed, as the reader can see, the equations defining binding-time coercions are the same ones as the equations defining type-directed partial evaluation (see Figure 2). These coercions can serve as an independent specialization mechanism because they implement a normalization function.

7 Conclusion and issues

Type-directed partial evaluation is still largely a topic under exploration. It stems from a normalization function operating on values instead of on symbolic expressions (i.e., annotated abstract-syntax trees), as is usual in traditional, syntax-directed partial evaluation. This normalization function in effect propagates constants and unfolds function calls. The user is left with deciding the policy of unfolding recursive function calls through the corresponding fixed-point operators. Otherwise, a type-directed partial evaluator provides essentially the same functionality as Lambda-Mix [39], though in a statically typed setting which makes much for its ease of use.

Type-directed partial evaluation was first developed in Scheme, and amounted to achieving specialization by Scheme evaluation [14–16]. Andrzej Filinski, and then Zhe Yang [59] and Morten Rhiger [49, 50] found how to express it in a Hindley-Milner type setting, i.e., in ML and in Haskell, thus achieving specialization by ML and Haskell evaluation. In addition, the Hindley-Milner typing system ensures that specialization will not go wrong. Then Kristoffer Rose expressed type-directed partial evaluation in Haskell using type classes [51] and Belmina Dzafic formalized it in Elf [27]. Andrzej Filinski, Zhe Yang, and Morten Rhiger also noticed that type-directed partial evaluation in ML could be made online by pattern matching over the residual abstract syntax. A more comprehensive review of related work is available elsewhere [16]. There, we distinguish between native and meta-level type-directed partial evaluation: a native implementation, such as the one presented here, uses an underlying evaluator, whereas a meta-level implementation uses an interpreter [1, 54]. This choice entails the usual tradeoff between flexibility and efficiency.

The most sizeable applications of type-directed partial evaluation so far involve the Futamura projections, type specialization, and run-time code generation [2, 17, 24, 25, 33, 34]. Having made type-directed partial evaluation online has improved its usability, but it is still limited because it only provides monovariant program-point specialization (as opposed to polyvariant program-point specialization as in Similix [41]) and does not handle inductive types very naturally.

An extended version of this chapter is available in the BRICS series [18].

Acknowledgements

Thanks to the organizers and to the participants of the summer school for a pleasant event, to Neil Jones, for his encouragement, and to the editors of this volume, especially Peter Thiemann, for their patience.

I am also grateful to Andrzej Filinski for several substantial rounds of critical comments; alas time was too short to address them all.

Finally, thanks are due to several participants of the BRICS Programming-Language Café (`http://www.brics.dk/~danvy/PLC/`): Belmina Dzafic, Daniel Damian, Niels O. Jensen, Lasse R. Nielsen, and Morten Rhiger, and also to Lars R. Clausen, Julia L. Lawall, Karoline Malmkjær, Eijiro Sumii, and Zhe Yang for their timely feedback.

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 LNCS, pages 182–199. Springer-Verlag, 1995.
2. Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in LNCS, pages 240–252. Springer-Verlag, 1998.
3. Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.
4. Henk P. Barendregt. *Functional Programming and Lambda Calculus*, chapter 7, pages 321–364. Volume B of van Leeuwen [38], 1990.
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. IEEE Computer Society Press, 1991.
6. Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1993. DIKU Rapport 93/22.
7. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.
8. Anders Bondorf and Jens Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.

9. The COMPOSE Project. Effective partial evaluation: Principles and applications. Technical report, IRISA (www.irisa.fr), Campus Universitaire de Beaulieu, Rennes, France, January 1996 – May 1998. A selection of representative publications.
10. 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 LNCS, pages 236–246. Springer-Verlag, 1988.
11. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501. ACM Press, 1993.
12. Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.
13. Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.
14. 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. ACM Press, 1996.
15. Olivier Danvy. Pragmatics of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in LNCS, pages 73–94. Springer-Verlag, 1996. Extended version available as the technical report BRICS RS-96-15.
16. Olivier Danvy. Online type-directed partial evaluation. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 271–295. World Scientific, 1998. Extended version available as the technical report BRICS RS-97-53.
17. Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, number 1443 in LNCS, pages 908–917. Springer-Verlag, 1998.
18. Olivier Danvy. Type-directed partial evaluation. Lecture Notes BRICS LN-98-3, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998. Extended version of the present lecture notes.
19. Olivier Danvy and Peter Dybjer, editors. *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Chalmers, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998.
20. 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. ACM Press, 1990.
21. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.
22. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995.
23. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.
24. Olivier Danvy and Morten Rhiger. Compiling actions by partial evaluation, revisited. Technical Report BRICS RS-98-13, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1998.

25. 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 LNCS, pages 182–197. Springer-Verlag, 1996. Extended version available as the technical report BRICS-RS-96-13.
26. Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, chapter 6, pages 243–320. Volume B of van Leeuwen [38], 1990.
27. Belmina Dzafic. Formalizing program transformations. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.
28. Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457. ACM Press, 1994.
29. Andrzej Filinski. From normalization-by-evaluation to type-directed partial evaluation. In Danvy and Dybjer [19].
30. Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, LNCS. Springer-Verlag, 1999. To appear. Extended version available as the technical report BRICS RS-99-17.
31. Mayer Goldberg. Gödelization in the λ-calculus. Technical Report BRICS RS-95-38, Computer Science Department, Aarhus University, Aarhus, Denmark, July 1995.
32. Mayer Goldberg. *Recursive Application Survival in the λ-Calculus*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, May 1996.
33. Bernd Grobauer. Types for proofs and programs. Progress report, BRICS PhD School, University of Aarhus. Available at `http://www.brics.dk/~grobauer`, June 1999.
34. William L. Harrison and Samuel N. Kamin. Modular compilers based on monads transformers. In Purush Iyer and Young il Choo, editors, *Proceedings of the IEEE International Conference on Computer Languages*, pages 122–131. IEEE Computer Society, 1998.
35. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997. Extended version available as the technical report BRICS RS-96-34.
36. Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In Jieh Hsiang and Atsushi Ohori, editors, *Advances in Computing Science - ASIAN'98*, number 1538 in LNCS, pages 188–205. Springer-Verlag, 1998.
37. Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1993.
38. Jan van Leeuwen, managing editor. *Handbook of Theoretical Computer Science*. The MIT Press, 1990.
39. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial evaluation for the lambda calculus. This volume.
40. 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.
41. Jesper Jørgensen. Similix: A self-applicable partial evaluator for Scheme. This volume.

42. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(3):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.
43. 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. ACM Press, 1994.
44. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
45. Eugenio Moggi. A categorical account of two-level languages. In Stephen Brookes and Michael Mislove, editors, *Proceedings of the 13th Annual Conference on Mathematical Foundations of Programming Semantics*, volume 6 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1997.
46. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
47. Jens Palsberg. Eta-redexes in partial evaluation. This volume.
48. Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.
49. Morten Rhiger. A study in higher-order programming languages. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1997.
50. Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 84-88, Technical report BRICS-NS-99-1, University of Aarhus, 1999.
51. Kristoffer Rose. Type-directed partial evaluation using type classes. In Danvy and Dybjer [19].
52. Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, California, February 1993. Technical report CSL-TR-93-563.
53. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
54. 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. ACM Press, 1997.
55. Guy L. Steele Jr. and Gerald J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.
56. Peter Thiemann. Aspects of the pgg system: Specialization for standard Scheme. This volume.
57. Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(32):365–387, 1993.
58. 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 LNCS, pages 165–191. Springer-Verlag, 1991.
59. Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300. ACM Press, 1998. Extended version available as the technical report BRICS RS-98-9.