# Partial Evaluation for the Lambda Calculus

Neil D. Jones, Carsten K. Gomard, Peter Sestoft*

DIKU, University of Copenhagen

This paper (essentially [12, Chapter 8]) describes partial evaluation for the lambda calculus, augmented with an explicit fixed-point operator. The techniques used here diverge from those used in [12, Chapters 4, 5] and [11] in that they are not based on specialization of *named* program points. The algorithm essentially leaves some operators (applications, lambdas, etc.) untouched and reduces others as standard evaluation would do it. This simple scheme is able to handle programs that rely heavily on higher-order facilities. The requirements on binding-time analysis are formulated via a type system and an efficient binding-time analysis via constraint solving is outlined. The partial evaluator is proven correct.

**History and recent developments** Self-applicable partial evaluation was first achieved in 1984 for a simple first-order functional language. This promising result was not immediately extendable to a higher-order language, the reason being that a specializer, given incomplete input data, in effect traces all possible program control flow paths and computes as many static values as possible. This seemed hard to do, since flow analysis of programs that manipulate functions as data values is non–trivial.

Breakthroughs occurred independently in 1989 by Bondorf (then at Dortmund) and by Gomard and Jones (Copenhagen). The latter, called *Lambdamix* and the subject of this paper, is conceptually simpler, theoretically motivated, and has been proven correct. Bondorf's work is more pragmatically oriented, led to the now widely distributed system Similix, and is the subject of [12, Chapter 10].

In common with the partial evaluators of the earlier book chapters, Lambdamix represents the concrete syntax of programs as constants (in fact Lisp S-expressions are used, though this is not essential). The natural question of whether partial evaluation is meaningful and possible in the classical *pure* lambda calculus without constants has recently been answered affirmatively.

Briefly: Mogensen devised a quite efficient self-interpreter for the pure lambda calculus, using 'higher-order abstract syntax' to encode lambda expressions as normal form lambda expressions. These are not difficult to interpret and even to specialize, although they are rather hard for humans to decipher. The ideas were later extended to give a self-applicable partial evaluator for the same language, using essentially the two level type system to be seen in this paper. The partial

* Chapter 8 from *Partial Evaluation and Automatic Program Generation*, Prentice-Hall International, 1993, ISBN 0-13-020249-5 (pbk). Reprinted with permission from Pentice-Hall International. All references to "book" in this article refers to this book.

evaluator was implemented, self-application gave the usual speedups, and it has since been proven correct by Wand using the technique of 'logical relations' [17, 26].

## 1 The lambda calculus and self-interpretation

The classical lambda calculus (extended with constants, conditionals, and a fix-point operator) is used here for simplicity and to allow a more complete treatment than would be possible for a larger and more practical language.

A lambda calculus program is an *expression*, `e`, together with an initial *environment*, $\rho$, which is a function from identifiers to values. The program takes its input through its free variables. The expression syntax given below differs from that of [12, Section 3.2] in that we have introduced an explicit fixed-point operator.

| | | |
|---|---|---|
| ⟨Lam⟩ ::= | ⟨Constant⟩ | Constants |
| \| | ⟨Var⟩ | Variables |
| \| | λ⟨Var⟩.⟨Lam⟩ | Abstraction |
| \| | ⟨Lam⟩ ⟨Lam⟩ | Application |
| \| | `fix` ⟨Lam⟩ | Fixed point operator |
| \| | `if` ⟨Lam⟩ `then` ⟨Lam⟩ `else` ⟨Lam⟩ | Conditional |
| \| | ⟨Op⟩ ⟨Lam⟩ ... ⟨Lam⟩ | Base application |

⟨Var⟩ ::= any identifier

Examples of relevant base functions include `=`, `*`, `cons`, etc. The fixed-point operator `fix` computes the least fixed point of its argument and is used to define recursive functions. For example, a program computing $x^n$ can be defined by

```
(fix λp.λn'.λx'.
   if (= n' 0)
   then 1
   else (* x' (p (- n' 1) x'))) n x
```

Note that `fix` λ`f.e` is equivalent to the Scheme constructs `(rec f e)` and `(letrec ((f e)) f)`. Why introduce an explicit fixed-point operator instead of using the Y-combinator written as a lambda expression [12, Section 3.2] to express recursion? This is because an explicit `fix` allows a simpler binding-time analysis.

As a first step towards partial evaluation we show a self-interpreter for the lambda calculus in Figure 1. Below we explain the notation used in Figure 1 and the remainder of the paper.

**Notation** *Const* is a 'flat' domain of constants large enough to include concrete syntax representations of lambda expressions (as input to and output from `mix`)

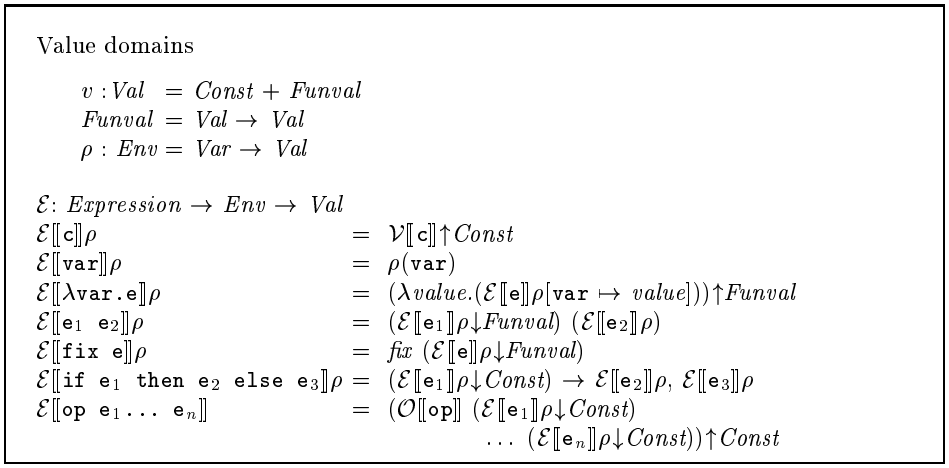


*Figure 1: Lambda calculus self-interpreter.*

and booleans for use in conditionals. As in earlier book chapters (and in our implementation) a suitable choice is the set of Lisp S-expressions. Further, we assume there are enough base functions to test equality, and to compose and decompose abstract syntax.

The separated sum of domains *Const* and *Funval* is written $Val = Const + Funval$. Given an element $b \in Const$, $v = b\uparrow Const \in Val$ is tagged as originating from *Const*. In SML or Miranda this would be written $v = Const\ b$. We have introduced the $\uparrow$ notation for symmetry with $v\downarrow Const$. This strips off the tag yielding an element in *Const* if $v$ is tagged as originating from *Const*. If $v$ has any other tag, then $v\downarrow Const$ produces an error.

We assume that all operations are strict in the error value but omit details. The domain $Funval = Val \to Val$ contains partial functions from *Val* to *Val*. Function $\mathcal{V}$ computes the value (in *Const*) of a constant expression (in *Exp*). Function $\mathcal{O}$ links names to base functions. The notation $\rho[\mathtt{var} \mapsto value]$ is, as in [12, Section 2.1], a shorthand for $\lambda$`x.if (x=var) then` *value* `else` $(\rho\ \mathtt{x})$ and is used to update environments. Expression $v_1 \to v_2, v_3$ has the value $v_2$ if $v_1$ equals *true* and value $v_3$ if $v_1$ equals *false*, else the error value.

Since we use lambda calculus both as an object level programming language and as a meta-language, we distinguish notationally between the two for clarity. Object level lambda expressions are written in `typewriter` style: `e e`, $\lambda$`var.e`, `fix e` etc., and the meta-language is in *italics*: *e e*, $\lambda$*var.e*, *fix e* etc.

**The self-interpreter** The structure of the self-interpreter is not much different from that of the lambda calculus interpreter written in ML and presented in [12, Section 3.3.1]. First-order structures have been replaced by functions in two places:

| | | |
|---|---|---|
| ⟨2Lam⟩ ::= | ⟨Constant⟩ | Constant |
| \| | ⟨Var⟩ | Variable |
| \| | `lift` ⟨2Lam⟩ | Lifting |
| \| | λ⟨Var⟩.⟨2Lam⟩ | Abstraction |
| \| | ⟨2Lam⟩ ⟨2Lam⟩ | Application |
| \| | `fix` ⟨2Lam⟩ | Fixed point |
| \| | `if` ⟨2Lam⟩ `then` ⟨2Lam⟩ `else` ⟨2Lam⟩ | Conditional |
| \| | ⟨Op⟩ ⟨2Lam⟩ ... ⟨2Lam⟩ | Base application |
| \| | $\underline{\lambda}$⟨Var⟩.⟨2Lam⟩ | Dyn. abstraction |
| \| | ⟨2Lam⟩ $\underline{@}$ ⟨2Lam⟩ | Dyn. application |
| \| | $\underline{\texttt{fix}}$ ⟨2Lam⟩ | Dyn. fixed point |
| \| | $\underline{\texttt{if}}$ ⟨2Lam⟩ $\underline{\texttt{then}}$ ⟨2Lam⟩ $\underline{\texttt{else}}$ ⟨2Lam⟩ | Dyn. conditional |
| \| | ⟨$\underline{\text{Op}}$⟩ ⟨2Lam⟩ ... ⟨2Lam⟩ | Dyn. base appl. |

*Figure 2: Two-level lambda calculus syntax.*

- The environment is implemented by a *function* from variables to values. Looking up the value of a variable `var` thus amounts to applying the environment $\rho$. This replaces the parallel lists of names and values seen in the interpreters from earlier book chapters.
- The value of an abstraction $\lambda$`var.e` is a *function* which, when applied to an argument value, evaluates `e` in an extended environment binding `var` to the value. The value of an application $\texttt{e}_1\ \texttt{e}_2$ is found by applying the value of $\texttt{e}_1$, which must be a function, to the value of $\texttt{e}_2$. This mechanism replaces the use of explicit closures.

It should be clear that, despite the extensive use of syntactic sugar, Figure 1 does define a self-interpreter, as the function $\mathcal{E}$ can easily be transformed into a lambda expression: `fix` $\lambda\mathcal{E}.\lambda$`e.`$\lambda\rho$`.if ...` .

## 2 Partial evaluation using a two-level lambda calculus

As in the previous book chapters we divide the task of partial evaluation into two phases: *first* we apply binding-time analysis, which yields a suitably annotated program, *then* reduce the static parts, blindly obeying the annotations. An annotated program is a two-level lambda expression. The two-level lambda calculus has two different versions of each of the following constructions: application, abstraction, conditionals, fixed points, and base function applications. One version is *dynamic*, the other is *static*. The static operators are those of the standard lambda calculus: `if`, `fix`, $\lambda$, etc. and the dynamic operators are underlined: $\underline{\texttt{if}}$, $\underline{\texttt{fix}}$, $\underline{\lambda}$, $\underline{@}$. ($\underline{@}$ denotes a dynamic application.) The abstract syntax of two-level expressions is given in Figure 2.

Intuitively, all static operators $\lambda$, `@`, ... are treated by the partial evaluator as they were treated by the self-interpreter. The result of evaluating a dynamic operator ($\underline{\lambda}$, $\underline{@}$, ...) is to produce a piece of *code* for execution at run-time —

a constant which is the concrete syntax representation of a residual one-level lambda expression, perhaps with free variables.

The `lift` operator also builds code — a constant expression with the same value as `lift`'s argument. The operator `lift` is applied to static subexpressions of a dynamic expression.

A two-level *program* is a two-level expression `te` together with an initial environment $\rho_s$ which maps the free variables of `te` to constants, functions, or code pieces. We shall assume that free dynamic variables are mapped to distinct, new variable names. The $\mathcal{T}$-rules (Figure 3) then ensure that these new variables become the free variables of the residual program.

Variables bound by $\underline{\lambda}$, will also (eventually) generate fresh variable names in the residual program, whereas variables bound by $\lambda$ can be bound at specialization time to all kinds of values: constants, functions, or code pieces.

The $\mathcal{T}$-rule for a dynamic application is

$$\mathcal{T}[\![\texttt{te}_1\ \underline{@}\ \texttt{te}_2]\!]\rho = \mathit{build}\text{-}@(\mathcal{T}[\![\texttt{te}_1]\!]\rho{\downarrow}\mathit{Code}, \mathcal{T}[\![\texttt{te}_2]\!]\rho{\downarrow}\mathit{Code}){\uparrow}\mathit{Code}$$

The recursive calls $\mathcal{T}[\![\texttt{te}_1]\!]\rho$ and $\mathcal{T}[\![\texttt{te}_2]\!]\rho$ produce the code for residual operator and operand expressions, and the function *build-@* 'glues' them together to form an application to appear in the residual program (concretely, an expression of the form `(te`$_1$`' te`$_2$`')`). All the *build*-functions are strict.

The projections ($\downarrow\mathit{Code}$) check that both operator and operand reduce to code pieces, to avoid applying specialization time operations (e.g. boolean tests) to residual program pieces. Finally, the newly composed expression is tagged ($\uparrow\mathit{Code}$) as being a piece of code.

The $\mathcal{T}$-rule for variables is

$$\mathcal{T}[\![\texttt{var}]\!]\rho = \rho(\texttt{var})$$

The environment $\rho$ is expected to hold the values of all variables regardless of whether they are predefined constants, functions, or code pieces. The environment is updated in the usual way in the rule for static $\lambda$, and in the rule for $\underline{\lambda}$, the formal parameter is bound to an as yet unused variable name, which we assume available whenever needed:

$$\begin{aligned}\mathcal{T}[\![\underline{\lambda}\texttt{var.te}]\!]\rho = {} & \text{let } \texttt{nvar} = \mathit{newname}(\texttt{var}) \\ & \text{in } \mathit{build}\text{-}\lambda(\texttt{nvar}, \mathcal{T}[\![\texttt{te}]\!]\rho[\texttt{var} \mapsto \texttt{nvar}]{\downarrow}\mathit{Code}){\uparrow}\mathit{Code}\end{aligned}$$

Each occurrence of `var` in `te` will then be looked up in $\rho[\texttt{var} \mapsto \texttt{nvar}]$, causing `var` to be replaced by the fresh variable `nvar`. Since $\underline{\lambda}$`var.te` might be duplicated, and thus become the 'father' of many $\lambda$-abstractions in the residual program, this renaming is necessary to avoid name confusion in residual programs. Any free dynamic variables must be bound to their new names in the initial static environment $\rho_s$. The generation of new variable names relies on a side effect on a global state (a name counter). In principle this could be avoided by adding an extra parameter to $\mathcal{T}$, but for the sake of notational simplicity we have used a less formal solution.

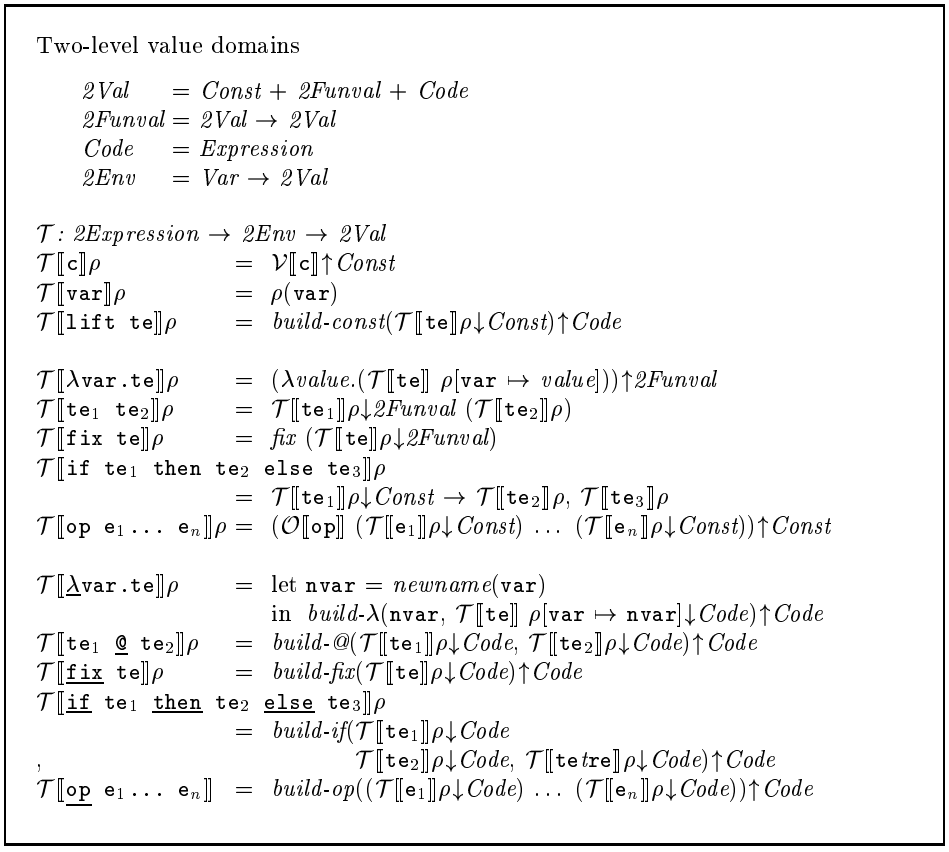


*Figure 3: Two-level lambda calculus interpreter.*

The valuation functions for two-level lambda calculus programs are given in Figure 3. The rules contain explicit tagging and untagging with ↑ and ↓; Section 3 will discuss sufficient criteria for avoiding the need to perform them.

*Example 1.* Consider again the power program:

```
(fix λp.λn'.λx'.
    if (= n' 0)
    then 1
    else (* x' (p (- n' 1) x'))) n x
```

and suppose that `n` is known and `x` is not. A suitably annotated power program, `power-ann`, would be:

```
(fix λp.λn'.λx'.
    if (= n' 0)
    then (lift 1)
    else (* x' (p (- n' 1) x'))) n x
```

Partial evaluation of `power` (that is, two-level evaluation of `power-ann`) in environment $\rho_s = [\texttt{n} \mapsto \mathit{2}\uparrow \mathit{Const}, \texttt{x} \mapsto \texttt{xnew}\uparrow \mathit{Code}]$ yields:

$$\begin{aligned}
&\mathcal{T}[\![\texttt{power-ann}]\!]\rho_s \\
&= \mathcal{T}[\![\texttt{(fix } \lambda\texttt{p.}\lambda\texttt{n'.}\lambda\texttt{x'.if ...) n x}]\!]\rho_s \\
&= \texttt{* xnew (* xnew 1)}
\end{aligned}$$

In the power example it is quite clear that for all $\mathit{d2}$, $\rho = [\texttt{n} \mapsto \mathit{2}, \texttt{x} \mapsto \mathit{d2}]$, $\rho_s = [\texttt{n} \mapsto \mathit{2}, \texttt{x} \mapsto \texttt{xnew}]$, and $\rho_d = [\texttt{xnew} \mapsto \mathit{d2}]$ (omitting injections for brevity) it holds that

$$\mathcal{E}[\![\texttt{power}]\!]\rho = \mathcal{E}[\![\mathcal{T}[\![\texttt{power-ann}]\!]\rho_s]\!]\rho_d$$

This is the mix equation (see [12, Section 4.2.2]) for the lambda calculus. [12, Section 8] contains a general correctness theorem for two-level evaluation.

## 3 Congruence and consistency of annotations

The semantic rules of Figure 3 check explicitly that the values of subexpressions are in the appropriate summands of the value domain, in the same way that a type-checking interpreter for a dynamically typed language would. Type-checking on the fly is clearly necessary to prevent partial evaluation from committing type errors itself on a poorly annotated program.

Doing type checks on the fly is not very satisfactory for practical reasons. Mix is supposed to be a general and automatic program generation tool, and one wishes for obvious reasons for it to be impossible for an automatically generated compiler to go down with an error message.

Note that it is in principle possible — but unacceptably inefficient in practice — to avoid partial evaluation-time errors by annotating as dynamic all operators in the subject program. This would place all values in the code summand so all type checks would succeed; but the residual program would always be isomorphic to the source program, so it would not be optimized at all.

The aim of this section is to develop a more efficient strategy, ensuring before specialization starts that the partial evaluator *cannot* commit a type error. This strategy was seen in [12, Chapters 4, 5] and [11]. The main difference now is that in a higher-order language it is less obvious *what* congruence is and *how* to ensure it.

### 3.1 Well-annotated expressions

A simple and traditional way to preclude type check errors is to devise a type system. In typed functional languages, a type inference algorithm such as algorithm W checks that a program is well-typed prior to program execution [15]. If it is, then no run-time summand tags or checks are needed. Type checking is

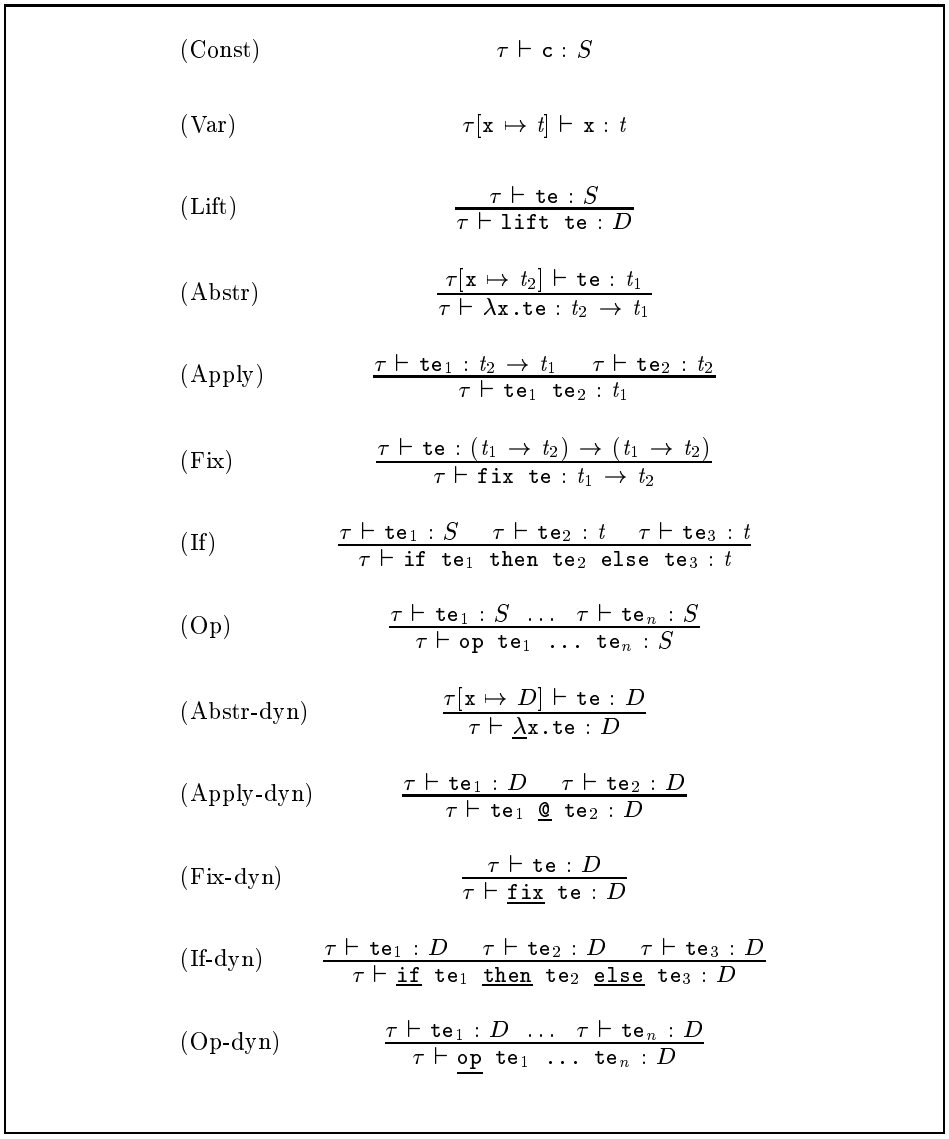


*Figure 4: Type rules checking well-annotatedness.*

quite well understood and can be used to get a nice formulation of the problem to be solved by binding-time analysis [4, 22].

We saw in [12, Section 5.7] and [11] that type rules can be used to check well-annotatedness, and we now apply similar reasoning to the lambda calculus.

**Definition 1.** *The* two-level types *$t$ are as follows, where $\alpha$ ranges over type variables:*

$t ::= \alpha \mid S \mid D \mid t \to t$

*A* type environment $\tau$ *is a mapping from program variables to types.*

**Definition 2.** *Let* $\tau$ *be a type environment mapping the free variables of a two-level expression* `te` *to their types. Then* `te` *is* well-annotated *if* $\tau \vdash$ `te` $: t$ *can be deduced from the inference rules in Figure 4 for some type* $t$.

For example, the two-level expression `power-ann` of Example 1 is well-annotated in type environment $\tau = [\mathtt{n} \mapsto S, \mathtt{x} \mapsto D]$. The whole expression has type $D$, and the part `(fix p ...)` has type $S \to D \to D$.

Our lambda calculus is basically untyped, but the well-annotatedness ensures that all program parts evaluated at partial evaluation time will be well-typed, thus ensuring specialization against type errors. The well-annotatedness criterion is, however, completely permissive concerning the run-time part of a two-level expression. Thus a lambda expression without static operators is trivially well-typed — *at partial evaluation time.*

Two-level expressions of type $S$ evaluate (completely) to *first-order* constants, and expressions of type $t_1 \to t_2$ evaluate to a function applicable *only at partial evaluation time.* The value by $\mathcal{T}$ of a two-level expression `te` of type $D$ is a one-level expression `e`. For partial evaluation we are only interested in fully annotated programs `p-ann` that have type $D$. In that case, $\mathcal{T}[\![\mathtt{p\text{-}ann}]\!]\rho_s$ (if defined) will be a piece of code, namely the residual program.

In our context, the result about error freedom of well-typed programs can be formulated as follows. Proof is omitted since the result is well-known.

**Definition 3.** *Let* $t$ *be a two-level type and* $v$ *be a two-level value. We say that* $t$ suits $v$ *iff one of the following holds:*

1. $t = S$ *and* $v = ct{\uparrow}Const$ *for some* $ct \in Const$.
2. $t = D$ *and* $v = cd{\uparrow}Code$ *for some* $cd \in Code$.
3. *(a)* $t = t_1 \to t_2$, $v = f{\uparrow}2Funval$ *for some* $f \in 2Funval$, *and*
   *(b)* $\forall\ v \in 2Val$: $t_1$ *suits* $v$ *implies* $t_2$ *suits* $f(v)$.

*A type environment* $\tau$ *suits an environment* $\rho$ *if for all variables* `x` *bound by* $\rho$, $\tau(\mathtt{x})$ *suits* $\rho(\mathtt{x})$.

The following is a non-standard application of a standard result [15].

**Proposition 1.** *('Well-annotated programs do not go wrong') If* $\tau \vdash$ `te` $: t$, *and* $\tau$ *suits* $\rho_s$, *then* $\mathcal{T}[\![\mathtt{te}]\!]\rho_s$ *does not yield a projection error.*

Of course $\mathcal{T}$ can 'go wrong' in other ways than by committing type errors. Reduction might proceed infinitely (so $\mathcal{T}[\![\mathtt{p\text{-}ann}]\!]\rho_s$ is not defined) or residual code might be duplicated. We shall not discuss these problems here.

## 4 Binding-time analysis

**Definition 4.** *The* annotation-forgetting function $\phi$*: 2Exp* $\rightarrow$*Exp, when applied to a two-level expression* `te`*, returns a one-level expression* `e` *which differs from* `te` *only in that all annotations (underlines) and* `lift` *operators are removed.*

**Definition 5.** *Given two-level expressions,* `te` *and* $\texttt{te}_1$*, define* $\texttt{te} \sqsubseteq \texttt{te}_1$ *by*

1. $\phi(\texttt{te}) = \phi(\texttt{te}_1)$
2. *All operators underlined in* `te` *are also underlined in* $\texttt{te}_1$

Thus $\sqsubseteq$ is a preorder on the set of two-level expressions. Given a $\lambda$-expression `e`, let a *binding-time assumption* for `e` be a type environment $\tau$ mapping each free variable of `e` to either $S$ or $D$.

**Definition 6.** *Given an expression* `e` *and a binding-time assumption* $\tau$*, a* completion *of* `e` *for* $\tau$ *is a two-level expression* $\texttt{te}_1$ *with* $\phi(\texttt{te}_1) = \texttt{e}$ *and* $\tau \vdash \texttt{te}_1 : t$ *for some type* $t$*. A* minimal completion *is an expression* $\texttt{te}_2$ *which is a completion of* `te` *fulfilling* $\texttt{te}_2 \sqsubseteq \texttt{te}_1$ *for all completions* $\texttt{te}_1$ *of* `e`*.*

Minimal completions are in general not unique. Assume $\tau = [\texttt{y} \mapsto D]$, and `e` = `(`$\lambda$`x.x+y) 4`. There are two minimal completions, $\texttt{te}_1$ = `(`$\lambda$`x.x`$\underline{\texttt{+}}$`y) (lift 4)` and $\texttt{te}_2$ = `(`$\lambda$`x.(lift x)`$\underline{\texttt{+}}$`y) 4` which yield identical residual programs when partially evaluated. The definition of $\sqsubseteq$ does not distinguish between (minimal) completions which differ only in the choice of `lift`-points. Residual programs are identical for completions $\texttt{te}_1$ and $\texttt{te}_2$ if $\texttt{te}_1 \sqsubseteq \texttt{te}_2$ and $\texttt{te}_2 \sqsubseteq \texttt{te}_1$, and the impact of different choices on efficiency of the partial evaluation process itself is of little importance.

The requirement that $\tau$ be a binding-time assumption implies that all free variables are first-order. This ensures the existence of a completion. Note that a $\lambda$-bound variable `x` can get any type in completions, in particular a functional type. Possible conflicts can be resolved by annotating the abstraction(s) and application(s) that force `x` to have a functional type.

The task of binding-time analysis in the $\lambda$-calculus is briefly stated: given an expression `e` and a binding-time assumption $\tau$ find a minimal completion of `e` for $\tau$. [12, Section 7], shows by example that this can be done by type inference, and [12, Section 8] shows how to do it in a much more efficient way.

**Proposition 2.** *Given an expression* `e` *and a binding-time assumption* $\tau$ *there exist(s) minimal completion(s) of* `e` *for* $\tau$*.*

*Proof. Follows from the properties of the constraint-based binding-time analysis algorithm in [12, Section 8.7].*

## 5 Simplicity versus power in Lambdamix

A value of type $t \neq D$ can only be bound to a variable by applying a function of type $t \to t'$. The partial evaluation time result of such a statically performed application is found by evaluating the function body, no matter what the type of the argument or the result is. This corresponds closely to unfolding on the fly of *all* static function calls (see [12, Section 5.5] and [11]).

Lambdamix does not perform specialization of *named* program points. Rather, generation of multiple variants of a source expression can be accomplished as an implicit result of unfolding a `fix` operator, since static variables may be bound to different values in the different unfoldings.

The only way to prevent a function call from being unfolded is to annotate the function as dynamic: $\underline{\lambda}$. All applications of that function must accordingly be annotated as dynamic. Dynamic functions $\underline{\lambda}\ldots$ can only have dynamic arguments (Figure 4). Note that this restriction does not exist in [12, Chapter 5] where named functions are specialized. As an example, consider the append function, `app`, written as a lambda expression:

```
(fix λapp.λxs.λys.
      if   (null? xs)
      then ys
      else (cons (car xs) (app (cdr xs) ys))) xs0 ys0
```

Partial evaluation with `xs0` = `'(a b)` and dynamic `ys0` yields `(cons 'a (cons 'b ys0))`, a result similar to that produced by the Scheme0 specializer from [12, Chapter 5] (with any reasonable unfolding strategy). Lambdamix handles this example well because the recursive calls to `app` should be unfolded to produce the optimal residual program. Unfolding the calls allows Lambdamix to exploit the static argument, `(cdr xs)`.

Now assume that `xs0` is dynamic and that `ys0` is static with value `'(c d)`. When applied to a corresponding problem, the techniques from [12, Chapter 5] would produce the residual Scheme0 program

```
(define (app-cd xs)
  (if (null? xs)
      '(c d)
      (cons (car xs) (app-cd (cdr xs)))))
```

where the recursive call to `app-cd` is not unfolded. Now consider this problem in the Lambdamix framework. With dynamic `ys0`, a minimal completion of the append program is:

```
(fix λapp.λxs.λys.
      if   (null? xs)
      then (lift ys)
      else (cons (car xs) (app (cdr xs) ys))) xs0 ys0
```

Note that even though `xs0` and `xs` are dynamic the function $\lambda$`xs.`$\lambda$`ys....` is still static in the minimal completion. Lambdamix will loop infinitely by unfolding

the recursive applications of `app`. To avoid infinite unfolding, the recursive application `(app (cdr xs) ys)` must be annotated as dynamic, which forces the whole expression `fix` $\lambda$`app....` to be annotated as dynamic. This means that no computation can be done by terminating partial evaluation.

In this particular example, specialization of the named function `app` with respect to first-order data `ys0 = '(c d)` could be obtained by simple methods but to get a general solution to this class of problems we must also consider specialization with respect to higher-order values, i.e., functions. We shall return to this in [12, Chapter 10].

### 5.1 Optimality of Lambdamix

Lambdamix has been tested on several interpreters derived from denotational language definitions [5]. Such interpreters are compositional in the program argument, which means that recursive calls in the interpreter can be safely unfolded when the interpreter is specialized with respect to a concrete source program. Lambdamix often performs well on interpreters fulfilling compositionality, and is often able to specialize away interpretive overhead such as syntactic dispatch, environment lookups, etc.

A compelling example: when the self-interpreter from Figure 1 (after removing all tagging and untagging operations) is specialized with respect to a lambda expression `e`, the residual program is an expression `e'` which is *identical* to `e` modulo renaming of variables and insignificant coding of base function applications. Thus Lambdamix is nearly optimal as defined in [12, Chapter 6]. (A small difference: the call: `(+ e1 e2)` is transformed into `(apply '+ e1 e2)`, etc. The problem can be fully eliminated by treating base functions as free variables, bound in the initial environment [5] or by a simple post processing like in [12, Chapter 10].)

## 6 Binding-time analysis by type inference

An intuitively natural approach to binding-time analysis for the lambda calculus uses a variant of the classical Algorithm W for polymorphic type inference [5, 16, 22]. The guiding principle is that the static parts of an annotated program must be well-typed. This naturally leads to an algorithm that tries to type a given program in its given type environment.

If this succeeds, all is well and specialization can proceed. If type inference fails, the application of a user-defined or base function that led to the type conflict is made dynamic (i.e. an underline is added), and the process is repeated. Eventually, enough underlines will be added to make the whole well-typed and so suitable for specialization.

We only give an example for brevity, since the next section contains a much more efficient algorithm. Recall the power program of Example 1:

```
(fix λp.λn'.λx'.
```

```
if (= n' 0) then 1
else (* x' (p (- n' 1) x'))) n x
```

with initial type environment $\tau = [\mathtt{n} \mapsto S, \mathtt{x} \mapsto D]$. At the `if`, Algorithm W works with the type environment:

$$[\mathtt{p} \mapsto (S \to D \to \alpha), \mathtt{n}' \mapsto S, \mathtt{x}' \mapsto D, \mathtt{n} \mapsto S, \mathtt{x} \mapsto D]$$

where $\alpha$ is an as yet unbound type variable. Thus expression `(p (- n' 1) x')` has type $\alpha$, which is no problem. This leads, however, to a type conflict in expression `(* x' (p (- n' 1) x'))` since static operator `*` has type $S \times S \to S$, in conflict with `x'`, which has type $D$.

The problem is resolvable by changing `*` to $\underline{*}$, with type $D \times D \to D$. This forces $\alpha = D$ so the `else` expression has type $D$. The single remaining conflict, that `1` has type $S \neq D$, is easily resolved by changing the `1` to `lift 1`, or by underlining it. The first solution leads to the annotation of Example 1.

## 7 BTA by solving constraints

- OMITTED –

## 8 Correctness of Lambdamix

- OMITTED –

## 9 Subsequent Work [thanks to John Hatcliff]

Due to its simplicity, aspects of lambda-mix have been widely used for exploring foundation issues such as correctness of binding-time analysis and specialization, binding-time improvements, and specialization of programs with computational effects.

### 9.1 Basic correctness issues

As noted in the introduction, Mogensen [17] presented a self-applicable offline partial evaluator for the pure lambda-calculus. His specification of binding-time analysis using a two-level language is essentially the same as the one used with lambda-mix. However, the specializer is more compact (and thus somewhat harder to decipher) since program terms are represented by higher-order abstract syntax. Wand [26] subsequently carried out a thorough investigation of the correctness issues for the partial evaluator and binding-time analysis of Mogensen. Mogensen has also defined an online self-applicable partial evaluator for the pure lambda calculus [18].

Palsberg [24] gives an alternative view of correctness of binding-time analysis for lambda-terms. In his presentation, correctness is not based on a single definition of specialization. Instead he gives conditions that an arbitrary specializer and binding-time analysis must satisfy if they are to avoid producing "confused redexes", i.e., situations where the specializer can "go wrong" (see Section 3).

Hatcliff [6] defines an alternate specification of the lambda-mix specializer using operational semantics and shows how this allows correctness to be mechanically checked using the Elf implementation of the logical framework LF [25]

### 9.2 Two-level languages

Two-level languages, as used in Flemming Nielson's work in abstract interpretation, were adapted by Gomard and Jones to define well-formedness of lambda-mix's annotated programs and to direct the actions of the specializer. The lambda-mix development was independent of Nielson and Nielson's [22].

Oddly, paper [22] has no references to Copenhagen's work in partial evaluation, which includes the invention of the term "binding-time analysis" and its first use in practice. The framework used in [22] has somewhat different assumptions about static-dynamic divisions and typing, making it unsuitable for lambda-mix in particular, or for self-applicable partial evaluation in general.

Since [22], the Nielsons have refined the theory of two-level languages [23], and extended it in several directions.

More recently, Moggi [21] has given a category-theoretic semantics for variants of the two-level language used in lambda-mix and in [22].

### 9.3 Binding-time analysis algorithms

The original binding-time analysis algorithm for lambda-mix was derived from the well-known type inference algorithm called "algorithm W" [16]. Henglein subsequently developed an constraint-solving algorithm that runs in almost-linear time [9]. Many partial evaluators (including the Similix system – See Chapter 12 of [12]) implement binding-time analysis using adaptations of Henglein's algorithm.

### 9.4 Binding-time improvements and continuation-based specialization

Many studies of binding-time improvements have used lambda-mix style specialization as a foundation. Danvy, Malmkjær, and Palsberg investigate dataflow binding-time improvements based on eta-redexes [3] (see also Palsberg's article in this volume). Lawall and Danvy show how control-flow binding-time improvements can be incorporated using control operators shift and reset [13]. Bondorf and Dussart present a hand-written continuation-based cogen for the lambda-calculus [1].

### 9.5 Dealing with computational effects

Naively incorporating language features that cause side-effects (such as I/O and update of mutable variables) into lambda-mix leads to unsound specialization. Hatcliff and Danvy [8, 7] give a formal presentation of offline partial evaluation for both the call-by-name and call-by-value lambda-calculus using Moggi's monadic metalanguage [20]. This provides a foundation for specializing programs with computational effects and provides a category-theoretic explanation of previous work on control-based binding-time improvements.

Lawall and Thiemann [14] give an alternate presentation for call-by-value lambda-terms using Moggi's computational lambda-calculus [19]. Their approach has the effect of extending lambda-mix with a **let**-construct an introducing extra reduction rules (based on the laws of monads) that ensure sound specialization in the presence of any computational effect that can be described using a monad.

### 9.6 Other approaches to specializing the lambda-terms

Danvy has described how the same effect as lambda-mix style specialization can be obtained using "type-directed partial evaluation" [2] (also see the article by Danvy in this volume). Type-directed partial evaluation is in essence a normalization procedure that works by systematically eta-expanding terms as directed by their type.

Also emphasizing types, Hughes [10] has shown how specialization of functional programs can be carried out using type inference. His approach was motivated by the desire to avoid producing the unnecessary type tag manipulation in residual programs that often appears when specializing an interpreter written in a strongly typed language. Lambda-mix is untyped and so does not suffer from this problem, but like any untyped language involves substantial run-time type tag checking.

## 10 Exercises

Some of the exercises involve finding a minimal completion. The formal algorithm to do this is targeted for an efficient implementation and is not suited to be executed by hand (for other than very small examples). So if not otherwise stated just use good sense for finding minimal completions.

*Exercise 1.* Find a minimal completion for the lambda expression listed below given the binding-time assumptions $\tau = [\texttt{m0} \mapsto S, \texttt{n0} \mapsto D]$. Specialize the program with respect to `m0 = 42`.

```
(λm.λn.+ m n) m0 n0
```

*Exercise 2.* Find a minimal completion for the lambda expression listed below given the binding-time assumptions $\tau = [\texttt{x0} \mapsto S, \texttt{xs0} \mapsto S, \texttt{vs0} \mapsto D]$. Specialize the program with respect to `x0 = c` and `xs0 = (a b c d)`.

```
(fix
λlookup.λx.λxs.λvs.
if (null? xs)
  then 'error
  else if (equal? x (car xs))
       then (car vs)
       else (lookup x (cdr xs) (cdr vs))) x0 xs0 vs0
```

*Exercise 3.* In the previous book chapters, a self-interpreter `sint` has been defined by

$$[\![\texttt{sint}]\!]_{\mathrm{L}}\ \texttt{p d} = [\![\texttt{p}]\!]_{\mathrm{L}}\ \texttt{d}$$

Define `sint`, basing it on $\mathcal{E}$ for instance, such that this equation holds for the lambda calculus.

*Exercise 4.*

1. Write a self-interpreter `sint` for the lambda calculus by transforming the function $\mathcal{E}$ into a lambda expression `fix` $\lambda\mathcal{E}.\lambda$`e`$.\lambda\rho$`.if ... e' ρ'` with free variables `e'` and $\rho$`'`.
2. Find a minimal completion for `sint` given binding-time assumptions $\tau = [$`env'` $\mapsto S$, $\rho$`'` $\mapsto D]$.
3. Find a minimal completion for `sint` given binding-time assumptions $\tau = [$`env'` $\mapsto S$, $\rho$`'` $\mapsto (S \to D)]$.
4. Specialize `sint` with respect to the power program in Section 1. The free variables `e'` and $\rho$`'` of `sint` shall have the following static values: `e' = ((fix` $\lambda$`p.`$\lambda$`n'.`$\lambda$`x'....) n x)` and $\rho$`'` $= [$`n` $\mapsto$ `n`, `x` $\mapsto$ `x`$]$.

*Exercise 5.* Implement the partial evaluator from Figure 3 in a programming language of your own choice.

*Exercise 6.* * Implement the partial evaluator from Figure 3 in the lambda calculus. It might be a good idea to implement the self-interpreter first and then extend it to handle the two-level expressions. Use the partial evaluator to specialize `sint` with respect to various lambda expressions. Is the partial evaluator optimal? Try self-application of the partial evaluator.

*Exercise 7.*

1. At the end of Section 1 is listed how the lambda calculus interpreter in [12, Section 3.1] has been revised to obtain that in Figure 1. How do these revisions affect the residual programs produced by partial evaluation of these interpreters?
2. What further revisions would be necessary to achieve optimality?

*Exercise 8.* Prove that residual programs are identical for completions $\texttt{te}_1$ and $\texttt{te}_2$ if $\texttt{te}_1 \sqsubseteq \texttt{te}_2$ and $\texttt{te}_2 \sqsubseteq \texttt{te}_1$. Discuss the impact of different choices on efficiency of the partial evaluation process itself.

## References


1. Anders Bondorf and Dirk Dussart. Improving cps-based partial evaluation: Writing cogen by hand. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, volume 94/9 of *Technical Report*, pages 1–9. University of Melbourne, Australia, 1994.
2. Olivier Danvy. Pragmatics of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 73–94. Springer-Verlag, 1996.
3. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995.
4. C.K. Gomard. Partial type inference for untyped functional programs. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 282–287. ACM, 1990.
5. C.K. Gomard and N.D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.
6. John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Proceedings of the Seventh International Symposium on Programming Languages, Implementations, Logics and Programs*, number 982 in Lecture Notes in Computer Science, pages 279–298, Utrecht, The Netherlands, September 1995.
7. John Hatcliff. Foundations of partial evaluation of functional programs with computational effects. *ACM Computing Surveys*, 1998. (in press).
8. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7:507–541, 1997. Special issue devoted to selected papers from the *Workshop on Logic, Domains, and Programming Languages*. Darmstadt, Germany. May, 1995.
9. Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *FPCA*, pages 448–472. 5th ACM Conference, Cambridge, MA, USA, Berlin: Springer-Verlag, August 1991. Lecture Notes in Computer Science, Vol. 523.
10. John Hughes. Type specialisation for the $\lambda$-calculus; or a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–215. Springer-Verlag, 1996.
11. J. Hatcliff. An introduction to partial evaluation using a simple flowchart language. *This volume*, 1998.
12. N.D. Jones, C. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
13. Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. *LFP*, pages 227–238, 1994.
14. Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proceedings of Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, September 1997. (to appear).
15. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
16. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. Cambridge, MA: MIT Press, 1990.

17. T. Mogensen. Self-applicable partial evaluation for pure lambda calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992. (Technical Report YALEU/DCS/RR-909, Yale University)*, pages 116–121, 1992.
18. T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In William L. Scherlis, editor, *Proceedings of PEPM '95*, pages 39–44. ACM, ACM Press, 1995.
19. Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.
20. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
21. Eugenio Moggi. A categorical account of two-level languages. In *Mathematical Foundations of Programming Semantics*, Technical Report, pages 199–212. Electronic Notes in Theoretical Computer Science, 1997.
22. H.R. Nielson and F. Nielson. Automatic binding time analysis for a typed $\lambda$-calculus. *Science of Computer Programming*, 10:139–176, 1988.
23. H.R. Nielson and F. Nielson. Two-Level Functional Languages. *Cambridge University Press*, 1992. Cambridge Tracts in Theoretical Computer Science **vol. 34**.
24. Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.
25. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
26. M. Wand. Specifying the correctness of binding-time analysis. In *Twentieth ACM Symposium on Principles on Programming Languages, Charleston, South Carolina*, pages 137–143. New York: ACM, 1993.