A Hybrid Approach to Online and Offline Partial Evaluation *

Eijiro Sumii (sumii@yl.is.s.u-tokyo.ac.jp)
Department of Information Science, Graduate School of Science, University of Tokyo. Hongo 7–3–1, Bunkyo-ku, Tokyo 113–0033, Japan.

Naoki Kobayashi (kobayasi@kb.cs.titech.ac.jp)
Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology. Oookayama 2–12–1, Meguro-ku, Tokyo 152–8552, Japan.

Abstract. This article presents a hybrid method of partial evaluation (PE), which is exactly as precise as naive online PE and nearly as efficient as state-of-the-art offline PE, for a statically typed call-by-value functional language.

PE is a program transformation that specializes a program with respect to a subset of its input by reducing the program and leaving a residual program. Online PE makes the reduction/residualization decision during specialization, while offline PE makes it before specialization by using a static analysis called binding-time analysis. Compared to offline PE, online PE is more *precise* in the sense that it finds more redexes, but less *efficient* in the sense that it takes more time.

To solve this dilemma, we begin with a naive online partial evaluator, and make it efficient without sacrificing its precision. To this end, we (1) use state (instead of continuations) for let-insertion, (2) take a so-called cogen approach (instead of self-application), and (3) remove unnecessary let-insertion, unnecessary tags, and unnecessary values/expressions by using a type-based representation analysis, which subsumes various monovariant binding-time analyses.

We implemented and compared our method and existing methods—both online and offline—in a subset of Standard ML. Experiments showed that (1) our method produces as fast residual programs as online PE and (2) it does so at least twice as fast as other methods (including a cogen approach to offline PE with a polyvariant binding-time analysis) that produce comparable residual programs.

Keywords: online partial evaluation, state-based let-insertion, cogen approach, binding-time analysis

1. Introduction

Partial evaluation (PE) is a program transformation that specializes a program with respect to a subset of its input, by reducing the pre-computable (*static*) portions and residualizing the other (*dynamic*) portions (Consel and Danvy, 1993; Jones et al., 1993). For example, given a source program $p(s, d) = (1 + s) + d$ and a static input $s = 2$,

* Extended and revised version of an article that appeared in *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.*

PE produces a residual program $p_2(d) = 3 + d$ by reducing the first addition $1 + 2$ and residualizing the second addition $3 + d$. For another example, by specializing a generic interpreter with respect to a specific program, PE generates compiled (and usually faster) code from an interpreted (and usually slower) program (Futamura, 1971).

According to when the decision between the reduction and the residualization is made, there exist two approaches to PE. One is *offline* PE, which makes the decision *before* specialization without the value of a static input, in a preprocessing phase called *binding-time analysis* (BTA). The other is *online* PE, which makes the decision *during* specialization with the value of a static input. Each of online PE and offline PE has disadvantages:

- Online PE is more complex than offline PE, where the examination of binding times is factored out as a preprocessing phase. For example, termination guarantee, speedup prediction, and in particular *self-application* are more difficult in online PE than in offline PE.
- Online PE is less efficient because it examines the binding time of values many times during specialization, while offline PE does so just once in the BTA. This disadvantage of online PE is even more significant when specialization is repeated with the same binding times (but with different values). In addition, in order to prevent code duplication, online PE has to residualize *all* structural data (such as functions and pairs) by inserting a let-binding (called *let-insertion*), while offline PE let-inserts dynamic expressions only.
- Offline PE requires more modifications (*binding-time improvements*) of a source program for satisfactory specialization, because it is conservative in the sense that it treats everything as dynamic that the BTA cannot infer as static.

In order to address these problems, we improve *online* PE by incorporating the techniques in offline PE as well as by inventing new techniques. Our contributions in this article are as follows.

State-based let-insertion to alleviate the overhead of let-insertion in naive online PE. Although this technique is useful for let-insertion in various forms of PE (for instance, see Section 2.7 of Danvy, 1999), it is especially useful in online PE where let-insertion happens particularly frequently.

Cogen approach to online PE to remove the interpretive overhead in naive online PE (and obviate self-application). Unlike a cogen

approach to offline PE, this technique brings little speedup by itself, but causes significant speedup in combination with the type-based representation analysis below.

Type-based representation analysis for removing unnecessary computations in naive online PE. This technique can be seen as a kind of "partial" BTA that uses binding-time information to optimize the specialization process. In addition, unlike an ordinary BTA, the analysis also helps to remove unnecessary let-insertion by detecting expressions that are never duplicated. Thanks to this, specialization using the analysis is more than twice as fast as the same specialization using existing BTA's.

The rest of this article is structured as follows. Section 2 discusses related work. Section 3 explains naive online PE and its problems; those who are familiar with online PE may skim through this section. Section 4, Section 5, and Section 6 present state-based let-insertion, the cogen approach to online PE, and the type-based representation analysis, respectively. Section 7 enumerates extensions and limitations of our method. Section 8 gives experimental results comparing our method with others, including naive online PE and Thiemann's cogen approach to offline PE (Thiemann, 1999). Section 9 concludes with future work.

2. Related Work

2.1. Self-Application and The Cogen Approach

In the history of PE, self-application has been a standard approach to efficient specialization. Much work has been invested in developing techniques for successful self-application, both in online PE (Ruf, 1993; Sperber, 1996) and in offline PE (Jones et al., 1990; Jones et al., 1993). However, self-application is difficult in essence because the more complex the source programs become, the more complex the partial evaluator becomes. Even in Similix (Bondorf and Danvy, 1991), which has been a leading offline partial evaluator for a subset of Scheme, self-application often brings little speedup and sometimes causes a *slow-down* of specialization (Thiemann, 1999). In addition, self-application causes the problem of double encoding in statically typed languages (Launchbury, 1991; Birkedal and Welinder, 1993). We avoided these difficulties by adopting a cogen approach (Thiemann, 1999) instead of self-application.

Type-directed partial evaluation (TDPE) (Danvy, 1996; Danvy, 1999) and normalization by evaluation (Berger and Schwichtenberg, 1991) are another cogen approach that takes the denotation (i.e., the compiled code) of a program with its type and constructs the long $\beta\eta$-normal form of the program. TDPE does not require BTA *except for* constants. Online TDPE (Danvy, 1997) avoids this problem of constants by partially adopting a cogen approach to online PE. Furthermore, a variant of online TDPE *coincides* with a cogen approach to online PE (Sumii and Kobayashi, 2000).

However, TDPE and its online variants incur the problem of code duplication, while our cogen approach to online PE avoids the problem by let-inserting a dynamic expression whenever necessary. For example, consider the following source program in Standard ML:

```
fn d => let val f = fn x => x
        in (f d, f, f)
        end
```

TDPE works on the denotation of this source program and yields its long $\beta\eta$-normal form, and therefore duplicates the function `fn x => x` and produces the following residual program.

```
fn d => (d, fn x => x, fn x => x)
```

In contrast, our method let-inserts the function and produces the following residual program:

```
fn d => let val f = fn x => x
        in (d, f, f)
        end
```

2.2. BTA

Our type-based representation analysis can be seen as a standard monovariant BTA (Henglein, 1991) extended with refined annotations. Several other BTA's also use annotations other than *static* and *dynamic*. Sperber's BTA (Sperber, 1996) has an annotation called *unknown*, which means "may be static and may be dynamic." However, his BTA rarely gives the *dynamic* annotation (at least in the source programs of our experiments) and often yields the same result as Ruf's BTA (Ruf, 1993), that is, the standard monovariant BTA with *dynamic* interpreted as *unknown*. Asai's BTA (Asai, 1999) has another annotation called *both*, which means "can be used both for reduction and for residualization." Our analysis subsumes all of these, as we will see in Section 8.

Another approach to more accurate BTA is polyvariance (Consel, 1993) or polymorphism (Henglein and Mossin, 1994), which can give

more than one annotation to each expression. It would also be possible to make our analysis more precise in this direction.

2.3. Use Analyses

Our analysis resembles type-based useless-variable elimination (Kobayashi, 1999) extended with sum types. In fact, useless-variable elimination can be used to remove unnecessary values and expressions in a generating extension. However, it does not remove unnecessary let-insertion.

As far as we are aware, there have been few analyses that detect never-duplicated expressions and remove unnecessary let-insertion *before* specialization. Similix's abstract occurrence counting analysis (Section 4.6 and Section 5.8 of Bondorf, 1990) has a similar goal. There also exist many results for inlining (e.g., Ashley, 1997), but they are orthogonal to our analysis because they work *after* specialization and never make the specialization faster.

Our analysis finds a kind of *linearity* of expressions (in the rough sense that they are "used once") in order to remove unnecessary let-insertion. In this respect, our analysis can be regarded as a combination of existing BTA's and linear type systems (e.g., Turner et al., 1995).

3. Naive Online PE

3.1. Naive Online PE with Let-Insertion

A standard way to implement online PE is to write an interpreter that takes an expression with an environment, and returns a *symbolic value*. A symbolic value is a pair of a static value for reduction and a dynamic expression for residualization. The static value is optional because it may be absent during specialization. Note that, even if the static value is present, the dynamic expression may still be necessary for residualization. The dynamic expression is *let-inserted*, i.e., bound to an identifier by a let-expression inserted in the residual program, in order to avoid duplicating, reordering, or discarding code or computation. This let-insertion can be implemented by exploiting delimited continuations (Danvy and Filinski, 1990), which can be manipulated by using control operators (Lawall and Danvy, 1994) or writing the partial evaluator in continuation-passing style (CPS) (Bondorf, 1992). Let us call this approach to online PE as "naive" online PE, in the sense that it (1) uses an interpreter, (2) keeps both an optional static value and a dynamic expression in every symbolic value, and (3) let-inserts each dynamic expression by manipulating delimited continuations.

```
(* env → exp → symval *)
fun onpe env (Var(x)) = lookup env x
  | onpe env (Abs(x, e)) =
    let val y = genid ()
    in (SOME(Func(fn sv => onpe (extend env x sv) e)),
        slet(Abs(y, rlet(fn _ =>
                  Var(getid(onpe (extend env x (NONE, y)) e))))))
    end
  | onpe env (App(e1, e2)) =
    let val arg = onpe env e2
    in case onpe env e1
         of (SOME(Func(vfunc)), _) => vfunc arg
          | (NONE, efunc) =>
                   (NONE, slet(App(Var(efunc), Var(getid arg))))
    end
  | onpe env (Pair(e1, e2)) =
    let val sv1 = onpe env e1
        val sv2 = onpe env e2
    in (SOME(Cons(sv1, sv2)),
        slet(Pair(Var(getid sv1), Var(getid sv2))))
    end
  | onpe env (Fst(e)) =
    (case onpe env e
       of (SOME(Cons(sv1, _)), _) => sv1
        | (NONE, epair) => (NONE, slet(Fst(Var(epair)))))
  | onpe env (Snd(e)) =
    (case onpe env e
       of (SOME(Cons(_, sv2)), _) => sv2
        | (NONE, epair) => (NONE, slet(Snd(Var(epair)))))

(* exp → exp *)
fun main e = pp(rlet(fn _ => Var(getid(onpe nil e))))
```

Figure 1. Naive online PE with let-insertion

In this article, we adopt the simply typed call-by-value λ-calculus with effects (in the form of primitive operators such as `print`, which appear as free variables of a term) as the object language. A naive online partial evaluator for this language can be implemented as shown in Figure 1, with the auxiliary definitions in Figure 2 and the let-inserting operators in Figure 3. For simplicity, only variables, functions, and pairs are considered here. Let-expressions are treated as syntax sugar – that is, **let** $x = e_1$ **in** e_2 is defined to be $(\lambda x.\, e_2)e_1$. Other constructs such as integers and booleans will be considered in Section 7.

```
type ident = string
val counter = ref 0
fun genid () = (counter := !counter + 1;
                "x" ^ Int.toString (!counter))

datatype exp =
    Var of ident
  | Abs of ident * exp
  | App of exp * exp
  | Pair of exp * exp
  | Fst of exp
  | Snd of exp
fun Let(x, e1, e2) = App(Abs(x, e2), e1)

datatype value =
    Func of symval -> symval
  | Cons of symval * symval
withtype symval = value option * ident
fun getid (_, x) = x (* symval → ident *)

type env = (string * symval) list
fun lookup ((y, v) :: env) x =
    if x = y then v else lookup env x
fun extend e x v = (x, v) :: e
```

Figure 2. Auxiliary definitions for naive online PE with let-insertion

```
(* exp → ident *)
fun slet e = shift (fn k => let val x = genid ()
                            in Let(x, e, reset (fn _ => k x))
                            end)
(* (unit → exp) → exp *)
val rlet = reset
```

Figure 3. Continuation-based let-inserting operators

For concreteness, we use Standard ML as the meta language. We also use Filinski's implementation (Filinski, 1994) of the control operators `shift` and `reset`, based on state and `call/cc`, with the answer type `ans` being the type `exp`. These control operators are usually used in the forms `reset (fn _ =>` E_1`)` and `shift (fn k =>` E_2`)`: the former delimits the context and evaluates the body E_1; the latter extracts the delimited context, binds it to the argument k, and evaluates the body E_2.

For value constructors (`Abs` and `Pair`), the partial evaluator (`onpe`) returns a symbolic value containing a static value. For value destructors (`App`, `Fst` and `Snd`), the partial evaluator checks whether the static values necessary for the reduction are present or absent. If they are present, the partial evaluator reduces the destructor and returns the result. If not, the partial evaluator residualizes the destructor and returns a symbolic value but no static value.

Let-insertion is implemented via the let-inserting operators (`rlet` and `slet`) in Figure 3. The former specifies a program point for let-insertion. It is interposed at the top level and wherever evaluation is delayed, for example at the body of λ-abstraction. The latter inserts a let-binding at the program point specified by the last `rlet`. It is interposed whenever expressions are generated, because they might be duplicated.

Too see how the partial evaluator works, let us consider the following examples.

- The partial evaluator reduces $\lambda x.\,(\lambda y.\,y)\,x$ to $\mathbf{let}\ x_4 = (\lambda x_1.\,\mathbf{let}\ x_3 = \lambda x_2.\,x_2\ \mathbf{in}\ x_1)\ \mathbf{in}\ x_4$, which the postprocessor (`pp` in `main` in Figure 1) then simplifies to $\lambda x.\,x$ modulo α-conversion.[1]

- $\lambda f.\,\lambda x.\,\mathbf{fst}(\mathbf{pair}(x, f\,x))$ is reduced to $\lambda f.\,\lambda x.\,\mathbf{let}\ y = f\,x\ \mathbf{in}\ x$ (with the postprocessing mentioned above). The dynamic function application $f\,x$ is let-inserted because it might cause dynamic effects if f is bound to a function such as $\lambda\_.\,$`print "hello"`. Without let-insertion, the residual program would be $\lambda f.\,\lambda x.\,x$, which is extensionally *in*equivalent to the source program in the presence of effects.

- $\lambda x.\,\mathbf{let}\ f = \lambda y.\,y\ \mathbf{in}\ \mathbf{pair}(f, f\,x)$ is reduced to $\lambda x.\,\mathbf{pair}(\lambda y.\,y, x)$. In this example, the function f is used both for reduction and for residualization, so the corresponding symbolic value should contain *both* the static value and the dynamic expression. This is the reason why a symbolic value (`symval`) is defined as a pair of an optional static value and a let-inserted dynamic expression (`value option * ident`) rather than as their sum (`Static of value | Dynamic of ident`).

[1] This postprocessor only inlines redundant let-bindings by transforming expressions of the form $\mathbf{let}\ x = e\ \mathbf{in}\ x$ or of the form $\mathbf{let}\ y = v\ \mathbf{in}\ e$ into the expression e, where v is a value and y does not appear free in e. Doing so never affects the efficiency of specialization or that of a residual program in any significant way.

3.2. Problems

Although the partial evaluator above is *sound* in the sense that it always produces a residual program extensionally equivalent to the source program, it is *inefficient* because of the following overhead.

1. Manipulation of delimited continuations for let-insertion. Even in SML/NJ, whose `call/cc` is quite efficient (Appel, 1992; Appel and Shao, 1996), the let-inserting online partial evaluator is about 3–4 times slower than a non-let-inserting one. Other implementations of delimited-continuation manipulation (e.g., writing the partial evaluator in CPS instead of using the control operators) also incur a similar overhead (Thiemann, 1999).

2. Interpretive overhead such as environment manipulation, syntax dispatch, and tagging.

3. Generation of unnecessary values and expressions. The partial evaluator generates both a static value and a dynamic expression for every value constructor (`Abs` or `Pair`), but uses only one of them in many cases. (It might help to generate them lazily, but doing so would also cause another overhead such as thunk manipulation, which necessitates another analysis such as strictness analysis for efficient specialization.)

4. Unnecessary tags (`SOME` and `NONE`) for the optional static value in a symbolic value. Many expressions always have a static value (respectively, no static value) and always get the `SOME` (respectively, `NONE`) tag throughout specialization.

5. Unnecessary let-insertion. The partial evaluator let-inserts many dynamic expressions that are never duplicated and have no effect. Existing partial evaluators such as Similix (Bondorf and Danvy, 1991) and call-by-value TDPE (Danvy, 1999) avoid this problem to some extent, e.g., by let-inserting only "non-trivial" dynamic expressions in function arguments, but such workarounds do not always solve the problem and sometimes risk code duplication (e.g., as illustrated in Section 2.1).

Among the problems above, the third and fourth problems are specific to online PE. The other problems are common to offline PE but more serious in online PE, where let-insertion is more frequent and self-application is more difficult.

4. State-Based Let-Insertion

In order to remove the overhead of delimited-continuation manipulation (the first problem in Section 3.2), we implement let-insertion by using state instead of continuations.

4.1. Intuition and Implementation

To begin with, let us recall how continuation-based let-insertion works. For example, consider the offline PE of $\lambda f.\lambda x.\lambda y.\mathbf{let}\ p = \mathbf{pair}(fx, fy)\ \mathbf{in}\ f$ with let-insertion. The PE is equivalent to the evaluation of the program below in a two-level λ-calculus with the let-inserting operators (*slet* and *rlet*). For readability, we omit identifier generation and unnecessary *rlet*'s.

$$\underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.rlet(\lambda\_.\mathbf{let}\ p = \mathbf{pair}(slet(f\ \underline{@}\ x), slet(f\ \underline{@}\ y))\ \mathbf{in}\ f)$$

The residual program should be $\underline{\lambda} f.\underline{\lambda} x.\underline{\mathbf{let}}\ z_1 = f \underline{@} x\ \underline{\mathbf{in}}\ \underline{\mathbf{let}}\ z_2 = f \underline{@} y\ \underline{\mathbf{in}}$ f with the dynamic function applications ($f \underline{@} x$ and $f \underline{@} y$) let-inserted. The continuation-based let-inserting operators achieve this as below, according to their definitions in Figure 3 and the intuitive semantics of *shift* and *reset* (Danvy and Filinski, 1989). Again, we omit unnecessary *rlet*'s for simplicity.

$$\begin{aligned}
& \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.rlet(\lambda_. \\
& \quad \mathbf{let}\ p = \mathbf{pair}(slet(f\ \underline{@}\ x), slet(f\ \underline{@}\ y))\ \mathbf{in}\ f) \\
\rightarrow^* \ & \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.\underline{\mathbf{let}}\ z_1 = f\ \underline{@}\ x\ \underline{\mathbf{in}}\ rlet(\lambda_. \\
& \quad \mathbf{let}\ p = \mathbf{pair}(z_1, slet(f\ \underline{@}\ y))\ \mathbf{in}\ f) \\
\rightarrow^* \ & \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.\underline{\mathbf{let}}\ z_1 = f\ \underline{@}\ x\ \underline{\mathbf{in}}\ \underline{\mathbf{let}}\ z_2 = f\ \underline{@}\ y\ \underline{\mathbf{in}}\ rlet(\lambda_. \\
& \quad \mathbf{let}\ p = \mathbf{pair}(z_1, z_2)\ \mathbf{in}\ f)
\end{aligned}$$

By regarding these as "one-step" reductions of *slet*, one can see that each *slet* appends a let-binding to the context outside the *rlet*, and returns the identifier of the let-binding. Let us make this context explicit by rewriting the reductions as follows.

$$\begin{aligned}
& \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.k(rlet(\lambda_.\mathbf{let}\ p = \mathbf{pair}(slet(f\ \underline{@}\ x), slet(f\ \underline{@}\ y))\ \mathbf{in}\ f)) \\
& \quad \text{where } k = id \\
\rightarrow^* \ & \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.k(rlet(\lambda_.\mathbf{let}\ p = \mathbf{pair}(z_1, slet(f\ \underline{@}\ y))\ \mathbf{in}\ f)) \\
& \quad \text{where } k = id \circ (\underline{\mathbf{let}}\ z_1 = f\ \underline{@}\ x\ \underline{\mathbf{in}}\ [\,]) \\
\rightarrow^* \ & \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.k(rlet(\lambda_.\mathbf{let}\ p = \mathbf{pair}(z_1, z_2)\ \mathbf{in}\ f)) \\
& \quad \text{where } k = id \circ (\underline{\mathbf{let}}\ z_1 = f\ \underline{@}\ x\ \underline{\mathbf{in}}\ [\,]) \circ (\underline{\mathbf{let}}\ z_2 = f\ \underline{@}\ y\ \underline{\mathbf{in}}\ [\,])
\end{aligned}$$

```
val empty : exp -> exp = fn e => e
val acc = ref empty

(* exp → ident *)
fun slet e =
    let val x = genid ()
    in (acc := (!acc o (fn body => Let(x, e, body)));
        x)
    end

(* (unit → exp) → exp *)
fun rlet thunk =
    let val tmp = !acc
        val _ = (acc := empty)
        val body = thunk ()
        val head = !acc
        val _ = (acc := tmp)
    in head body
    end
```

Figure 4. State-based let-inserting operators

The basic idea of state-based let-insertion is to accumulate the let-bindings in a store instead of the context. Let us name the store σ. Roughly speaking, state-based let-inserting operators should work as follows.

$$
\begin{aligned}
& \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.rlet(\lambda\_.\mathbf{let}\ p = \mathbf{pair}(slet(f\ \underline{@}\ x), slet(f\ \underline{@}\ y))\ \mathbf{in}\ f) \\
\rightarrow^* \ & \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.\ldots.(\mathbf{let}\ p = \mathbf{pair}(slet(f\ \underline{@}\ x), slet(f\ \underline{@}\ y))\ \mathbf{in}\ f) \\
& \quad \text{where } \sigma = id \\
\rightarrow^* \ & \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.\ldots.(\mathbf{let}\ p = \mathbf{pair}(z_1, slet(f\ \underline{@}\ y))\ \mathbf{in}\ f) \\
& \quad \text{where } \sigma = id \circ (\underline{\mathbf{let}}\ z_1 = f\ \underline{@}\ x\ \underline{\mathbf{in}}\ [\,]) \\
\rightarrow^* \ & \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.\ldots.(\mathbf{let}\ p = \mathbf{pair}(z_1, z_2)\ \mathbf{in}\ f) \\
& \quad \text{where } \sigma = id \circ (\underline{\mathbf{let}}\ z_1 = f\ \underline{@}\ x\ \underline{\mathbf{in}}\ [\,]) \circ (\underline{\mathbf{let}}\ z_2 = f\ \underline{@}\ y\ \underline{\mathbf{in}}\ [\,]) \\
\rightarrow^* \ & \underline{\lambda} f.\underline{\lambda} x.\underline{\lambda} y.\underline{\mathbf{let}}\ z_1 = f\ \underline{@}\ x\ \underline{\mathbf{in}}\ \underline{\mathbf{let}}\ z_2 = f\ \underline{@}\ y\ \underline{\mathbf{in}}\ f
\end{aligned}
$$

What should they do to achieve this? Obviously, *slet* should append a let-binding to the store and return the identifier. Let us consider what *rlet* should do before and after evaluating its operand.

– Before evaluating the operand, *rlet* should initialize the store with an identity function. In addition, in order to preserve the previous let-bindings, it should also save the previous store to a temporary variable.

– After evaluating the operand, *rlet* should insert the let-bindings in the store. In addition, in order to recover the previous let-bindings, it should also restore the previous store from the temporary variable.

As a whole, the state-based let-inserting operators can be implemented as shown in Figure 4. The variable `acc` refers to a function accumulating let-bindings, and the operator `slet` appends another let-binding to the function. The operator `rlet` saves the function to a temporary variable, reinitializes the reference, evaluates the thunk, inserts the let-bindings, and restores the function from the temporary variable.

Note that state-based let-insertion does not subsume continuation-based PE (Lawall and Danvy, 1994), whose goal is not only context propagation in let-expressions but also context *duplication* in conditional expressions.[2] However, state-based let-insertion suffices for preserving effects and avoiding code duplication by inserting let-bindings. It can straightforwardly deal with conditional expressions by interposing an `rlet` at every branch when the condition is dynamic, though it cannot reduce, say, $1 + (\textbf{if}\ x\ \textbf{then}\ 2\ \textbf{else}\ y)$ to $\textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 1 + y$ when x is dynamic.

4.2. Formalization

We show the correctness of state-based let-insertion by proving its equivalence to continuation-based let-insertion. To this end, we (1) define a language with let-inserting operators (*slet* and *rlet*), (2) implement the language by transformations into continuation-passing style (CPS) or state-passing style (SPS), (3) define a correspondence between the two implementations, which is similar to logical relations, and (4) prove their equivalence by using the correspondence.[3]

For brevity, we adopt the simply typed call-by-value λ-calculus with let-inserting operators as the source language of the transformations. (This may be an oversimplification, but we conjecture that it is possible to extend our result to practical languages such as ML by adapting

[2] It is possible as well to treat conditional expressions by using state instead of continuations (Zhe Yang, personal communication, January 2000), but it remains to see whether the state-based "if"-insertion is correct and efficient, because it is more complex than state-based let-insertion and because it duplicates some static computation.

[3] Independently of us, Filinski (Section 3 of Filinski, 2001) gave another correctness proof of state-based let-insertion, based on the notion of the *accumulation monad*, in the context of TDPE. Our proof is more general in that we prove the equivalence of continuation-based and state-based let-inserting operators under *any* context in the language (though Filinski's proof could probably be generalized as well).

$$
\begin{array}{lll}
e \text{ (expression)} ::= & x & \text{(variable)} \\
\quad | & \lambda x.e & (\lambda\text{-abstraction}) \\
\quad | & e_1 e_2 & \text{(function application)} \\
\quad | & \delta_i^{(b_{j_1},\ldots,b_{j_n})\to b_j}(e_1,\ldots,e_n) & \text{(primitive operator)} \\
\quad | & \mathit{slet}(e) & \\
\quad | & \mathit{rlet}(\lambda\_.e) & \\
v \text{ (value)} ::= & x & \text{(variable)} \\
\quad | & \lambda x.e & (\lambda\text{-abstraction}) \\
\quad | & \delta_i^{b_j} & \text{(constant)} \\
\tau \text{ (type)} ::= & b_i & \text{(base type)} \\
\quad | & \tau_1 \to \tau_2 & \text{(function type)}
\end{array}
$$

$$
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : b_{j_1} \quad \ldots \quad \Gamma \vdash e_n : b_{j_n}}{\Gamma \vdash \delta_i^{(b_{j_1},\ldots,b_{j_n})\to b_j}(e_1,\ldots,e_n) : b_j}
$$

$$
\frac{\Gamma \vdash e : \texttt{exp}}{\Gamma \vdash \mathit{slet}(e) : \texttt{ident}} \qquad \frac{\Gamma \vdash e : \texttt{exp}}{\Gamma \vdash \mathit{rlet}(\lambda\_.e) : \texttt{exp}}
$$

Figure 5. Syntax and typing rules of the source language

existing work—such as Birkedal and Harper, 1999—on logical relations in more expressive languages.) The syntax and the typing rules of the source language are given in Figure 5. We take `exp` and `ident` as base types. We regard a primitive operator with no operands as a constant, and abbreviate $\delta_i^{()\to b_j}()$ to $\delta_i^{b_j}$. We also abbreviate $\delta_i^{(b_{j_1},\ldots,b_{j_n})\to b_j}(e_1,\ldots,e_n)$ to $\delta_i(e_1,\ldots,e_n)$ when the types are unimportant.

The semantics of the source language is defined by transformations into CPS or SPS, as shown in Figure 6. It can be obtained with standard CPS (respectively, SPS) transformation of the continuation-based (respectively, state-based) let-inserting operators. For concreteness, we assume left-to-right evaluation order. The target language of the transformations is the simply typed call-by-value λ-calculus with pairs, primitive operators, and state for identifier generation. (Let-expressions can be defined either as a built-in construct or as syntactic sugar. This choice does not matter to the following formalization.) Be aware that the state for identifier generation is independent of the state for state-based let-insertion. For clarity, we write $k[e]$ for an application of a continuation k to an expression e, and $\langle e, \sigma \rangle$ for a pair of

$\tilde{x} : CPSVal_{\tilde{\tau}} \vdash [\![e]\!]_c : CPSComp_\tau$ for $\tilde{x} : \tilde{\tau} \vdash e : \tau$

$$
\begin{array}{rcl}
CPSComp_\tau & = & Cont_\tau \to \mathtt{exp} \\
Cont_\tau & = & CPSVal_\tau \to \mathtt{exp} \\
CPSVal_{b_i} & = & b_i \\
CPSVal_{\tau_1 \to \tau_2} & = & CPSVal_{\tau_1} \to CPSComp_{\tau_2}
\end{array}
$$

$$
\begin{array}{rcl}
[\![x]\!]_c & = & \lambda k.k[x] \\
[\![\lambda x.e]\!]_c & = & \lambda k.k[\lambda x.[\![e]\!]_c] \\
[\![e_1 e_2]\!]_c & = & \lambda k.[\![e_1]\!]_c(\lambda x_1.[\![e_2]\!]_c(\lambda x_2.x_1 x_2 k)) \\
[\![\delta_i(e_1,\ldots,e_n)]\!]_c & = & \lambda k.[\![e_1]\!]_c(\lambda x_1.\ldots.[\![e_n]\!]_c(\lambda x_n.k[\delta_i(x_1,\ldots,x_n)])) \\
[\![slet(e)]\!]_c & = & \lambda k.[\![e]\!]_c(\lambda x.\mathbf{let}\ y = \mathtt{genid}()\ \mathbf{in}\ \mathtt{Let}(y,x,k[y])) \\
[\![rlet(\lambda\_.e)]\!]_c & = & \lambda k.k[[\![e]\!]_c id]
\end{array}
$$

$\tilde{x} : SPSVal_{\tilde{\tau}} \vdash [\![e]\!]_s : SPSComp_\tau$ for $\tilde{x} : \tilde{\tau} \vdash e : \tau$

$$
\begin{array}{rcl}
SPSComp_\tau & = & State \to SPSVal_\tau \times State \\
State & = & \mathtt{exp} \to \mathtt{exp} \\
SPSVal_{b_i} & = & b_i \\
SPSVal_{\tau_1 \to \tau_2} & = & SPSVal_{\tau_1} \to SPSComp_{\tau_2}
\end{array}
$$

$$
\begin{array}{rcl}
[\![x]\!]_s & = & \lambda\sigma.\langle x,\sigma\rangle \\
[\![\lambda x.e]\!]_s & = & \lambda\sigma.\langle \lambda x.[\![e]\!]_s,\sigma\rangle \\
[\![e_1 e_2]\!]_s & = & \lambda\sigma.\mathbf{let}\ \langle x_1,\sigma_1\rangle = [\![e_1]\!]_s\sigma\ \mathbf{in} \\
 & & \quad \mathbf{let}\ \langle x_2,\sigma_2\rangle = [\![e_2]\!]_s\sigma_1\ \mathbf{in}\ x_1 x_2 \sigma_2 \\
[\![\delta_i(e_1,\ldots,e_n)]\!]_s & = & \lambda\sigma.\mathbf{let}\ \langle x_1,\sigma_1\rangle = [\![e_1]\!]_s\sigma\ \mathbf{in} \\
 & & \quad \ldots \\
 & & \quad \mathbf{let}\ \langle x_n,\sigma_n\rangle = [\![e_n]\!]_s\sigma_{n-1}\ \mathbf{in}\ \langle \delta_i(x_1,\ldots,x_n),\sigma_n\rangle \\
[\![slet(e)]\!]_s & = & \lambda\sigma.\mathbf{let}\ \langle x,\sigma'\rangle = [\![e]\!]_s\sigma\ \mathbf{in} \\
 & & \quad \mathbf{let}\ y = \mathtt{genid}()\ \mathbf{in}\ \langle y, \sigma' \circ (\lambda x'.\mathtt{Let}(y,x,x'))\rangle \\
[\![rlet(\lambda\_.e)]\!]_s & = & \lambda\sigma.\mathbf{let}\ \langle x,\sigma'\rangle = [\![e]\!]_s id\ \mathbf{in}\ \langle \sigma' x,\sigma\rangle
\end{array}
$$

Figure 6. Definition of $[\![\ ]\!]_c$ and $[\![\ ]\!]_s$

$$
\begin{array}{lrcll}
(\beta) & \textbf{let } x = v \textbf{ in } e & \approx & [v/x]e & \\
(\beta\text{-p}) & \textbf{let } \langle x, y\rangle = \langle v, w\rangle \textbf{ in } e & \approx & [v/x, w/y]e & \\
(\text{unit}) & \textbf{let } x = e \textbf{ in } x & \approx & e & \\
(\lambda) & (\lambda x.e_1)e_2 & \approx & \textbf{let } x = e_2 \textbf{ in } e_1 & \\
(\eta) & \lambda x.vx & \approx & v & (x \notin \mathit{free}(v)) \\
(\text{app}) & e_1 e_2 & \approx & \textbf{let } x = e_1 \textbf{ in let } y = e_2 \textbf{ in } xy & \\
 & & & & (x \notin \mathit{free}(e_2)) \\
(\text{prim}) & \delta_i(e_1, \ldots, e_n) & \approx & \textbf{let } \tilde{x} = \tilde{e} \textbf{ in } \delta_i(x_1, \ldots, x_n) & \\
 & & & & (x_i \notin \mathit{free}(e_j) \text{ for any } i < j) \\
(\text{pair}) & \langle e_1, e_2\rangle & \approx & \textbf{let } x = e_1 \textbf{ in let } y = e_2 \textbf{ in } \langle x, y\rangle & \\
 & & & & (x, y \notin \mathit{free}(e_1, e_2)) \\
(\text{assoc}) & \multicolumn{4}{l}{\textbf{let } x = (\textbf{let } y = e_1 \textbf{ in } e_2) \textbf{ in } e_3} \\
 & \approx & \multicolumn{3}{l}{\textbf{let } y = e_1 \textbf{ in let } x = e_2 \textbf{ in } e_3 \qquad (x \notin \mathit{free}(e_1))} \\
(\text{assoc-p1}) & \multicolumn{4}{l}{\textbf{let } \langle x, y\rangle = (\textbf{let } z = e_1 \textbf{ in } e_2) \textbf{ in } e_3} \\
 & \approx & \multicolumn{3}{l}{\textbf{let } z = e_1 \textbf{ in let } \langle x, y\rangle = e_2 \textbf{ in } e_3 \qquad (x, y \notin \mathit{free}(e_1))} \\
(\text{assoc-p2}) & \multicolumn{4}{l}{\textbf{let } z = (\textbf{let } \langle x, y\rangle = e_1 \textbf{ in } e_2) \textbf{ in } e_3} \\
 & \approx & \multicolumn{3}{l}{\textbf{let } \langle x, y\rangle = e_1 \textbf{ in let } z = e_2 \textbf{ in } e_3 \qquad (z \notin \mathit{free}(e_1))} \\
(\text{assoc-p3}) & \multicolumn{4}{l}{\textbf{let } \langle x_1, y_1\rangle = (\textbf{let } \langle x_2, y_2\rangle = e_1 \textbf{ in } e_2) \textbf{ in } e_3} \\
 & \approx & \multicolumn{3}{l}{\textbf{let } \langle x_2, y_2\rangle = e_1 \textbf{ in let } \langle x_1, y_1\rangle = e_2 \textbf{ in } e_3} \\
 & & & & (x_1, y_1 \notin \mathit{free}(e_1))
\end{array}
$$

Figure 7. Congruence in the target language

an expression e and a state σ. Semantically, they are not different from ordinary applications and pairs. We also use $\tilde{}$ to abbreviate a sequence – e.g., we write $\tilde{x} : \tilde{\tau}$ for $x_1 : \tau_1, \ldots, x_n : \tau_n$. As usual, we implicitly apply α-conversion to avoid capturing free variables. We write $\mathit{free}(\tilde{e})$ to denote the set of free variables in the expressions $\tilde{e}$. We also write id for $\lambda z.z$ and $e_1 \circ e_2$ for $\textbf{let } f = e_1 \textbf{ in let } g = e_2 \textbf{ in } \lambda x.f(gx)$ where $f \notin \mathit{free}(e_2)$. Furthermore, we write $\Gamma \vdash e : \tau$ to denote the standard simple typing relation of the target language.

We write $\approx$ for the congruence (taking effects into account) in the target language. We assume that this congruence satisfies the rules in Figure 7. They can be proved by encoding the target language into Moggi's meta language (Moggi, 1991) and using the congruence in the meta language. By using $\approx$, we define a correspondence similar to logical relations, between closed values and computations in CPS and SPS as below. We use superscripts (such as e^c, e^s, e^k and e^σ) to distinguish meta-variables.

Definition 4.1 *$\sim_\tau^{val}$ and $\sim_\tau^{comp}$ are the relations defined as follows by induction on τ.*

$$\begin{aligned}
\sim_{b_i}^{val} &= \{(e^c, e^s) \mid \emptyset \vdash e^c : b_i \wedge \emptyset \vdash e^s : b_i \wedge e^c \approx e^s\} \\
\sim_{\tau_1 \to \tau_2}^{val} &= \{(e_1^c, e_1^s) \mid e_1^c e_2^c \sim_{\tau_2}^{comp} e_1^s e_2^s \text{ for all } e_2^c \sim_{\tau_1}^{val} e_2^s\} \\
\sim_\tau^{comp} &= \{(e^c, e^s) \mid e^c \approx \lambda k.\mathbf{let}\ \tilde{x} = \tilde{e}^k\ \mathbf{in}\ (x_i \circ k)[v^c], \\
&\qquad\qquad e^s \approx \lambda\sigma.\mathbf{let}\ \tilde{x} = \tilde{e}^\sigma\ \mathbf{in}\ \langle v^s, \sigma \circ x_i\rangle, \\
&\qquad\qquad \tilde{e}^k \approx \tilde{e}^\sigma, v^c \sim_\tau^{val} v^s\}
\end{aligned}$$

We omit τ in $\sim_\tau^{val}$ and $\sim_\tau^{comp}$ when it is unimportant. It is easy to see that $\sim^{val}$ and $\sim^{comp}$ are left- and right-closed with respect to $\approx$.

The intuition behind $\sim^{comp}$ is as follows. Given a continuation or a state, both the continuation-based and state-based implementations first perform equivalent computations ($\tilde{e}^k \approx \tilde{e}^\sigma$) including identifier generation, and then return equivalent values ($v^c \sim_\tau^{val} v^s$) *except for* the let-bindings. The continuation-based implementation prepends the let-bindings to the continuation $(x_i \circ k)$ and thereby accumulates them in the context outside *rlet*. On the other hand, the state-based implementation appends the let-bindings to the state $(\sigma \circ x_i)$ and thereby accumulates them in the store inside *rlet*. Here, the variable x_i is one of the variables $\tilde{x}$, which denotes the result of the computation e_i^k or e_i^σ that yields a function containing the let-bindings.

With $\sim^{comp}$, the equivalence of $[\![e]\!]_c$ and $[\![e]\!]_s$ can be proved as follows.

Theorem 4.2 *Let $\Gamma \vdash e : \tau$ be any well-typed expression in the source language and $z_1, \ldots, z_n$ be the free variables of e. Then,*

$$[\tilde{w}^c/\tilde{z}][\![e]\!]_c \ \sim_\tau^{comp}\ [\tilde{w}^s/\tilde{z}][\![e]\!]_s$$

for any $\tilde{w}^c \sim^{val} \tilde{w}^s$ of appropriate types.

Proof. By induction on the structure of e. See the Appendix for details. □

Corollary 4.3 *For any $\emptyset \vdash e : \mathtt{exp}$ and $\emptyset \vdash v^\sigma : State$,*

$$[\![rlet(\lambda\_.e)]\!]_c id \ \approx\ \mathbf{let}\ \langle x, \_\rangle = [\![rlet(\lambda\_.e)]\!]_s v^\sigma\ \mathbf{in}\ x$$

That is, a program with a top-level rlet denotes the same computation under the continuation-based and state-based implementations.

Proof. By the theorem above. See the Appendix for details. □

```
datatype 'a hexp =
    HVar of 'a
  | HAbs of 'a -> 'a hexp
  | HApp of 'a hexp * 'a hexp
  | HPair of 'a hexp * 'a hexp
  | HFst of 'a hexp
  | HSnd of 'a hexp
```

Figure 8. Higher-order abstract syntax

5. The Cogen Approach to Online PE

The interpretive overhead of the partial evaluator (the second problem in Section 3.2) can be removed by taking the *cogen approach*, that is, using a program generator generator (*cogen*) in the first place instead of generating it by self-application. The cogen can be derived from an interpreter (Thiemann, 1999) by means of higher-order abstract syntax (Pfenning and Elliott, 1988), deforestation (Wadler, 1990), and untagging. The derivation is similar to Thiemann's, although it has never been applied to online PE, as far as we know. For the rest of this section, we roughly sketch each step of the derivation. The correctness proof of the derivation is omitted because it would also be similar to Thiemann's proof.

5.1. Higher-Order Abstract Syntax

First, we eliminate the overhead of environment manipulation by using higher-order abstract syntax (HOAS).

HOAS is a meta-programming technique that represents binding in the object language by binding in the meta language (Pfenning and Elliott, 1988). HOAS of our object language can be defined as shown in Figure 8. For example, $\lambda x.\,(\lambda y.\,y)\,x$ can be represented as:

```
HAbs(fn x => HApp(HAbs(fn y => HVar(y)), HVar(x)))
  : 'a hexp
```

instead of:

```
Abs("x", App(Abs("y", Var("y")), Var("x"))) : exp
```

Here, the type `'a hexp` of expressions is parameterized over the type `'a` of the values of variables. For example, in an ordinary (i.e., non-partial) evaluator for the HOAS, `'a hexp` would be instantiated with `'a` being `value`. Similarly, in a pretty-printer for the HOAS, `'a` would be bound to `string`.

By using HOAS for the input, we obtain the partial evaluator in Figure 9 from the naive online partial evaluator in Figure 1. Here, `'a` in

```
(* type symval = value option * ident *)

(* symval hexp → symval *)
fun onpe (HVar(x)) = x
  | onpe (HAbs(f)) =
    let val y = genid ()
    in (SOME(Func(fn sv => onpe (f sv))),
        slet(Abs(y, rlet(fn _ =>
                              Var(getid(onpe (f (NONE, y))))))))
    end
  | onpe (HApp(e1, e2)) =
    let val arg = onpe e2
    in case onpe e1
         of (SOME(Func(vfunc)), _) => vfunc arg
          | (NONE, efunc) =>
                   (NONE, slet(App(Var(efunc), Var(getid arg))))
    end
...
(* symval hexp → exp *)
fun main e = pp(rlet(fn _ => Var(getid(onpe e))))
```

Figure 9. Online PE using HOAS for input (excerpt)

`'a hexp` is instantiated to `symval`. Since binding in the object language is replaced by binding in the meta language, `lookup` and `extend` are replaced by variables and function applications in the meta language.

It is also possible to use HOAS for the output. Doing so would simplify the implementation—especially in a pure functional language—by moving the identifier generator from the partial evaluator to the pretty-printer. Here, however, we do not take this approach, in order to keep the change as small as possible (and make the derivation as similar as possible to Thiemann's).

5.2. Deforestation

Next, we eliminate the overhead of syntax dispatch by composing the syntax constructors with the partial evaluator. When partially evaluating an expression, a user first constructs the expression by using the syntax constructors (`HAbs`, `HApp`, etc.) and then destructs it by using the partial evaluator (`onpe`). However, this is both a waste of memory to store the intermediate data structure and a waste of time to traverse the intermediate data structure.

```
(* type symval = value option * ident *)

(* (symval → symval) → symval *)
fun abs f =
    let val y = genid ()
    in (SOME(Func(fn sv => f sv)),
        slet(Abs(y, rlet(fn _ => Var(getid(f (NONE, y)))))))
    end

(* symval * symval → symval *)
fun app((SOME(Func(vfunc)), _), arg) = vfunc arg
  | app((NONE, efunc), arg) =
                   (NONE, slet(App(Var(efunc), Var(getid arg))))
...
(* (unit → symval) → exp *)
fun main thunk = pp(rlet(fn _ => Var(getid(thunk ()))))
```

Figure 10. Online PE with the syntax constructors composed (excerpt)

Thus, in order to eliminate the intermediate expression, we compose the syntax constructors and the partial evaluator, achieving deforestation (Wadler, 1990). The result is shown in Figure 10.

Note that, when applying `main`, a user needs to delay the evaluation of the argument by using a thunk, in order to interpose the `rlet` *before* the composed syntax constructors partially evaluate themselves. (If there were conditional expressions in the object language, one should also use thunks to delay the partial evaluation of each branch of a conditional expression.)

5.3. Untagging

Last, we eliminate the overhead of tagging by omitting the tags `Func` and `Cons`.

Before the deforestation above, the tags were mandatory because the *type* of the output of the partial evaluator was dependent on the *value* of the input to the partial evaluator. For instance, if the input was either `Pair` or `Abs`, the output had either a pair type or a function type, respectively. This is no longer a problem because the previous derivation have divided the single dependent function into several polymorphic combinators (Yang, 1998). Thus, as long as the source program is well-typed in the object language (i.e., the simply typed λ-calculus), its generating extension is also well-typed in the meta language (i.e., Standard ML), even without the tags.

```
type 'a symval = 'a option * ident

(* ('a symval → 'b symval) →
   ('a symval → 'b symval) symval *)
fun abs f =
    let val y = genid ()
    in (SOME(f),
        slet(Abs(y, rlet(fn _ => Var(getid(f (NONE, y)))))))
    end

(* ('a symval → 'b symval) symval * 'a symval → 'b symval *)
fun app((SOME(vfunc), _), arg) = vfunc arg
  | app((NONE, efunc), arg) =
                   (NONE, slet(App(Var(efunc), Var(getid arg))))

(* 'a symval * 'b symval -> ('a symval * 'b symval) symval *)
fun pair(sv1, sv2) =
    (SOME(sv1, sv2),
     slet(Pair(Var(getid sv1), Var(getid sv2))))

(* ('a symval * 'b symval) symval -> 'a symval *)
fun fst(SOME(sv1, _), _) = sv1
  | fst(NONE, epair) = (NONE, slet(Fst(Var(epair))))

(* ('a symval * 'b symval) symval -> 'b symval *)
fun snd(SOME(_, sv2), _) = sv2
  | snd(NONE, epair) = (NONE, slet(Snd(Var(epair))))

(* (unit → 'a symval) → exp *)
fun main thunk = pp(rlet(fn _ => Var(getid(thunk ()))))
```

Figure 11. Combinators for online PE

The result of this untagging is the combinators in Figure 11. For example, they work as follows.

```
- main(fn _ => abs(fn x => app(abs(fn y => y), x)));
val it = Abs ("x1",Var "x1") : exp
```

Thus, these combinators transform $\lambda x.\,(\lambda y.\,y)\,x$ to $\lambda x.\,x$.

6. Type-Based Representation Analysis

The last and largest overhead is caused by the "online-ness" itself (the third, fourth, and fifth problems in Section 3.2); we remove this

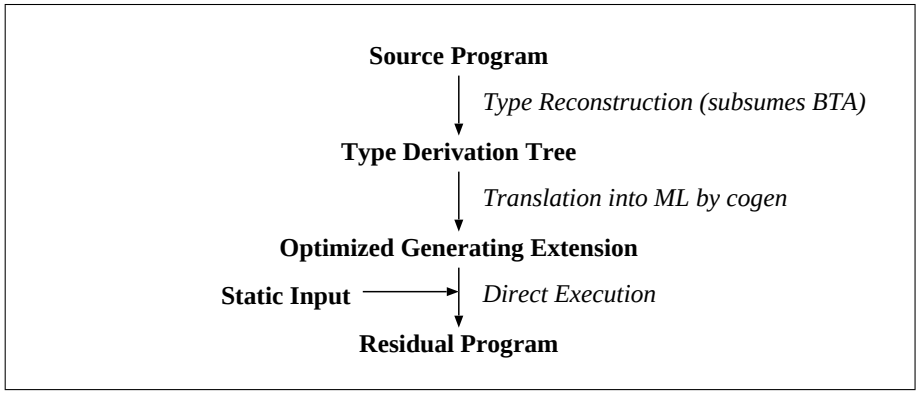

Figure 12. Framework of our method

overhead by optimizing the generating extension—more specifically, the *representations* of symbolic values in the generating extension.

For example, consider the online partial evaluation of $\lambda x.\,(\lambda y.\,y)\,x$ using the combinators in Figure 11. The symbolic values for $\lambda x.\,(\lambda y.\,y)\,x$ and $\lambda y.\,y$ contain both a tagged static value and a let-inserted dynamic expression and therefore have a type of the form $(\tau_1 \to \tau_2)$ `option * ident`. However, the symbolic value for $\lambda x.\,(\lambda y.\,y)\,x$ could contain only the non-let-inserted dynamic expression and therefore have the type `unit * exp` or just `exp`, because the static value is never used and the dynamic expression is never duplicated (and has no effect). Similarly, the symbolic value for $\lambda y.\,y$ could contain only the untagged static value and therefore have a type of the form $(\tau_1 \to \tau_2)$ `* unit` or just $\tau_1 \to \tau_2$, because the static value is always present and the dynamic expression is never used in the generating extension.

The rest of this section presents a type system for such an analysis and an optimization as above. Sections 6.1 and 6.2 define types and typing rules to specify what representations are *valid* in the sense that they never cause code duplication or binding-time mismatch in a generating extension. Note that there are many such representations possible in general – for example, it is "valid" in the sense above even to represent everything as a pair of an optional static value and a let-inserted dynamic expression, that is, a value of type τ `option * ident`. Section 6.3 gives a type reconstruction algorithm that, for a given source program, infers the "best" (under a certain criterion) representation out of the valid representations and constructs a type derivation tree. Section 6.4 shows a translation from the type derivation tree into a generating extension.

As a whole, our partial evaluator works as shown in Figure 12. Given a source program, the cogen constructs a type derivation tree by using

$$
\begin{array}{lll}
\tau ::= \rho^{(s,d)} & & [\![\tau]\!] = [\![\tau]\!]_S \texttt{ * } [\![\tau]\!]_D \\
\\
\rho ::= \tau_1 \to \tau_2 & [\![\tau_1 \to \tau_2]\!]_S & = [\![\tau_1]\!] \to [\![\tau_2]\!] \\
\mid \tau_1 \times \tau_2 & [\![\tau_1 \times \tau_2]\!]_S & = [\![\tau_1]\!] \texttt{ * } [\![\tau_2]\!] \\
\mid \alpha & [\![\alpha]\!]_S & = \alpha \\
\\
s ::= 0 & [\![\rho^{(0,d)}]\!]_S & = \texttt{unit} \\
\mid \omega & [\![\rho^{(\omega,d)}]\!]_S & = [\![\rho]\!]_S \\
\mid \top & [\![\rho^{(\top,d)}]\!]_S & = [\![\rho]\!]_S \texttt{ option} \\
\\
d ::= 0 & [\![\rho^{(s,0)}]\!]_D & = \texttt{unit} \\
\mid 1 & [\![\rho^{(s,1)}]\!]_D & = \texttt{exp} \\
\mid \omega & [\![\rho^{(s,\omega)}]\!]_D & = \texttt{ident}
\end{array}
$$

Figure 13. Syntax and semantics of types

the type reconstruction algorithm, and translates it into a generating extension optimized according to the types; then, the generating extension takes a static input and produces a residual program.

6.1. Types

The syntax and the semantics of the types are given in Figure 13. An annotated type $\rho^{(s,d)}$ denotes a way to represent a symbolic value of the raw type ρ in a generating extension. The static part s indicates whether the static value is always absent (0), always present (ω), or optional ($\top$) in the generating extension. The dynamic part d indicates whether the dynamic expression is absent (0), present but not let-inserted (1), or present and let-inserted (ω). For readability, we write $\tau_1 \to^{(s,d)} \tau_2$ for $(\tau_1 \to \tau_2)^{(s,d)}$ and $\tau_1 \times^{(s,d)} \tau_2$ for $(\tau_1 \times \tau_2)^{(s,d)}$.

In the combinators in Figure 11, every symbolic value is represented as $\rho^{(\top,\omega)}$ (i.e., ρ `option * ident`) with the static value optional and the dynamic expression present and let-inserted. As mentioned above, however, the representation can be optimized in many ways. For instance, in the previous example, the symbolic value for $\lambda y.\, y$ can be represented as $\tau_1 \to^{(\omega,0)} \tau_2$ (i.e., `(`$[\![\tau_1]\!] \to [\![\tau_2]\!]$`) * unit`) with the static value always present and the dynamic expression absent. (The static value itself is a function that takes a symbolic value represented as τ_1 and returns a symbolic value represented as τ_2.) Section 6.2 gives a set of typing rules that specify when such optimization is valid in the sense that it causes no code duplication and no binding-time mismatch.

Definition of $\Gamma_1 \rightsquigarrow \Gamma_2$:

$$\Gamma_1 \rightsquigarrow \Gamma_2 \iff \Gamma_1(x) \rightsquigarrow \Gamma_2(x) \text{ for all } x \in dom(\Gamma_1) = dom(\Gamma_2)$$

Definition of $\tau_1 \rightsquigarrow \tau_2$:

$$\rho_1^{(s_1,d_1)} \rightsquigarrow \rho_2^{(s_2,d_2)} \iff \rho_1 = \rho_2 \wedge s_1 \rightsquigarrow_S s_2 \wedge d_1 \rightsquigarrow_D d_2$$

Definition of $s_1 \rightsquigarrow_S s_2$:

$$s \rightsquigarrow_S s \qquad s \rightsquigarrow_S 0 \qquad s \rightsquigarrow_S \top \qquad \text{(for any } s\text{)}$$

Definition of $d_1 \rightsquigarrow_D d_2$:

$$d \rightsquigarrow_D d \qquad d \rightsquigarrow_D 0 \qquad \omega \rightsquigarrow_D d \qquad 1 \rightsquigarrow_D \omega \qquad \text{(for any } d\text{)}$$

Figure 14. Coercibility of annotations

6.2. Typing Rules

In the typing rules, we use the relations and operators on the annotations in Figure 14 and Figure 15. We use Γ as the meta variable for type environments.

The relation $\tau_1 \rightsquigarrow \tau_2$ means that a symbolic value represented as τ_1 can be coerced into another symbolic value represented as τ_2. The definition is straightforward, given the semantics of the types in Figure 13. The relation $1 \rightsquigarrow_D \omega$ is defined to hold, because an expression (value of type `exp`, which corresponds to 1) is coercible to an identifier (value of type `ident`, which corresponds to ω) by means of let-insertion.[4]

The addition operation $d_1 + d_2$ is used in the typing rules for multi-operand operations, such as function application and pair construction. It indicates how to represent a dynamic expression represented as d_1 in one operand and represented as d_2 in the other operand. The addition $1+1$ is defined to be ω, because if the dynamic expression is represented as `exp` in both operands, it might be duplicated and should be let-inserted.

For example, consider the partial evaluation of $\mathbf{pair}(x, x)$ where x contains some dynamic expression. Since the dynamic expression will be residualized as `exp` in *both* operands of **pair**, it must be let-inserted beforehand in order not to be duplicated, even though it occurs only once in *each* element of the pair. That is, if the dynamic annotation on

[4] The let-insertion can be omitted when the expression is already a variable. This optimization can be implemented straightforwardly as `fun slet' (Var(x)) = x | slet' e = slet e`.

Definition of $d_1 + d_2$

$d_1 + d_2$	0	1	ω
0	0	1	ω
1	1	ω	ω
ω	ω	ω	ω

Definition of $\tau_1 + \tau_2$

$$\begin{aligned} \rho^{(s,d_1)} + \rho^{(s,d_2)} &= \rho^{(s,d_1+d_2)} \\ \rho_1^{(s_1,d_1)} + \rho_2^{(s_2,d_2)} &= \text{undefined} \quad (\text{if } \rho_1 \neq \rho_2 \text{ or } s_1 \neq s_2) \end{aligned}$$

Definition of $\Gamma_1 + \Gamma_2$

$$\begin{aligned} dom(\Gamma_1 + \Gamma_2) &= dom(\Gamma_1) \cup dom(\Gamma_2) \\ (\Gamma_1 + \Gamma_2)(x) &= \begin{cases} \Gamma_1(x) + \Gamma_2(x) \\ \qquad (\text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2)) \\ \Gamma_1(x) \quad (\text{if } x \in dom(\Gamma_1) \backslash dom(\Gamma_2)) \\ \Gamma_2(x) \quad (\text{if } x \in dom(\Gamma_2) \backslash dom(\Gamma_1)) \end{cases} \end{aligned}$$

Definition of $(s, d) \cdot d'$

$(s,d) \cdot d'$	0	1	ω
$(0,0)$	0	0	0
$(0,1)$	0	1	ω
$(0,\omega)$	0	1	ω
$(\omega,\_)$	0	ω	ω
$(\top,\_)$	0	ω	ω

Definition of $(s, d) \cdot \tau$

$$(s,d) \cdot \rho^{(s',d')} = \rho^{(s',(s,d)\cdot d')}$$

Definition of $(s, d) \cdot \Gamma$

$$\begin{aligned} dom((s,d) \cdot \Gamma) &= dom(\Gamma) \\ ((s,d) \cdot \Gamma)(x) &= (s,d) \cdot (\Gamma(x)) \end{aligned}$$

Figure 15. Addition and multiplication of annotations

x is 1 in the type environments for each element of the pair, it should be $1 + 1 = \omega$ in the type environment for the whole pair.

The multiplication operation $(s, d) \cdot d'$ is used in the typing rule for functions. It is explained later in detail. Note that it is a single operator taking three operands s, d and d'.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}(\text{Var})$$

$$\frac{\begin{array}{c}\Gamma \rightsquigarrow (s,d) \cdot \Gamma_0 \qquad (1)\\ d \neq 0 \implies s_1 \neq \omega \qquad (2)\\ \Gamma_0, x : \rho_1^{(s_1,d_1)} \vdash t : \tau_2\end{array}}{\Gamma \vdash \lambda x.\, t : \rho_1^{(s_1,d_1)} \rightarrow^{(s,d)} \tau_2}(\text{Abs})$$

$$\frac{\begin{array}{c}s \neq \omega \implies s_2 \neq \omega \qquad (3)\\ s \neq \omega \implies d \neq 0 \wedge d_1 \neq 0 \qquad (4)\\ \Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \rho_1^{(s_1,d_1)} \rightsquigarrow \tau_1 \qquad \tau_2 \rightsquigarrow \rho_2^{(s_2,d_2)}\\ \Gamma_1 \vdash t_1 : \tau_1 \rightarrow^{(s,d)} \tau_2 \qquad \Gamma_2 \vdash t_2 : \rho_1^{(s_1,d_1)}\end{array}}{\Gamma \vdash t_1\, t_2 : \rho_2^{(s_2,d_2)}}(\text{App})$$

$$\frac{\Gamma \rightsquigarrow \Gamma_1 + \Gamma_2 \qquad \Gamma_1 \vdash t_1 : \tau_1 \qquad \Gamma_2 \vdash t_2 : \tau_2}{\Gamma \vdash \mathbf{pair}(t_1, t_2) : \tau_1 \times^{(s,d)} \tau_2}(\text{Pair})$$

$$\frac{\begin{array}{c}s \neq \omega \implies s_1 \neq \omega \qquad (5)\\ s \neq \omega \wedge d_1 \neq 0 \implies d \neq 0 \qquad (6)\\ \Gamma \vdash t : \tau_1 \times^{(s,d)} \tau_2 \qquad \tau_1 \rightsquigarrow \rho_1^{(s_1,d_1)}\end{array}}{\Gamma \vdash \mathbf{fst}(t) : \rho_1^{(s_1,d_1)}}(\text{Fst})$$

$$\frac{\begin{array}{c}s \neq \omega \implies s_2 \neq \omega \qquad (5)\\ s \neq \omega \wedge d_2 \neq 0 \implies d \neq 0 \qquad (6)\\ \Gamma \vdash t : \tau_1 \times^{(s,d)} \tau_2 \qquad \tau_2 \rightsquigarrow \rho_2^{(s_2,d_2)}\end{array}}{\Gamma \vdash \mathbf{snd}(t) : \rho_2^{(s_2,d_2)}}(\text{Snd})$$

Figure 16. Typing rules

The typing rules are given in Figure 16. Since they are syntax-directed, every expression in a source program is elaborated (by the type reconstruction algorithm in Section 6.3) into an instance of the typing rules, which is translated (by the generating-extension generation algorithm in Section 6.4) into a combinator optimized according to the types. The typing rules are defined so that the generating extension causes no code duplication and no binding-time mismatch (i.e., type error in the generating extension). They are similar to the typing

rules of the simply typed λ-calculus, except for the constraints on the annotations. Their meanings are as follows.

(1) This constraint describes how the dynamic expression bound to a free variable of a function should be represented (Γ). The representation depends on how the dynamic expression is used when the function *body* is partially evaluated (Γ_0) and how the function *itself* is used (s, d). This relationship is defined by the three-operand operator $(s, d) \cdot \Gamma_0$ in Figure 15 on the basis of the following observations.

- If the static value of the function itself might be present ($s = \omega$ or $\top$), then it might be applied many times and the function body might also be partially evaluated many times.[5] Therefore, if the dynamic expression bound to a free variable of the function might be used when the function body is partially evaluated ($d' = 1$ or ω), then the dynamic expression should be let-inserted because it might be duplicated. (Consider, for example, **let** $f = \lambda x.\, y$ **in pair**$(f\,1, f\,2)$ where y is dynamic.) Thus, $(s, d) \cdot d'$ is defined to be ω when $s = \omega$ or $\top$ and $d' = 1$ or ω.
- If the static value of the function is unused ($s = 0$) and a dynamic expression for the function is required ($d = 1$ or ω), then the function body is partially evaluated just once when the function is residualized, regardless of whether it is let-inserted or not. Thus, $(s, d) \cdot d'$ is defined to be d' when $s = 0$ and $d = 1$ or ω.
- If the dynamic expression bound to a free variable of the function is unused ($d' = 0$) when the function body is partially evaluated, then the dynamic expression can be omitted. This is also the case if the function itself is unused ($s = d = 0$). Thus, $(s, d) \cdot d'$ is defined to be 0 when $d' = 0$ or $s = d = 0$.

(2) When the function is residualized ($d \neq 0$), the argument is bound to a fresh identifier with no static value ($s_1 \neq \omega$). Recall that, in the definition of the combinator `abs` in Figure 11, the argument is bound to a fresh identifier with no static value (`f (NONE, y)` in the fourth line) in order to residualize the function (`Abs(y,...)` in the same line).

[5] It would also be possible to distinguish the case where the static value of a function is applied only once, by extending the analysis with such annotations as "$s = 1$." We leave this extension for future work.

(3) If the static value of the function might be absent ($s \neq \omega$), then the static value of the result might also be absent ($s_2 \neq \omega$). Recall that, in the definition of the combinator `app`, if the static value of the function is absent (`NONE` in the second line), then the static value of the result is also absent (`NONE` in the third line).

(4) If the static value of the function is absent, then the dynamic function application should be let-inserted *regardless of* how the result is used, because the function application may cause dynamic effects (Hatcliff and Danvy, 1996; Lawall and Thiemann, 1997). Thus, whenever the static value of the function might be absent ($s \neq \omega$), the dynamic expressions of the operands should be present ($d \neq 0 \wedge d_1 \neq 0$) in order to residualize the function application.[6] Recall that, in the definition of `app`, if the static value of the function is absent (`NONE` in the second line), then the dynamic function application is let-inserted (`slet(App(...,...))` in the third line) and the dynamic expressions of the operands should be present (`efunc` and `getid arg` in the same line).

(5) If the static value of the pair might be absent ($s \neq \omega$), then the static value of the result might also be absent ($s_1 \neq \omega$ and $s_2 \neq \omega$). Recall that, in the definitions of `fst` and `snd`, if the static value of the pair is absent (the first `NONE` in the second lines), then the static value of the result is also absent (the second `NONE` in the same lines).

(6) If the static value of the pair might be absent ($s \neq \omega$), and if the dynamic expression of the result should be present ($d_1 \neq 0$ and $d_2 \neq 0$), then the dynamic expression of the operand should also be present ($d \neq 0$). Again, recall that, in the definitions of `fst` and `snd`, if the static value of the pair is absent (the first `NONE` in the second lines), then the dynamic expression of the operand should be present (`epair` in the same lines) in order to generate the dynamic expression of the result (`Fst(...)` and `Snd(...)` in the same lines).

The other constraints are straightforward, given the meanings of $\rightsquigarrow$ and $+$. It may seem somewhat surprising that the rule (Pair) has *no* constraint at all on the annotations s and d, but this is no problem because all the necessary constraints are imposed where the pair is *used*, that is, in the rules (Fst) and (Snd) (and at the top level, where all the dynamic annotations are required to be non-0).

[6] If the function application is known to be effect-free, e.g., by means of an effect analysis (Talpin and Jouvelot, 1994), then this constraint can be weakened to $s \neq \omega \wedge d_2 \neq 0 \Longrightarrow d \neq 0 \wedge d_1 \neq 0$.

6.3. Type Reconstruction

The typing rules above only specifies a set of possible, valid representations of the symbolic values in a generating extension. Of these representations, however, we want to infer the "best" representation. This section gives a type reconstruction algorithm to infer such a representation on the basis of the following criteria:

- Provide as many *useful* (i.e., might be used for reduction) static values as possible. On the other hand, remove as many *useless* (i.e., never used for reduction) static values as possible.
- Remove as many *useless* tags (i.e., `SOME` on always present static values and `NONE` on always absent static values) as possible.
- Remove as many dynamic expressions and let-insertions as possible.

Given a source program, the type reconstruction algorithm constructs the "frame" of the type derivation tree in a standard way (similar to type inference in the simply typed λ-calculus), while assigning an annotation variable to every annotation and generating constraints for the annotation variables (in the same way as standard type-based analyses). Then, the algorithm adds $d \neq 0$ to the constraints for every d in the type of the whole program, just as the standard BTA requires the whole program to be dynamic. The constraints thus generated can straightforwardly be simplified into a conjunction (logical "and") of constraints of the following forms.

$$\begin{array}{c} d \neq 0 \\ d \neq 0 \implies s \neq \omega \\ s_1 \neq \omega \implies s_2 \neq \omega \\ s \neq \omega \implies d \neq 0 \\ s \neq \omega \wedge d_1 \neq 0 \implies d_2 \neq 0 \end{array} \qquad \begin{array}{c} s_1 \rightsquigarrow_S s_2 \\ d_1 \rightsquigarrow_D d_2 \\ d_1 \rightsquigarrow_D d_2 + d_3 \\ d_1 \rightsquigarrow_D (s, d_2) \cdot d_3 \end{array}$$

Although the constraints may seem somewhat complex, the constraint solving algorithm is not so complex. It starts by assigning $\top$ to all of the static annotation variables and ω to all of the dynamic annotation variables, [7] and refines the annotations by the iterations below. Intuitively, Step 1a removes useless `Some` tags on static values, while Step 2a removes useless `None` tags and useless static values. In addition, both Step 1b and Step 2b remove unnecessary let-insertion and dynamic expressions.

[7] Actually, the algorithm may start with all dynamic annotation variables assigned 1 rather than ω, because this also satisfies all the constraints. (Let-insertion occurs in the coercion from 1 to $1 + 1 = \omega$ by $\rightsquigarrow_D$.)

Step 1a For each $s = \top$, assign ω to s if doing so does not contradict the constraints.

Step 1b For each $d = \omega$ (respectively, $d = 1$), assign 1 (respectively, 0) to d if doing so does not contradict the constraints. Go back to Step 1a until the annotations reach a fixed point, i.e., no further changes are possible.

Step 2a For each $s \neq 0$, assign 0 to s if either $\mathit{Absent}(s)$ or $\mathit{Unused}(s)$, where the predicates *Absent* and *Unused* on static annotation variables are defined below. Repeat this until reaching a fixed point.

$\mathit{Absent}(s) \iff s = \top$ and there is no constraint of the form $\omega \rightsquigarrow_S s$ or $\top \rightsquigarrow_S s$.

Intuitively, $\mathit{Absent}(s)$ asserts that the static value is always absent during specialization, by checking (the lack of) constraints implying its presence. If this predicate holds, the `None` tag is found useless (and therefore s is assigned 0).

$\mathit{Unused}(s) \iff$ there is no constraint of the form $s \rightsquigarrow_S \omega$, $s \rightsquigarrow_S \top$, or of the form $s \neq \omega \Longrightarrow s' \neq \omega$.

Intuitively, $\mathit{Unused}(s)$ asserts that the static value is never used for reduction during specialization, by checking (the lack of) constraints implying such use, in particular, (3) and (5) in the typing rules. If this predicate holds, the static value is found useless (and therefore s is assigned 0).

Note that assigning 0 to such an s never contradicts any constraints, in particular, constraints of the form $s \neq \omega \Longrightarrow d \neq 0$ or of the form $s \neq \omega \wedge d_1 \neq 0 \Longrightarrow d_2 \neq 0$. They always appear with a constraint of the form $s \neq \omega \Longrightarrow s' \neq \omega$ in the typing rules, which prevents assigning 0 to such an s in the first place.

Step 2b For each $d = \omega$ (respectively, $d = 1$), assign 1 (resp. 0) to d if doing so does not contradict the constraints. Repeat this until reaching a fixed point.

The algorithm has two iterations on static annotations (Step 1a and Step 2a) because we have two conflicting demands on them: one is to provide as many (useful) static values as possible, and the other is to remove as many (useless) static values as possible. We satisfy the former, more important demand in the first iteration (Step 1a), and the latter, less important demand in the second iteration (Step 2a), as Asai did in his BTA (Asai, 1999).

All the steps of the algorithm monotonically decrease the annotations under the orderings $\top \succ_S \omega \succ_S 0$ and $\omega \succ_D 1 \succ_D 0$. Since there are only three possible values for each annotation variable, the decrease terminates within a linear number of iterations with respect to the number of the annotation variables. Furthermore, the decrease is locally confluent. Therefore, it is strongly normalizing, i.e., the result of the algorithm exists uniquely.

However, note that the algorithm does not give the *least* solution with respect to the orderings above, because we do not want to remove useful static values. For example, consider the partial evaluation of $\lambda x.\,\mathbf{let}\ f = \lambda y.\,y\ \mathbf{in}\ \mathbf{pair}(f, f\,x)$. It is valid *not* to provide the static value of f, but we *want* to provide it because doing so enables more reduction.

Note also that the decrease of d in Step 1b enables further decrease of s in Step 1a through constraints of the form $d \neq 0 \implies s \neq \omega$, so those steps are interleaved with each other. On the other hand, the decrease of d in Step 2b does not enable any decrease of s in Step 2a (which involves no d at all), so Step 2b is separated from Step 2a.

In the worst cases, the algorithm can take $O(n^6)$ time with respect to the size of the source program for the following reasons. Let M be the number of the constraint variables and N be the number of the constraints. Both M and N can be $O(n^2)$, because the size of the type derivation tree can be $O(n)$ and because each instance of the typing rules can generate $O(n)$ constraint variables and constraints. On the other hand, the number of the iterations can be $O(M)$, and each step of the iterations can take $O(MN)$ time, because the step must check $O(M)$ constraint variables against $O(N)$ constraints. Therefore, the algorithm can take $O(M) \times O(MN) = O(n^6)$ time in the worst case.

However, this estimate is very rough and can be refined in many ways. For example, if for each constraint variable, the number of the constraints involving the constraint variable can be bounded by a constant, then each step of the iterations takes only $O(M)$ time and therefore the time complexity of the algorithm can be bounded by $O(M) \times O(M) = O(n^4)$. Similarly, if the number of the free variables at each point of a source program can be bounded by a constant, then M and N can be bounded by $O(n)$ and therefore the time complexity of the algorithm can be bounded by $O(n^3)$. (For the same reasons, the time complexity can be bounded by $O(n^2)$ if both of these are the case.)

In practice, even if the cost of the analysis turns out to be a problem, the algorithm can be interrupted at any time because all the constraints are satisfied throughout the iterations. Although the accuracy of the analysis affects the efficiency of a generating extension, it does not affect the efficiency of the residual program.

$$[\![\frac{}{\Gamma \vdash x : \tau}]\!] \;=\; x$$

$$[\![\frac{\dfrac{\cdots}{\Gamma_0, x : \rho_1^{(s_1,d_1)} \vdash t : \rho_2^{(s_2,d_2)}}}{\Gamma \vdash \lambda x.\, t : \rho_1^{(s_1,d_1)} \to^{(s,d)} \rho_2^{(s_2,d_2)}}]\!] \;=$$

```
let val f = fn x => ↓[Γ, Γ0] ⟦ ... / Γ0, x : ρ1^(s1,d1) ⊢ t : ρ2^(s2,d2) ⟧
in ↓[(ω,1), (s,d)] (f, let z = genid ()
                       in Abs(z, ↑[d2] f(↓[(0,ω), (s1,d1)] ((), z)))
                       end)
end
```

$$[\![\frac{\dfrac{\cdots}{\Gamma_1 \vdash t_1 : \rho_1^{(s_1,d_1)}} \quad \dfrac{\cdots}{\Gamma_2 \vdash t_2 : \rho_2^{(s_2,d_2)}}}{\Gamma \vdash \mathbf{pair}(t_1, t_2) : \rho_1^{(s_1,d_1)} \times^{(s,d)} \rho_2^{(s_2,d_2)}}]\!] \;=$$

```
let val p = (↓[Γ, Γ1] ⟦ ... / Γ1 ⊢ t1 : ρ1^(s1,d1) ⟧, ↓[Γ, Γ2] ⟦ ... / Γ2 ⊢ t2 : ρ2^(s2,d2) ⟧)
in ↓[(ω,1), (s,d)] (p, Pair(↑[d1] #1(p), ↑[d2] #2(p)))
end
```

Figure 17. Translation of type derivation tree to generating extension in ML (for variables, functions, and pairs)

6.4. Generation of Generating Extension

As mentioned above, the type derivation tree is translated into a generating extension in ML. The translation rules are given in Figure 17, Figure 18, and Figure 19 using the coercion operators in Figure 20. The rules for **snd** are omitted because they are similar to the rules for **fst**.

The translation generates a generating extension optimized according to the annotations, omitting unnecessary code such as unused expressions, unused values, unnecessary tags, and unnecessary let-insertion. It translates each code fragment of annotated type τ in the type deriva-

$$[\![\frac{\dfrac{\dots}{\Gamma_1 \vdash t_1 : \rho_1^{(s_1,d_1)} \to^{(0,d)} \rho_2^{(s_2,d_2)}} \quad \dfrac{\dots}{\Gamma_2 \vdash t_2 : \rho_1^{(s_1',d_1')}}}{\Gamma \vdash t_1\, t_2 : \rho_2^{(s_2',d_2')}}]\!] =$$

$$\overset{(0,\omega)}{\underset{(s_2',d_2')}{\downarrow}} \texttt{((), slet(App(}\overset{}{\uparrow_d}\texttt{(}\overset{\Gamma}{\underset{\Gamma_1}{\downarrow}}[\![\frac{\dots}{\Gamma_1 \vdash t_1 : \rho_1^{(s_1,d_1)} \to^{(0,d)} \rho_2^{(s_2,d_2)}}]\!]\texttt{),}$$

$$\overset{}{\uparrow_{d_1'}}\texttt{(}\overset{\Gamma}{\underset{\Gamma_2}{\downarrow}}[\![\frac{\dots}{\Gamma_2 \vdash t_2 : \rho_1^{(s_1',d_1')}}]\!]\texttt{))))}$$

$$[\![\frac{\dfrac{\dots}{\Gamma_1 \vdash t_1 : \rho_1^{(s_1,d_1)} \to^{(\omega,d)} \rho_2^{(s_2,d_2)}} \quad \dfrac{\dots}{\Gamma_2 \vdash t_2 : \rho_1^{(s_1',d_1')}}}{\Gamma \vdash t_1\, t_2 : \rho_2^{(s_2',d_2')}}]\!] =$$

$$\overset{(s_2,d_2)}{\underset{(s_2',d_2')}{\downarrow}} \texttt{(\#1(}\overset{\Gamma}{\underset{\Gamma_1}{\downarrow}}[\![\frac{\dots}{\Gamma_1 \vdash t_1 : \rho_1^{(s_1,d_1)} \to^{(\omega,d)} \rho_2^{(s_2,d_2)}}]\!]\texttt{)}$$

$$\texttt{(}\overset{(s_1',d_1')}{\underset{(s_1,d_1)}{\downarrow}}\texttt{(}\overset{\Gamma}{\underset{\Gamma_2}{\downarrow}}[\![\frac{\dots}{\Gamma_2 \vdash t_2 : \rho_1^{(s_1',d_1')}}]\!]\texttt{)))}$$

$$[\![\frac{\dfrac{\dots}{\Gamma_1 \vdash t_1 : \rho_1^{(s_1,d_1)} \to^{(\top,d)} \rho_2^{(s_2,d_2)}} \quad \dfrac{\dots}{\Gamma_2 \vdash t_2 : \rho_1^{(s_1',d_1')}}}{\Gamma \vdash t_1\, t_2 : \rho_2^{(s_2',d_2')}}]\!] =$$

```
let val x = ↓Γ2 [[ ... / Γ2 ⊢ t2 : ρ1^(s1',d1') ]]
in case ↓Γ1 [[ ... / Γ1 ⊢ t1 : ρ1^(s1,d1) →^(⊤,d) ρ2^(s2,d2) ]]
     of (SOME(f), _) => ↓(s2,d2)(s2',d2') f(↓(s1',d1')(s1,d1) x)
      | f => ↓(0,ω)(s2',d2') ((), slet(App(↑d f, ↑d1' x)))
end
```

Figure 18. Translation of type derivation tree to generating extension in ML (for function applications)

$$[\![\frac{\dots}{\Gamma \vdash t : \tau_1 \times^{(0,d)} \tau_2}{\Gamma \vdash \mathbf{fst}(t) : \rho_1^{(s_1,d_1)}}]\!] = \overset{(0,1)}{\underset{(s_1,d_1)}{\downarrow}} \texttt{((), Fst(}\overset{}{\underset{d}{\uparrow}}[\![\frac{\dots}{\Gamma \vdash t : \tau_1 \times^{(0,d)} \tau_2}]\!]\texttt{))}$$

$$[\![\frac{\dots}{\Gamma \vdash t : \rho_1^{(s_1',d_1')} \times^{(\omega,d)} \tau_2}{\Gamma \vdash \mathbf{fst}(t) : \rho_1^{(s_1,d_1)}}]\!] = \overset{(s_1',d_1')}{\underset{(s_1,d_1)}{\downarrow}} \texttt{\#1(\#1}[\![\frac{\dots}{\Gamma \vdash t : \rho_1^{(s_1',d_1')} \times^{(\omega,d)} \tau_2}]\!]\texttt{)}$$

$$[\![\frac{\dots}{\Gamma \vdash t : \rho_1^{(s_1',d_1')} \times^{(\top,d)} \tau_2}{\Gamma \vdash \mathbf{fst}(t) : \rho_1^{(s_1,d_1)}}]\!] = \texttt{case } [\![\frac{\dots}{\Gamma \vdash t : \rho_1^{(s_1',d_1')} \times^{(\top,d)} \tau_2}]\!]$$
$$\texttt{of (SOME(p), \_) => } \overset{(s_1',d_1')}{\underset{(s_1,d_1)}{\downarrow}} \texttt{\#1(p)}$$
$$\texttt{| p => } \overset{(0,1)}{\underset{(s_1,d_1)}{\downarrow}} \texttt{((), Fst(}\overset{}{\underset{d_1'}{\uparrow}}\texttt{p))}$$

Figure 19. Translation of type derivation tree to generating extension in ML (for pair projections)

tion tree to the corresponding code fragment of ML type $[\![\tau]\!]$ (cf. Figure 13) in the generating extension.

The coercibility relation $\rightsquigarrow$ in the typing rules is translated into the corresponding coercion code in the generating extension. This coercion code is denoted by $\downarrow$, defined as in Figure 20. The other coercion code, denoted by $\uparrow$, extracts a dynamic expression from a symbolic value. When the symbolic value contains no dynamic expression, the code gives the dummy dynamic expression `Unit`.

To see how the translation works, let us look at the three translation rules for function applications in Figure 18. There is one case for each possible value of the static annotation on the function, i.e., whether the static value of the function is present or absent. If the annotation is 0, the function application is residualized as a dynamic expression. The dynamic function application is let-inserted (regardless of the annotation on the result) because it might cause dynamic effects. If the annotation is ω, the function application is reduced statically. If the annotation is $\top$, the tag on the static value of the function is examined, and either of those two actions is taken.

$$\overset{(s_1,d_1)}{\underset{(s_2,d_2)}{\downarrow}}(M_1,\ M_2) \ =\ (\overset{s_1}{\underset{s_2}{\downarrow}} M_1,\ \overset{d_1}{\underset{d_2}{\downarrow}} M_2)$$

$$\overset{s}{\underset{0}{\downarrow}} M_1 \ =\ \texttt{()} \qquad \overset{d}{\underset{0}{\downarrow}} M_2 \ =\ \texttt{()}$$

$$\overset{0}{\underset{\top}{\downarrow}} M_1 \ =\ \texttt{NONE} \qquad \overset{\omega}{\underset{1}{\downarrow}} M_2 \ =\ \texttt{Var}(M_2)$$

$$\overset{\omega}{\underset{\top}{\downarrow}} M_1 \ =\ \texttt{SOME}(M_1) \qquad \overset{1}{\underset{\omega}{\downarrow}} M_2 \ =\ \texttt{slet}\ M_2$$

$$\overset{s}{\underset{s}{\downarrow}} M_1 \ =\ M_1 \quad (\text{if } s \neq 0) \qquad \overset{d}{\underset{d}{\downarrow}} M_2 \ =\ M_2 \quad (\text{if } d \neq 0)$$

$$\overset{\Gamma_1}{\underset{\Gamma_2}{\downarrow}} M \ =\ (\texttt{let}\ x_1\ \texttt{=}\ \overset{(s_1,d_1)}{\underset{(s'_1,d'_1)}{\downarrow}} x_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_n\ \texttt{=}\ \overset{(s_n,d_n)}{\underset{(s'_n,d'_n)}{\downarrow}} x_n\ \texttt{in}\ M)$$

where $\Gamma_1(x_i) = \rho_i^{(s_i,d_i)}$ and $\Gamma_2(x_i) = \rho_i^{(s'_i,d'_i)}$ for each $x_i \in dom(\Gamma_2)$

$$\overset{\uparrow}{\underset{0}{}} M \ =\ \texttt{Unit} \qquad \overset{\uparrow}{\underset{1}{}} M \ =\ \texttt{\#2}(M) \qquad \overset{\uparrow}{\underset{\omega}{}} M \ =\ \texttt{Var(\#2}(M)\texttt{)}$$

Figure 20. Coercion operators for generation of generating extension

As an example of the translation, consider the source program $\lambda x.\,(\lambda y.\,y)\,x$. The analysis infers the annotation $(\omega, 0)$ for the function $\lambda y.\,y$ and $(0,1)$ for $\lambda x.\,(\lambda y.\,y)\,x$. Thus, the cogen generates the following generating extension.

```
- let val f = fn x => (#1 (fn y => y, ())) x
  in ((), let val z = genid ()
          in Abs(z, #2 (f ((), Var(z))))
          end)
  end;
val it = ((),Abs ("x1",Var "x1")) : unit * exp
```

This generating extension can be optimized as below by eliminating `()` in the symbolic values. Since many compilers would achieve similar optimization by unboxing the pairs, we do not go into the details.

```
- let val f = fn x => (fn y => y) x
  in let val z = genid ()
     in Abs(z, f (Var(z)))
     end
```

```
  end;
val it = Abs ("x1",Var "x1") : exp
```

This optimized generating extension yields the residual program $\lambda x.\, x$ (modulo α-conversion).

In general, we expect the following three properties for a generating extension obtained by the translation above.

- It is well-typed in ML and "never goes wrong," provided that the source program is well-typed in the simply typed λ-calculus.
- It does not duplicate (i.e., leave in the residual program more than once) any dynamic expression that it generates.
- It yields a residual program extensionally equivalent to the source program.

A formal proof of these properties is beyond the scope of this article.

7. Extensions and Limitations

7.1. Primitive Values and Primitive Operators

It is straightforward to incorporate primitive values (such as integers) into our method. Since a primitive value can be *lifted* for free, the corresponding dynamic expression is unnecessary whenever its static value is present. Therefore, when ρ is a primitive type, the translation in Figure 13 can be defined as

$$\begin{array}{rcl} [\![\rho^{(0,d)}]\!] & = & [\![\rho^{(0,d)}]\!]_D \\ [\![\rho^{(\omega,d)}]\!] & = & \rho \\ [\![\rho^{(\top,d)}]\!] & = & (\rho,\ [\![\rho^{(0,d)}]\!]_D)\ \texttt{two\_level\_value} \end{array}$$

where

```
datatype ('v, 'e) two_level_value =
    Static of 'v
  | Dynamic of 'e
```

The coercion operators in Figure 20 can be defined accordingly. The translation of primitive *operators* (i.e., operators on primitive values) is also simplified accordingly – for example, the addition $x+y$ in a source program is translated by the cogen into just $x + y$ in the generating extension, if the static values of x and y are always present.

7.2. Sum Types and Recursive Types

It is also straightforward to extend our analysis with sum types such as `bool` and recursive types such as `int list`. The type and type environment of a conditional expression can be approximated by the upper bound (resp. sum) of all the branches if the condition has a static value (resp. no static value). Recursive types come for free. They can naturally be introduced by the occur-check in the standard unification-based type reconstruction algorithm. For example, unifying the type variable α with the type $\alpha \to \alpha$ yields the recursive type $\mu\alpha.\,\alpha \to \alpha$. (Thus, our approach is *equi*-recursive rather than *iso*-recursive.)

7.3. Recursion and Non-Termination

Recursion is problematic both in online PE and in offline PE, because it may cause specialization to diverge. In naive online PE, unnecessary residualization makes the problem even harder. For example, consider the following source program.

```
let fun power _ 0 = 1
  | fun power b e = b * (power b (e - 1))
in power dynamic 10
end
```

In standard offline PE, the recursion is not a problem because `e` is static. In naive online PE, however, `e` can be dynamic in the (unnecessary) residualization of `power`. Thus, without some heuristics to stop the recursion, specialization does not terminate.

Our analysis alleviates this problem because it subsumes a standard monovariant BTA (Henglein, 1991). For instance, in the example above, our analysis annotates `power` and `e` with $(\omega, 0)$, and prevents the non-termination of specialization. Of course, the analysis does not solve the problem in all cases. Currently, our implementation relies on user annotations to stop specialization. It would be helpful to incorporate standard techniques such as memoization.

7.4. Effects

Our partial evaluator let-inserts every dynamic function application, so it is sound under any dynamic, monadic effect (Hatcliff and Danvy, 1996; Lawall and Thiemann, 1997). It might also be possible to statically reduce some effects by incorporating existing analyses for offline PE (e.g., Thiemann and Dussart, 1997).

7.5. Inlining and Hoisting

Since our method does not let-insert never-duplicated, effect-free expressions, it may duplicate some "trivial" computations. This would be no problem, however, because such computations can be hoisted out either by the compiler (although *ensuring* this is somewhat difficult) or by the postprocessor.

For example, the source program $\lambda p.\,\mathbf{let}\ x = \mathbf{fst}(p)\ \mathbf{in}\ \lambda y.\,x + y$ is partially evaluated to the residual program $\lambda p.\,\lambda y.\,\mathbf{fst}(p) + y$. Thus, the pair projection $\mathbf{fst}(p)$ might be computed every time the function $\lambda y.\,\mathbf{fst}(p) + y$ is applied. Because the computation has no effects, however, it can be hoisted out of λy.

7.6. Suboptimal Annotations

Our analysis may give suboptimal annotations with respect to the number of the coercions (`Var`) from an identifier (`ident`) to an expression (`exp`). For example, given $\mathbf{let}\ f = \lambda x.\,x\ \mathbf{in}\ \mathbf{pair}(f, f)$ as a source program, our cogen yields the following generating extension.

```
let val f = slet(Abs("x", Var("x")))
in Pair(Var(f), Var(f)) end
```

However, this generating extension is less efficient than the following, which duplicates the expression `Var(...)` and therefore is not well-typed in our type system.

```
let val f = Var(slet(Abs("x", Var("x"))))
in Pair(f, f) end
```

7.7. Polymorphism

It should be straightforward to extend our analysis with polymorphism over ordinary types. It might also be possible to extend the analysis with polymorphism over annotations (Henglein and Mossin, 1994). However, it remains to see whether such extension pays for its cost. Recall that the accuracy of our analysis does not affect the efficiency of the residual program, though it does affect the efficiency of a generating extension.

8. Experiments

In order to assess the effectiveness of our method, we specialized the three source programs in Table I by using several methods including

Table I. Source programs for experiments

imp	An interpreter for an imperative language. The static input is a program to compute $\sum_{i=1}^{10} i + \sum_{j=1}^{d} j$, and the dynamic input is $d = 10$.
match	A pattern matcher to test whether an integer list is a prefix of another integer list. In **match1**, the static input is the former list (= $[1, 2, 3]$) and the dynamic input is the latter list (= $[1, 2, 4, 3, 5]$). In **match2**, the static input is the latter list (= `[1,2,3]`) and the dynamic input is the former list (= `[1,2]`).
power	A standard example for PE. The static input is the exponent (= 10) and the dynamic input is the basis (= 3).

Table II. Effects of optimizations: Time for specialization of source programs (in μs)

optimization \ source program	**imp**	**match1**	**match2**	**power**
naive online PE	135000	423	465	207
+ state-based let-insertion	86600	272	304	150
+ cogen approach	75900	200	220	146
+ type-based representation analysis	1310	47.8	56.0	6.59

Table III. BTA's for experiments

No PE	Make everything dynamic. The only possible annotation is $(0, 1)$.
MonoBTA	A standard monovariant BTA (Henglein, 1991). The possible annotations are $(\omega, 0)$ and $(0, \omega)$.
PolyBTA	Apply MonoBTA after code duplication by hand to enable as much specialization as possible.
No BTA	Make everything "unknown." The only possible annotation is $(\top, \omega)$.
(Ruf, 1993)	Apply MonoBTA, and interpret "dynamic" as "unknown" (Ruf, 1993). The possible annotations are $(\omega, 0)$ and $(\top, \omega)$.
(Sperber, 1996)	Sperber's BTA (Sperber, 1996). The possible annotations are $(\omega, 0)$, $(0, \omega)$, and $(\top, \omega)$.
Ours	The analysis in Section 6. All annotations are possible.

Table IV. Comparison of BTA's: Time for specialization of source programs (in μs)

BTA \ source program	**imp**	**match1**	**match2**	**power**
MonoBTA	327	92.1	92.1	51.1
PolyBTA	350	85.0	93.9	52.3
No BTA	75900	200	220	146
(Ruf, 1993)	5180	107	124	52.1
(Sperber, 1996)	5180	109	125	52.5
Ours	1310	47.8	56.0	6.59

Table V. Comparison of BTA's: Number of let-insertion

BTA \ source program	**imp**	**match1**	**match2**	**power**
MonoBTA	54	12	12	10
PolyBTA	54	9	10	10
No BTA	7124	22	23	21
(Ruf, 1993)	622	13	14	10
(Sperber, 1996)	622	13	14	10
Ours	35	0	0	0

Table VI. Comparison of BTA's: Time for execution of residual programs (in μs)

BTA \ source program	**imp**	**match1**	**match2**	**power**
No PE	90.1	3.42	2.80	4.88
MonoBTA	33.4	3.26	2.76	0.46
PolyBTA	33.9	0.81	0.35	0.46
No BTA	22.3	0.83	0.37	0.46
(Ruf, 1993)	21.7	0.80	0.33	0.46
(Sperber, 1996)	21.2	0.80	0.34	0.45
Ours	22.9	0.82	0.35	0.45

ours, extended with primitive values, sum types, recursive types, and recursive functions as explained in Section 7. There are many techniques that are orthogonal in principle, but we selected a few combinations that are significant in practice. Although the programs and their inputs (both static and dynamic) are rather small, we believe that the results are still informative for comparing various methods of specialization. All the experiments were performed with Standard ML of New Jersey Version 110.0.3 and Linux 2.2.10 on Mobile Pentium II 400 MHz and 128 MB main memory.

First, we measured the effectiveness of the optimizations in Section 4, Section 5, and Section 6, by the efficiency of specialization. The results are shown in Table II. It is observed that the cogen approach to online PE by itself does not achieve as much speedup as Thiemann's cogen approach to offline PE does (Thiemann, 1999), because the overhead of "online-ness" are too large.

Second, we compared the effectiveness of our analysis and various BTA's in Table III by the efficiency of the generating extensions and the residual programs. For the sake of convenience, we call the first three BTA's "offline" and the last four BTA's "online". We implemented them on the basis of our analysis in Section 6 by restricting the possible annotations accordingly in each analysis. We adopted state-based let-insertion and the cogen approach in all the cases. The results are shown

in Table IV, Table V, and Table VI. The time for the analyses was minimal in all the benchmarks.

In all of the source programs, the decrease of unnecessary let-insertion was crucial to the efficiency of specialization. Especially, as compared to the other methods (except for the offline BTA's in **imp**, which were unable to perform the same specialization as our method did), our method was much more efficient because it performed far fewer let-insertions. (Recall that some instances of let-insertion are *mandatory* in order to preserve effects and avoid code duplication.)

Below is our explanation for other details of the results.

- In **imp**, the offline BTA's were unable to statically reduce $\sum_{i=1}^{10} i$ to 55. As a result, their generating extensions were faster but their residual programs were slower than those of the online BTA's.

- In **match**, MonoBTA just reconstructed the source program because it inferred both arguments as dynamic. For satisfactory specialization, a user must specify different binding-times for **match1** and **match2** and thereby *duplicate* the generating extension, as we did by hand in PolyBTA. (Although this setting itself is artificial, it gives information about the effectiveness of various BTA's for polyvariant use of a function.)

- In **power**, all the BTA's were able to perform satisfactory specialization.

As a whole, these experimental results show that our method (1) yields as fast residual programs as naive online PE and (2) enables more than twice as fast specialization as state-of-the-art (i.e., with state-based let-insertion, cogen approach, and various BTA's) offline PE when they yield similar residual programs, thanks to the removal of unnecessary let-insertion.

9. Conclusion

We presented a hybrid approach to online and offline PE that combines the advantages of both approaches. Experiments showed that our method is more than twice as fast as existing methods when they perform the same specialization, thanks to the optimizations on let-insertion.

Although we focused on PE for a statically typed call-by-value functional language, our method is applicable to PE for functional languages in general. For example, even in a pure language that has no effects

at all, state-based let-insertion would be useful to prevent code duplication, because the state monad can usually be implemented more efficiently than the continuation monad. For another example, even in a lazy language that does not evaluate unused values at all, the analysis would be useful to remove unnecessary let-insertion. Even in a dynamically typed language such as Scheme, it would be possible to adopt our analysis via the *soft typing* approach (Cartwright and Fagan, 1991), as is usual with standard type-based analyses including BTA.

Experiments with larger programs and a correctness proof of the analysis are left as future work. Although little can be said for sure without real experience, our analysis might scale (in terms of precision) to larger programs, because there is already a kind of (non-structural) subtyping polymorphism in the form of the coercion relation $\rightsquigarrow$. Of course, it would also be worth considering to incorporate parametric polymorphism and/or structural subtyping to make our analysis more precise. Further analysis of the experimental results would also be interesting.

From a broader point of view, our experience with the cogen approach to online PE suggests that Thiemann's cogen approach (Thiemann, 1999) to offline PE is useful for deriving combinators from a denotational interpreter in general. For instance, we have already applied this approach to a denotational interpreter of π-calculus (Milner, 1993) in ML and obtained "concurrency combinators" in ML. We are trying to generalize these results by using the notion of monad reflection (Filinski, 1996).

Our results on state-based let-insertion suggests that continuation-based operations can be simulated by state-based ones in certain cases. It would also be interesting to study when and how such simulation is possible.

Acknowledgements

We would like to thank many people including Kenichi Asai, Olivier Danvy, Atsushi Igarashi, Hidehiko Masuhara, the members of Prof. Yonezawa's group, and the anonymous reviewers for their advice on this work. In particular, the tremendous amount of comments by the anonymous reviewers helped to tune the presentation of this paper in great detail.

This work was partially supported by the Japan Society for the Promotion of Science.

References

Appel, A. W.: 1992, *Compiling with Continuations*. Cambridge University Press.

Appel, A. W. and Z. Shao: 1996, 'An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures'. *Journal of Functional Programming* **6**(1), 47–74.

Asai, K.: 1999, 'Binding-Time Analysis for Both Static and Dynamic Expressions'. In: *Static Analysis Symposium*, Vol. 1694 of *Lecture Notes in Computer Science*. pp. 117–133.

Ashley, J. M.: 1997, 'The effectiveness of flow analysis for inlining'. In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. pp. 99–111.

Berger, U. and H. Schwichtenberg: 1991, 'An Inverse of the Evaluation Functional for Typed Lambda-Calculus'. In: *Sixth Annual IEEE Symposium on Logic in Computer Science*. pp. 203–211.

Birkedal, L. and R. Harper: 1999, 'Relational Interpretations of Recursive Types in an Operational Setting'. *Information and Computation* **155**(1–2), 3–63.

Birkedal, L. and M. Welinder: 1993, 'Partial Evaluation of Standard ML'. Master's thesis, Department of Computer Science, University of Copenhagen. `http://www.cs.cmu.edu/afs/cs/user/birkedal/pub/smlmix.ps.gz`.

Bondorf, A.: 1990, 'Self-Applicable Partial Evaluation'. Technical Report 90/17, Department of Computer Science, University of Copenhagen. Ph.D. Thesis (Revised Version).

Bondorf, A.: 1992, 'Improving binding times without explicit CPS-conversion'. In: *Proceedings of the Conference on LISP and Functional Programming*. pp. 1–10.

Bondorf, A. and O. Danvy: 1991, 'Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types'. *Science of Computer Programming* **16**(2), 151–195.

Cartwright, R. and M. Fagan: 1991, 'Soft Typing'. In: *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. pp. 278–292. In *ACM SIGPLAN Notices*, 26(6), June 1991.

Consel, C.: 1993, 'Polyvariant binding-time analysis for applicative languages'. In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. pp. 66–77.

Consel, C. and O. Danvy: 1993, 'Tutorial notes on partial evaluation'. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 493–501.

Danvy, O.: 1996, 'Type-Directed Partial Evaluation'. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 242–257.

Danvy, O.: 1997, 'Online Type-Directed Partial Evaluation'. Technical Report RS-97-53, Basic Research in Computer Science. `http://www.brics.dk/RS/97/53/`. Extended version of an article that appeared in *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, Masahiko Sato and Yoshihito Toyama, editors, World Scientific, 1998.

Danvy, O.: 1999, 'Type-Directed Partial Evaluation'. In: *Partial Evaluation – Practice and Theory*, Vol. 1706 of *Lecture Notes in Computer Science*. pp. 367–411.

Danvy, O. and A. Filinski: 1989, 'A Functional Abstraction of Typed Contexts'. Technical Report 89/12, Institute of Datalogy, University of Copenhagen.

Danvy, O. and A. Filinski: 1990, 'Abstracting Control'. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. pp. 151–160.

Filinski, A.: 1994, 'Representing Monads'. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 446–457.

Filinski, A.: 1996, 'Controlling Effects'. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. `http://www.brics.dk/~andrzej/papers/CE.ps.gz`.

Filinski, A.: 2001, 'Normalization by Evaluation for the Computational Lambda-Calculus'. In: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, Vol. 2044 of *Lecture Notes in Computer Science*. pp. 151–165.

Futamura, Y.: 1971, 'Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler'. *Systems, Computers, Controls* **2**(5), 45–50. Reprinted in *Higher-Order and Symbolic Computation*, Vol. 12, No. 4, Pages 381–391, 1999.

Hatcliff, J. and O. Danvy: 1996, 'A Computational Formalization for Partial Evaluation (Extended Version)'. Technical Report RS-96-34, Basic Research in Computer Science. `http://www.brics.dk/RS/96/34/`. Extended version of an article that appeared in Mathematical Structures in Computer Science, Vol. 7, No. 5, Pages 507–541, October 1997.

Henglein, F.: 1991, 'Efficient Type Inference for Higher-Order Binding-Time Analysis'. In: *Proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture*, Vol. 523 of *Lecture Notes in Computer Science*. pp. 448–472.

Henglein, F. and C. Mossin: 1994, 'Polymorphic Binding-Time Analysis'. In: *5th European Symposium on Programming*, Vol. 788 of *Lecture Notes in Computer Science*. pp. 287–301.

Jones, N. D., C. K. Gomard, A. Bondorf, O. Danvy, and T. Æ. Mogensen: 1990, 'A Self-applicable Partial Evaluator for the Lambda Calculus'. In: *1990 International Conference on Computer Languages*. pp. 49–58.

Jones, N. D., C. K. Gomard, and P. Sestoft: 1993, *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

Kobayashi, N.: 1999, 'Type-Based Useless Variable Elimination'. Technical report, Department of Information Science, Faculty of Science, University of Tokyo. `http://www.yl.is.s.u-tokyo.ac.jp/members/koba/pub/UVE-full.ps.gz`. A summary appeared in *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*.

Launchbury, J.: 1991, 'A Strongly-Typed Self-Applicable Partial Evaluator'. In: *Proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture*. pp. 145–164.

Lawall, J. and P. Thiemann: 1997, 'Sound Specialization in the Presence of Computational Effects'. In: *Theoretical Aspects of Computer Software*, Vol. 1281 of *Lecture Notes in Computer Science*. pp. 165–190.

Lawall, J. L. and O. Danvy: 1994, 'Continuation-Based Partial Evaluation'. In: *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, Vol. VII of *ACM SIGPLAN Lisp Pointers*. pp. 227–238.

Milner, R.: 1993, 'The Polyadic π-Calculus: A Tutorial'. In: F. L. Bauer, W. Brauer, and H. Schwichtenberg (eds.): *Logic and Algebra of Specification*. Springer-Verlag. `http://www.dcs.ed.ac.uk/home/lfcsreps/EXPORT/91/ECS-LFCS-91-180/`.

Moggi, E.: 1991, 'Notions of Computation and Monads'. *Information and Computation* **93**(1), 55–92.

Pfenning, F. and C. Elliott: 1988, 'Higher-order Abstract Syntax'. In: *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*. pp. 199–208.

Ruf, E.: 1993, 'Topics in Online Partial Evaluation'. Ph.D. thesis, Stanford University. `http://research.microsoft.com/~daniel/fuse-memos/FUSE-MEMO-93-14.ps`.

Sperber, M.: 1996, 'Self-Applicable Online Partial Evaluation'. In: *Partial Evaluation*, Vol. 1110 of *Lecture Notes in Computer Science*. pp. 465–480.

Sumii, E. and N. Kobayashi: 2000, 'Online Type-Directed Partial Evaluation for Dynamically-Typed Languages'. *Computer Software* **17**(3), 38–62.

Talpin, J.-P. and P. Jouvelot: 1994, 'The Type and Effect Discipline'. *Information and Computation* **111**(2), 245–296.

Thiemann, P.: 1999, 'Combinators for Program Generation'. *Journal of Functional Programming* **9**(5), 483–525.

Thiemann, P. and D. Dussart: 1997, 'Partial Evaluation for Higher-Order Languages with State'. `http://www.informatik.uni-freiburg.de/~thiemann/papers/mlpe.ps.gz`.

Turner, D. N., P. Wadler, and C. Mossin: 1995, 'Once upon a Type'. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. pp. 1–11.

Wadler, P.: 1990, 'Deforestation: Transforming Programs to Eliminate Trees'. *Theoretical Computer Science* **73**(2), 231–248.

Yang, Z.: 1998, 'Encoding types in ML-like languages'. In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. pp. 289–300.

Appendix

In the proofs below, we use the following two lemmas on $\circ$.

Lemma. $(id \circ e) \approx (e \circ id) \approx e$ for any e.

Proof.

$$
\begin{aligned}
& id \circ e \\
& \text{(* Definition of } id \text{ and } \circ \text{ *)} \\
=\ & \mathbf{let}\ f = \lambda z.z\ \mathbf{in}\ \mathbf{let}\ g = e\ \mathbf{in}\ \lambda x.f(gx) \\
& \text{(* } \beta \text{ *)} \\
\approx\ & \mathbf{let}\ g = e\ \mathbf{in}\ \lambda x.(\lambda z.z)(gx) \\
& \text{(* } \lambda \text{ *)} \\
\approx\ & \mathbf{let}\ g = e\ \mathbf{in}\ \lambda x.\mathbf{let}\ z = gx\ \mathbf{in}\ z \\
& \text{(* unit *)}
\end{aligned}
$$

$\approx$ **let** $g = e$ **in** $\lambda x.gx$
(* η *)
$\approx$ **let** $g = e$ **in** g
(* unit *)
$\approx$ e

$e \circ id$
(* Definition of id and $\circ$ *)
$=$ **let** $f = e$ **in let** $g = \lambda z.z$ **in** $\lambda x.f(gx)$
(* β *)
$\approx$ **let** $f = e$ **in** $\lambda x.f((\lambda z.z)x)$
(* λ *)
$\approx$ **let** $f = e$ **in** $\lambda x.f($**let** $z = x$ **in** $z)$
(* unit *)
$\approx$ **let** $f = e$ **in** $\lambda x.fx$
(* η *)
$\approx$ **let** $f = e$ **in** f
(* unit *)
$\approx$ e

□

Lemma. $e_1 \circ (e_2 \circ e_3) \approx (e_1 \circ e_2) \circ e_3$ for any e_1, e_2, and e_3. Thanks to this lemma, it does not matter whether $\circ$ associates to the left or to the right.

Proof.

$e_1 \circ (e_2 \circ e_3)$
(* Definition of $\circ$ *)
$=$ **let** $f_1 = e_1$ **in let** $g =$ (**let** $f_2 = e_2$ **in let** $f_3 = e_3$ **in** $\lambda y.f_2(f_3 y)$) **in**
$\lambda x.f_1(gx)$
(* assoc *)
$\approx$ **let** $f_1 = e_1$ **in let** $f_2 = e_2$ **in let** $f_3 = e_3$ **in let** $g = \lambda y.f_2(f_3 y)$ **in**
$\lambda x.f_1(gx)$
(* β *)
$\approx$ **let** $f_1 = e_1$ **in let** $f_2 = e_2$ **in let** $f_3 = e_3$ **in**

$$
\begin{array}{ll}
 & \lambda x.f_1((\lambda y.f_2(f_3 y))x) \\
 & (*\ \lambda\ *) \\
\approx & \textbf{let } f_1 = e_1 \textbf{ in let } f_2 = e_2 \textbf{ in let } f_3 = e_3 \textbf{ in} \\
 & \lambda x.f_1(\textbf{let } y = x \textbf{ in } (f_2(f_3 y))) \\
 & (*\ \beta\ *) \\
\approx & \textbf{let } f_1 = e_1 \textbf{ in let } f_2 = e_2 \textbf{ in let } f_3 = e_3 \textbf{ in} \\
 & \lambda x.f_1(f_2(f_3 x)) \\
 & (*\ \text{app}\ *) \\
\approx & \textbf{let } f_1 = e_1 \textbf{ in let } f_2 = e_2 \textbf{ in let } f_3 = e_3 \textbf{ in} \\
 & \lambda x.\textbf{let } f_1' = f_1 \textbf{ in let } f_2' = f_2 \textbf{ in let } z = f_3 x \textbf{ in } f_1'(f_2' z) \\
 & (*\ \beta\ *) \\
\approx & \textbf{let } f_1 = e_1 \textbf{ in let } f_2 = e_2 \textbf{ in let } f_3 = e_3 \textbf{ in} \\
 & \lambda x.\textbf{let } z = f_3 x \textbf{ in } f_1(f_2 z) \\
 & (*\ \lambda\ *) \\
\approx & \textbf{let } f_1 = e_1 \textbf{ in let } f_2 = e_2 \textbf{ in let } f_3 = e_3 \textbf{ in} \\
 & \lambda x.(\lambda z.f_1(f_2 z))(f_3 x) \\
 & (*\ \beta\ *) \\
\approx & \textbf{let } f_1 = e_1 \textbf{ in let } f_2 = e_2 \textbf{ in let } g = \lambda z.f_1(f_2 z) \textbf{ in let } f_3 = e_3 \textbf{ in} \\
 & \lambda x.g(f_3 x) \\
 & (*\ \text{assoc}\ *) \\
\approx & \textbf{let } g = (\textbf{let } f_1 = e_1 \textbf{ in let } f_2 = e_2 \textbf{ in } \lambda z.f_1(f_2 z)) \textbf{ in let } f_3 = e_3 \textbf{ in} \\
 & \lambda x.g(f_3 x) \\
 & (*\ \text{Definition of } \circ\ *) \\
= & (e_1 \circ e_2) \circ e_3
\end{array}
$$

□

Proof of Theorem 4.2

By structural induction on the abstract syntax of e. Because it is too lengthy to write down all the equations, we only outline them.

If $e = x$:

$$
\begin{array}{rl}
 & [w^c/x][\![x]\!]_c \\
= & \lambda k.kw^c \\
\approx & \lambda k.\textbf{let } z = id \textbf{ in } (z \circ k)w^c \\
\sim^{comp} & \lambda\sigma.\textbf{let } z = id \textbf{ in } \langle w^s, \sigma \circ z\rangle
\end{array}
$$

$$\begin{array}{lll} & \approx & \lambda\sigma.\langle w^s, \sigma\rangle \\ & = & [w^s/x][\![x]\!]_s \end{array}$$

If $e = \lambda x.e_0$: By the induction hypothesis,

$$[w^c/x, \tilde{w}^c/\tilde{z}][\![e_0]\!]_c \ \sim_\tau^{comp} \ [w^s/x, \tilde{w}^s/\tilde{z}][\![e_0]\!]_s$$

for any $w^c \sim^{val} w^s$. Therefore, by the definition of $\sim^{val}$ for function types,

$$[\tilde{w}^c/\tilde{z}](\lambda x.[\![e_0]\!]_c) \ \sim^{val} \ [\tilde{w}^s/\tilde{z}](\lambda x.[\![e_0]\!]_s) \tag{1}$$

Hence:

$$\begin{array}{lll} & & [\tilde{w}^c/\tilde{z}][\![\lambda x.e_0]\!]_c \\ & = & \lambda k.k[\tilde{w}^c/\tilde{z}](\lambda x.[\![e_0]\!]_c) \\ & \approx & \lambda k.\mathbf{let}\ z = id\ \mathbf{in}\ (z \circ k)[\tilde{w}^c/\tilde{z}](\lambda x.[\![e_0]\!]_c) \\ & & \quad (*\ \text{using (1)}\ *) \\ & \sim^{comp} & \lambda\sigma.\mathbf{let}\ z = id\ \mathbf{in}\ \langle[\tilde{w}^s/\tilde{z}](\lambda x.[\![e_0]\!]_s), \sigma \circ z\rangle \\ & \approx & \lambda\sigma.\langle[\tilde{w}^s/\tilde{z}](\lambda x.[\![e_0]\!]_s), \sigma\rangle \\ & = & [\tilde{w}^s/\tilde{z}][\![\lambda x.e_0]\!]_s \end{array}$$

If $e = e_1 e_2$: By the induction hypothesis, for each $j = 1, 2$,

$$[\tilde{w}^c/\tilde{z}][\![e_j]\!]_c(\lambda x_j.\ldots) \ \approx \ \mathbf{let}\ \tilde{x}_j = \tilde{e}_j^k\ \mathbf{in}\ (x_{j_i} \circ (\lambda x_j.\ldots))[v_j^c] \tag{2}$$

and

$$[\tilde{w}^s/\tilde{z}][\![e_j]\!]_s\sigma \ \approx \ \mathbf{let}\ \tilde{x}_j = \tilde{e}_j^\sigma\ \mathbf{in}\ \langle v_j^s, \sigma \circ x_{j_i}\rangle \tag{3}$$

for some $\tilde{e}_j^k \approx \tilde{e}_j^\sigma$ and $v_j^c \sim^{val} v_j^s$. Since e_1 has a function type,

$$v_1^c v_2^c k \ \approx \ \mathbf{let}\ \tilde{y} = \tilde{f}^k\ \mathbf{in}\ (y_i \circ k)[v^c] \tag{4}$$

and

$$v_1^s v_2^s(\sigma \circ \cdots) \ \approx \ \mathbf{let}\ \tilde{y} = \tilde{f}^\sigma\ \mathbf{in}\ \langle v^s, \sigma \circ \cdots \circ y_i\rangle \tag{5}$$

for some $\tilde{f}^k \approx \tilde{f}^\sigma$ and $v^c \sim^{val} v^s$. Hence:

$$\begin{array}{lll} & & [\tilde{w}^c/\tilde{z}][\![e_1 e_2]\!]_c \\ & = & \lambda k.[\tilde{w}^c/\tilde{z}][\![e_1]\!]_c(\lambda x_1.[\tilde{w}^c/\tilde{z}][\![e_2]\!]_c(\lambda x_2.x_1 x_2 k)) \\ & & \quad (*\ \text{using (2)}\ *) \\ & \approx & \lambda k.\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^k\ \mathbf{in}\ (x_{1_i} \circ (\lambda x_1. \\ & & \qquad \mathbf{let}\ \tilde{x}_2 = \tilde{e}_2^k\ \mathbf{in}\ (x_{2_i} \circ (\lambda x_2.x_1 x_2 k))[v_2^c]))[v_1^c] \end{array}$$

$$
\begin{array}{ll}
\approx & \lambda k.\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^k\ \mathbf{in} \\
& \quad \mathbf{let}\ \tilde{x}_2 = \tilde{e}_2^k\ \mathbf{in}\ (x_{1i} \circ x_{2i})[v_1^c v_2^c k] \\
& \text{(* using (4) *)} \\
\approx & \lambda k.\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^k\ \mathbf{in} \\
& \quad \mathbf{let}\ \tilde{x}_2 = \tilde{e}_2^k\ \mathbf{in\ let}\ \tilde{y} = \tilde{f}^k\ \mathbf{in}\ (x_{1i} \circ x_{2i} \circ y_i \circ k)[v^c] \\
\sim^{comp} & \lambda\sigma.\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^\sigma\ \mathbf{in} \\
& \quad \mathbf{let}\ \tilde{x}_2 = \tilde{e}_2^\sigma\ \mathbf{in\ let}\ \tilde{y} = \tilde{f}^\sigma\ \mathbf{in}\ \langle v^s, \sigma \circ x_{1i} \circ x_{2i} \circ y_i \rangle \\
& \text{(* using (5) *)} \\
\approx & \lambda\sigma.\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^\sigma\ \mathbf{in} \\
& \quad \mathbf{let}\ \tilde{x}_2 = \tilde{e}_2^\sigma\ \mathbf{in}\ v_1^s v_2^s(\sigma \circ x_{1i} \circ x_{2i}) \\
\approx & \lambda\sigma.\mathbf{let}\ \langle x_1, \sigma_1 \rangle = (\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^\sigma\ \mathbf{in}\ \langle v_1^s, \sigma \circ x_{1i} \rangle)\ \mathbf{in} \\
& \quad \mathbf{let}\ \langle x_2, \sigma_2 \rangle = (\mathbf{let}\ \tilde{x}_2 = \tilde{e}_2^\sigma\ \mathbf{in}\ \langle v_2^s, \sigma_1 \circ x_{2i} \rangle)\ \mathbf{in}\ x_1 x_2 \sigma_2 \\
& \text{(* using (3) *)} \\
\approx & \lambda\sigma.\mathbf{let}\ \langle x_1, \sigma_1 \rangle = [\tilde{w}^s/\tilde{z}][\![e_1]\!]_s\sigma\ \mathbf{in} \\
& \quad \mathbf{let}\ \langle x_2, \sigma_2 \rangle = [\tilde{w}^s/\tilde{z}][\![e_2]\!]_s\sigma_1\ \mathbf{in}\ x_1 x_2 \sigma_2 \\
= & [\tilde{w}^s/\tilde{z}][\![e_1 e_2]\!]_s
\end{array}
$$

If $e = \delta_i(e_1, \ldots, e_n)$: By the induction hypothesis, for each $j = 1, \ldots, n$,

$$[\tilde{w}^k/\tilde{z}][\![e_j]\!]_c(\lambda x_j. \ldots) \approx \mathbf{let}\ \tilde{x}_j = \tilde{e}_j^k\ \mathbf{in}\ (x_{j_i} \circ (\lambda x_j. \ldots))[v_j^c] \quad (6)$$

and

$$[\tilde{w}^s/\tilde{z}][\![e_j]\!]_s\sigma \approx \mathbf{let}\ \tilde{x}_j = \tilde{e}_j^\sigma\ \mathbf{in}\ \langle v_j^s, \sigma \circ x_{j_i} \rangle \quad (7)$$

for some $\tilde{e}_j^k \approx \tilde{e}_j^\sigma$ and $v_j^c \sim^{val} v_j^s$. Hence:

$$
\begin{array}{ll}
& [\tilde{w}^c/\tilde{z}][\![\delta_i(e_1, \ldots, e_n)]\!]_c \\
= & \lambda k.[\tilde{w}^c/\tilde{z}][\![e_1]\!]_c(\lambda x_1. \ldots [\tilde{w}^c/\tilde{z}][\![e_n]\!]_c(\lambda x_n.k[\delta_i(x_1, \ldots, x_n)])) \\
& \text{(* using (6) *)} \\
\approx & \lambda k.\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^k\ \mathbf{in}\ (x_{1i} \circ (\lambda x_1. \\
& \quad \ldots \\
& \quad \mathbf{let}\ \tilde{x}_n = \tilde{e}_n^k\ \mathbf{in}\ (x_{ni} \circ (\lambda x_n.k[\delta_i(x_1, \ldots, x_n)]))[v_n^c] \ldots))[v_1^c] \\
\approx & \lambda k.\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^k\ \mathbf{in} \\
& \quad \ldots \\
& \quad \mathbf{let}\ \tilde{x}_n = \tilde{e}_n^k\ \mathbf{in}\ (x_{1i} \circ \cdots \circ x_{ni} \circ k)[\delta_i(v_1^c, \ldots, v_n^c)] \\
\approx & \lambda k.\mathbf{let}\ \tilde{x}_1 = \tilde{e}_1^k\ \mathbf{in} \\
& \quad \ldots
\end{array}
$$

$$
\begin{array}{rl}
 & \quad \textbf{let } \tilde{x}_n = \tilde{e}_n^k \textbf{ in let } x = \delta_i(v_1^c, \ldots, v_n^c) \textbf{ in } (x_{1i} \circ \cdots \circ x_{ni} \circ k)[x] \\
\sim^{comp} & \lambda\sigma.\textbf{let } \tilde{x}_1 = \tilde{e}_1^\sigma \textbf{ in} \\
 & \quad \ldots \\
 & \quad \textbf{let } \tilde{x}_n = \tilde{e}_n^\sigma \textbf{ in let } x = \delta_i(v_1^s, \ldots, v_n^s) \textbf{ in } \langle x, \sigma \circ x_{1i} \circ \cdots \circ x_{ni} \rangle \\
\approx & \lambda\sigma.\textbf{let } \tilde{x}_1 = \tilde{e}_1^\sigma \textbf{ in} \\
 & \quad \ldots \\
 & \quad \textbf{let } \tilde{x}_n = \tilde{e}_n^\sigma \textbf{ in } \langle \delta_i(v_1^s, \ldots, v_n^s), \sigma \circ x_{1i} \circ \cdots \circ x_{ni} \rangle \\
\approx & \lambda\sigma.\textbf{let } \langle x_1, \sigma_1 \rangle = (\textbf{let } \tilde{x}_1 = \tilde{e}_1^\sigma \textbf{ in } \langle v_1^s, \sigma \circ x_{1i} \rangle) \textbf{ in} \\
 & \quad \ldots \\
 & \quad \textbf{let } \langle x_n, \sigma_n \rangle = (\textbf{let } \tilde{x}_n = \tilde{e}_n^\sigma \textbf{ in } \langle v_n^s, \sigma_{n-1} \circ x_{ni} \rangle) \textbf{ in } \langle \delta_i(x_1, \ldots, x_n), \sigma_n \rangle \\
 & \quad (* \text{ using (7) } *) \\
\approx & \lambda\sigma.\textbf{let } \langle x_1, \sigma_1 \rangle = [\tilde{w}^s/\tilde{z}][\![e_1]\!]_s\sigma \textbf{ in} \\
 & \quad \ldots \\
 & \quad \textbf{let } \langle x_n, \sigma_n \rangle = [\tilde{w}^s/\tilde{z}][\![e_n]\!]_s\sigma_{n-1} \textbf{ in } \langle \delta_i(x_1, \ldots, x_n), \sigma_n \rangle \\
= & [\tilde{w}^s/\tilde{z}][\![\delta_i(e_1, \ldots, e_n)]\!]_s
\end{array}
$$

If $e = slet(e_0)$: By the induction hypothesis,

$$[\tilde{w}^c/\tilde{z}][\![e_0]\!]_c(\lambda x.\ldots) \;\approx\; \textbf{let } \tilde{x} = \tilde{e}^k \textbf{ in } (x_i \circ (\lambda x.\ldots))[v^c] \qquad (8)$$

and

$$[\tilde{w}^s/\tilde{z}][\![e_0]\!]_s\sigma \;\approx\; \textbf{let } \tilde{x} = \tilde{e}^\sigma \textbf{ in } \langle v^s, \sigma \circ x_i \rangle \qquad (9)$$

for some $\tilde{e}^k \approx \tilde{e}^\sigma$ and $v^c \sim^{val} v^s$. Hence:

$$
\begin{array}{rl}
 & [\tilde{w}^c/\tilde{z}][\![slet(e_0)]\!]_c \\
= & \lambda k.[\tilde{w}^c/\tilde{z}][\![e]\!]_c(\lambda x.\textbf{let } y = \texttt{genid}() \textbf{ in } \texttt{Let}(y, x, k[y])) \\
 & (* \text{ using (8) } *) \\
\approx & \lambda k.\textbf{let } \tilde{x} = \tilde{e}^k \textbf{ in} \\
 & \quad (x_i \circ (\lambda x.\textbf{let } y = \texttt{genid}() \textbf{ in } \texttt{Let}(y, x, k[y])))[v^c] \\
\approx & \lambda k.\textbf{let } \tilde{x} = \tilde{e}^k \textbf{ in} \\
 & \quad x_i[\textbf{let } y = \texttt{genid}() \textbf{ in } \texttt{Let}(y, v^c, k[y])] \\
\approx & \lambda k.\textbf{let } \tilde{x} = \tilde{e}^k \textbf{ in} \\
 & \quad \textbf{let } y = \texttt{genid}() \textbf{ in } x_i[\texttt{Let}(y, v^c, k[y])] \\
\approx & \lambda k.\textbf{let } \tilde{x} = \tilde{e}^k \textbf{ in} \\
 & \quad \textbf{let } y = \texttt{genid}() \textbf{ in } (x_i \circ (\lambda x'.\texttt{Let}(y, v^c, x')) \circ k)[y] \\
\sim^{comp} & \lambda\sigma.\textbf{let } \tilde{x} = \tilde{e}^\sigma \textbf{ in} \\
 & \quad \textbf{let } y = \texttt{genid}() \textbf{ in } \langle y, \sigma \circ x_i \circ (\lambda x'.\texttt{Let}(y, v^s, x')) \rangle
\end{array}
$$

$$
\begin{array}{ll}
\approx & \lambda\sigma.\mathbf{let}\ \langle x,\sigma'\rangle = (\mathbf{let}\ \tilde{x} = \tilde{e}^{\sigma}\ \mathbf{in}\ \langle v^s, \sigma \circ x_i\rangle)\ \mathbf{in} \\
& \quad \mathbf{let}\ y = \mathtt{genid}()\ \mathbf{in}\ \langle y, \sigma' \circ (\lambda x'.\mathtt{Let}(y,x,x'))\rangle \\
& \text{(* using (9) *)} \\
\approx & \lambda\sigma.\mathbf{let}\ \langle x,\sigma'\rangle = [\tilde{w}^s/\tilde{z}][\![e_0]\!]_s\sigma\ \mathbf{in} \\
& \quad \mathbf{let}\ y = \mathtt{genid}()\ \mathbf{in}\ \langle y, \sigma' \circ (\lambda x'.\mathtt{Let}(y,x,x'))\rangle \\
= & [\tilde{w}^s/\tilde{z}][\![slet(e_0)]\!]_s
\end{array}
$$

If $e = rlet(\lambda\_.e_0)$: By the induction hypothesis,

$$[\tilde{w}^c/\tilde{z}][\![e_0]\!]_c id \ \approx\ \mathbf{let}\ \tilde{x} = \tilde{e}^k\ \mathbf{in}\ x_i[v^c] \tag{10}$$

and

$$[\tilde{w}^k/\tilde{z}][\![e_0]\!]_s id \ \approx\ \mathbf{let}\ \tilde{x} = \tilde{e}^{\sigma}\ \mathbf{in}\ \langle v^s, x_i\rangle \tag{11}$$

for some $\tilde{e}^k \approx \tilde{e}^{\sigma}$ and $v^c \sim^{val} v^s$. Hence:

$$
\begin{array}{ll}
& [\tilde{w}^c/\tilde{z}][\![rlet(\lambda\_.e_0)]\!]_c \\
= & \lambda k.k[[\tilde{w}^c/\tilde{z}][\![e_0]\!]_c id] \\
& \text{(* using (10) *)} \\
\approx & \lambda k.k[\mathbf{let}\ \tilde{x} = \tilde{e}^k\ \mathbf{in}\ x_i[v^c]] \\
\approx & \lambda k.\mathbf{let}\ \tilde{x} = \tilde{e}^k\ \mathbf{in}\ k[x_i[v^c]] \\
\approx & \lambda k.\mathbf{let}\ \tilde{x} = \tilde{e}^k\ \mathbf{in}\ \mathbf{let}\ y = x_i v^c\ \mathbf{in}\ \mathbf{let}\ z = id\ \mathbf{in}\ (z \circ k)y \\
\sim^{comp} & \lambda\sigma.\mathbf{let}\ \tilde{x} = \tilde{e}^{\sigma}\ \mathbf{in}\ \mathbf{let}\ y = x_i v^s\ \mathbf{in}\ \mathbf{let}\ z = id\ \mathbf{in}\ \langle y, \sigma \circ z\rangle \\
\approx & \lambda\sigma.\mathbf{let}\ \tilde{x} = \tilde{e}^{\sigma}\ \mathbf{in}\ \langle x_i v^s, \sigma\rangle \\
\approx & \lambda\sigma.\mathbf{let}\ \langle x,\sigma'\rangle = (\mathbf{let}\ \tilde{x} = \tilde{e}^{\sigma}\ \mathbf{in}\ \langle v^s, x_i\rangle)\ \mathbf{in}\ \langle \sigma' x, \sigma\rangle \\
& \text{(* using (11) *)} \\
\approx & \lambda\sigma.\mathbf{let}\ \langle x,\sigma'\rangle = [\tilde{w}^s/\tilde{z}][\![e_0]\!]_s id\ \mathbf{in}\ \langle \sigma' x, \sigma\rangle \\
= & [\tilde{w}^s/\tilde{z}][\![rlet(\lambda\_.e_0)]\!]_s
\end{array}
$$

□

Proof of Corollary 4.3

By Theorem 4.2, $[\![e]\!]_c \sim^{comp} [\![e]\!]_s$. Therefore,

$$[\![e]\!]_c id \ \approx\ \mathbf{let}\ \tilde{x} = \tilde{e}^k\ \mathbf{in}\ x_i[v^c] \tag{12}$$

and

$$[\![e]\!]_s id \ \approx\ \mathbf{let}\ \tilde{x} = \tilde{e}^{\sigma}\ \mathbf{in}\ \langle v^s, x_i\rangle \tag{13}$$

for some $\tilde{e}^k \approx \tilde{e}^\sigma$ and $v^c \sim^{val}_{\mathtt{exp}} v^s$. By the definition of $\sim^{val}$ for base types, $v^c \sim^{val}_{\mathtt{exp}} v^s$ implies $v^c \approx v^s$. Hence:

$$
\begin{array}{ll}
 & [\![rlet(\lambda\_.e)]\!]_c id \\
= & (\lambda k.k[[\![e]\!]_c id])id \\
\approx & [\![e]\!]_c id \\
 & \text{(* using (12) *)} \\
\approx & \textbf{let } \tilde{x} = \tilde{e}^k \textbf{ in } x_i[v^c] \\
\approx & \textbf{let } \tilde{x} = \tilde{e}^\sigma \textbf{ in } x_i v^s \\
\approx & \textbf{let } \langle x, \_\rangle = (\textbf{let } \tilde{x} = \tilde{e}^\sigma \textbf{ in } \langle x_i v^s, v^\sigma\rangle) \textbf{ in } x \\
\approx & \textbf{let } \langle x, \_\rangle = (\textbf{let } \langle x, \sigma'\rangle = (\textbf{let } \tilde{x} = \tilde{e}^\sigma \textbf{ in } \langle v^s, x_i\rangle) \textbf{ in } \langle \sigma' x, v^\sigma\rangle) \textbf{ in } x \\
 & \text{(* using (13) *)} \\
\approx & \textbf{let } \langle x, \_\rangle = (\textbf{let } \langle x, \sigma'\rangle = [\![e]\!]_s id \textbf{ in } \langle \sigma' x, v^\sigma\rangle) \textbf{ in } x \\
\approx & \textbf{let } \langle x, \_\rangle = (\lambda\sigma.\textbf{let } \langle x, \sigma'\rangle = [\![e]\!]_s id \textbf{ in } \langle \sigma' x, \sigma\rangle)v^\sigma \textbf{ in } x \\
= & \textbf{let } \langle x, \_\rangle = [\![rlet(\lambda\_.e)]\!]_s v^\sigma \textbf{ in } x
\end{array}
$$

□