Continuation-based Partial Evaluation Without Continuations

Peter Thiemann
thiemann@informatik.uni-freiburg.de

Universität Freiburg, Germany

Abstract. Continuation-based partial evaluation is an implementation technique for program specializers that allows generated code contexts to commute with specialization-time values. Thus it enables additional specialization-time computations. As the name suggests, a popular implementation is based on the manipulation of continuations.
We show that a substitution monad provides an alternative implementation of the context manipulation performed by continuation-based partial evaluation. We prove its equivalence to the original continuation-based implementation. Our new formalization allows for a more precise binding-time analysis that is based on a type and effect system. We prove the soundness of the type and effect system with respect to our implementation and point out how a binding-time analysis based on this system may avoid code duplication and enable the generation of more efficient generating extensions.

Key Words: functional programming, partial evaluation, type systems

1 Introduction

Partial evaluation [4,15] is an automatic program transformation that specializes a program with respect to known input data. Of particular interest is offline-partial evaluation, where each part of the program is annotated as either executable at specialization time or as generating specialized code. For example, in the program

$$\texttt{power}\ x\ n = \texttt{if}^s\ n ==^s 0\ \texttt{then}\ (\texttt{lift}\ 1)\ \texttt{else}\ x *^d \texttt{power}\ x\ (n -^s 1)$$

the superscript s (*static*) indicates a specialization-time expression, whereas the superscript d (*dynamic*) indicates a run-time expression. The function `lift` converts the internal representation of a number into a constant expression (a syntax tree). In the example, running the program `power x` 2 yields the syntax tree

```
x * (x * 1)
```

The annotations may be added to the program manually or by means of a program analysis (binding-time analysis). In either case, a type system can be used to specify when an annotation is sound.

The binding-time analysis is an asset and a drawback at the same time. It is an asset because it clearly indicates which expressions are processed at specialization time (and which are not). It is also a drawback because some programs require non-trivial transformation (*binding-time improvements*) before they specialize well. Hence, some research has been geared towards automatizing these transformations and including them into partial evaluation systems [1, 7, 8, 16, 20].

The present work concerns continuation-based partial evaluation, which incorporates one of the most essential binding-time improvements for call-by-value functional programming languages. It was pioneered by Bondorf, Consel, and Danvy [1, 3] and later refined and explained by Danvy, Hatcliff, Lawall, and the present author [13, 16, 17]. The basic idea is that code may be generated interleaved with specialization-time computations. For example

$$\texttt{lift}\ (17 +^s \texttt{let}^d\ x\ \texttt{=}\ e\ \texttt{in}\ 4)$$

specializes to

$$\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ 21$$

that is, the code generation for the surrounding $\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ \ldots$ is interleaved with the specialization-time computation $17 + 4$. It turns out [13, 17], that this feature is essential to achieve sound specialization for languages with effects.

Danvy and Lawall [16] have proposed to extend interleaved code generation to also include dynamic conditionals. Extending the previous example

$$\texttt{lift}\ (17 +^s \texttt{if0}^d\ e\ 4\ 25)$$

specializes to

$$\texttt{if0}\ e\ 21\ 42$$

that is, the computation $(17 + [\ ])$ is performed twice and the conditional floats to the top level of the expression. This feature vastly simplifies "The Trick", a common binding-time improvement [7].

As the name "continuation-based partial evaluation" suggests, the first implementation of interleaved code generation was in terms of a non-standard use of continuations, either in the source program [3] or in the implementation of the specializer [1, 16]. Later on, further state-based implementations have been suggested [11, 23].

In the present work, we start from a monadic specification for continuation-based partial evaluation and suggest an implementation of the underlying monad that does not rely on continuations. We show that our new implementation is equivalent to an implementation with continuations and develop a novel annotated type and effect system for two-level call-by-value lambda calculus that models interleaved code generation (as demonstrated with the `let` and `if` above) as a specialization-time effect. This type system may be used to optimize specialization by taking advantage of (the absence of) specialization-time effects.

Terms	$e ::=$	$x \mid \lambda^b x.e \mid e @^b e \mid \mathtt{let}^b\ x\ \mathtt{=}\ e\ \mathtt{in}\ e \mid$
		$c \mid \mathtt{succ}^b\ e \mid \mathtt{if0}^b\ e\ e\ e \mid \mathtt{lift}\ e \mid \mathtt{reset}\ e$
Binding times	$b ::=$	$s \mid d \qquad s < d$
Types	$\tau ::=$	$\mathtt{int} \mid \rho \xrightarrow{t} \rho$
Annotated types	$\rho ::=$	(τ, b)
Effects	$t ::=$	$\mathbf{1} \mid i(t,t) \mid l(t) \mid t \gg t \mid t \sqcup t$

Fig. 1. Two-level language

Overview In Sec. 2, we present a call-by-value two-level lambda calculus and a type and effect system specifying its static semantics. Section 3 specifies its dynamic semantics via a translation into the computational metalanguage augmented with code-generating operators. The translation requires some operations from the underlying monad and Sec. 4 states these requirements in the form of equational axioms and provides two implementations for the monad, one with continuations and another with substitutions. Section 5 argues informally that the two monads are equivalent. In Sec. 6, we prove type soundness for the two-level lambda calculus and in Sec. 7 we give some examples, how specialization may be optimized with the presented system. Section 8 discusses some related work and Section 9 concludes.

2 Two-level lambda calculus

We consider an applied two-level call-by-value lambda calculus with built-in natural numbers. Its syntax is given in Fig. 1. Each subexpression except for variables, constants, `left` and `reset` expressions is annotated with a binding-time annotation b, which can be s (static) or d (dynamic). The computation rules are beta value reduction for static beta redexes and static `let` expressions, and delta reduction for static successors `succ` and testing for zero `if0`. The only non-standard expression is `reset` e and we will defer its explanation to Sec. 7.

The syntax of two-level types in Fig. 1 is standard [9], except for the addition of effects. Effects characterize the interleaved code generation as exemplified by the `let` x `=` e `in` ... in the introduction. The effect $\mathbf{1}$ indicates that no such code is generated, $l(t)$ indicates that a `let` x `=` e `in` e' is generated where e and x are arbitrary and e' is described by t, $i(t_1, t_2)$ indicates an interleaved `if0` $e\ e_1\ e_2$ with arbitrary e but branches e_1 and e_2 specified by t_1 and t_2, respectively. Of the remaining operations, effect union, $t_1 \sqcup t_2$, specifies interleaved code described by either t_1 or t_2 and $t_1 \gg t_2$ specifies sequential composition. Fig. 2 specifies these operations on effects. In essence, each effect can be simplified to a union of simple effects generated by $\mathbf{1}$, $l(t)$, and $i(t', t'')$. The union operator can be floated to the top level using equations (8), (9), (10), and (11). Sequential composition $\gg$ can be pushed through towards $\mathbf{1}$s using the equations (5), (6), and (7) and then

$$t \sqcup t = t \tag{1}$$
$$t \sqcup t' = t' \sqcup t \tag{2}$$
$$t \sqcup (t' \sqcup t'') = (t \sqcup t') \sqcup t'' \tag{3}$$
$$\mathbf{1} \gg t = t \tag{4}$$
$$l(t') \gg t = l(t' \gg t) \tag{5}$$
$$i(t', t'') \gg t = i((t' \gg t), (t'' \gg t)) \tag{6}$$
$$(t' \gg t'') \gg t = t' \gg (t'' \gg t) \tag{7}$$
$$(t' \sqcup t'') \gg t = (t' \gg t) \sqcup (t'' \gg t) \tag{8}$$
$$l(t' \sqcup t'') = l(t') \sqcup l(t'') \tag{9}$$
$$i(t' \sqcup t'', t) = i(t', t) \sqcup i(t'', t) \tag{10}$$
$$i(t, t' \sqcup t'') = i(t, t') \sqcup i(t, t'') \tag{11}$$

Fig. 2. Algebraic properties of effects

eliminated by (4)[1]. The type and effect system presented in Fig. 3 for specifying binding times is new. It relies on the inference system in Fig. 4 to specify a well-formedness criterion for types. The typing part of the system is standard, the new ingredient is the effect part.

To illustrate the operations on effects, let us consider the rule for function application. It states that if specialization of e_1 and e_2 (in that order) generates interleaved code as described by t_1 and t_2, and if applying the value of e_1 to the value of e_2 generates interleaved code as described by the latent effect t, then the interleaved code generated by $e_1 \mathtt{@}^b e_2$ starts as described by t_1, followed by t_2, and finally t. Similar reasoning applies to the static and the dynamic `let` expressions. In the latter, the explicit $l(\ldots)$ stands for the `let` created by the expression itself (which is always interleaved, so that it can float out of static contexts). Since the outcome of the static conditional cannot be determined before specialization, the interleaved code is approximated by taking the union of the two effects. The dynamic conditional also generates its code interleaved by default and hence need not constrain the binding time of its branches[2].

The `reset` $\cdot$ expression is also a novelty of our presentation. As its typing rule specifies, it is applicable to any dynamic value (a syntax tree) and is used to flush the interleaved code by dumping it on top of that syntax tree. Its effect is empty, **1**, since all interleaved code is now part of the dynamic value.

As usual in an effect system [18], each function type carries a latent effect indicating the effect that is unleashed when the function is applied. The well-formedness judgement enforces the usual constraints on the binding-time an-

[1] This elimination relies on the fact that there are no effect variables in the system.

[2] A thorough discussion of this feature is out of the scope of this paper but may be found in the literature [7, 16].

$$\frac{\vdash_{wft}\ \rho}{\Gamma, x:\rho \vdash x:\rho\ !\ \mathbf{1}} \qquad \Gamma \vdash c : (\mathtt{int}, s)\ !\ \mathbf{1}$$

$$\frac{\Gamma \vdash e : (\mathtt{int}, b)\ !\ t}{\Gamma \vdash \mathtt{succ}^b\ e : (\mathtt{int}, b)\ !\ t} \qquad \frac{\Gamma \vdash e : (\mathtt{int}, s)\ !\ t}{\Gamma \vdash \mathtt{lift}\ \ e : (\mathtt{int}, d)\ !\ t}$$

$$\frac{\begin{array}{c}\Gamma, x:\rho_2 \vdash e : \rho_1\ !\ t\\ \rho = (\rho_2 \xrightarrow{t} \rho_1, b) \quad \vdash_{wft}\ \rho\end{array}}{\Gamma \vdash \lambda^b x.e : \rho\ !\ \mathbf{1}} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : (\rho_2 \xrightarrow{t} \rho_1, b)\ !\ t_1\\ \Gamma \vdash e_2 : \rho_2\ !\ t_2\end{array}}{\Gamma \vdash e_1 @^b e_2 : \rho_1\ !\ t_1 \gg t_2 \gg t}$$

$$\frac{\Gamma \vdash e_1 : (\tau, s)\ !\ t_1 \quad \Gamma, x:(\tau, s) \vdash e_2 : \rho_2\ !\ t_2}{\Gamma \vdash \mathtt{let}^s\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : \rho_2\ !\ t_1 \gg t_2}$$

$$\frac{\Gamma \vdash e_1 : (\tau, d)\ !\ t_1 \quad \Gamma, x:(\tau, d) \vdash e_2 : \rho_2\ !\ t_2}{\Gamma \vdash \mathtt{let}^d\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 : \rho_2\ !\ t_1 \gg l(t_2)}$$

$$\frac{\Gamma \vdash e_1 : (\mathtt{int}, s)\ !\ t_1 \quad \Gamma \vdash e_2 : \rho\ !\ t_2 \quad \Gamma \vdash e_3 : \rho\ !\ t_3}{\Gamma \vdash \mathtt{if0}^s\ e_1\ e_2\ e_3 : \rho\ !\ t_1 \gg (t_2 \sqcup t_3)}$$

$$\frac{\Gamma \vdash e_1 : (\mathtt{int}, d)\ !\ t_1 \quad \Gamma \vdash e_2 : \rho\ !\ t_2 \quad \Gamma \vdash e_3 : \rho\ !\ t_3}{\Gamma \vdash \mathtt{if0}^d\ e_1\ e_2\ e_3 : \rho\ !\ t_1 \gg i(t_2, t_3)}$$

$$\frac{\Gamma \vdash e : (\tau, d)\ !\ t}{\Gamma \vdash \mathtt{reset}\ \ e : (\tau, d)\ !\ \mathbf{1}}$$

Fig. 3. Typing rules

$$\vdash_{wft}\ (\mathtt{int}, b)$$

$$\frac{\begin{array}{c}\vdash_{wft}\ (\tau_2, b_2) \quad \vdash_{wft}\ (\tau_1, b_1) \quad b \le b_1 \quad b \le b_2\\ \textit{if}\ b = d\ \textit{then}\ t = \mathbf{1}\end{array}}{\vdash_{wft}\ ((\tau_2, b_2) \xrightarrow{t} (\tau_1, b_1), b)}$$

Fig. 4. Wellformedness of annotated types

$$
\begin{array}{ll}
\mathcal{T}[\![x]\!] & = \texttt{return}\ x \\
\mathcal{T}[\![\lambda^s x.e]\!] & = \texttt{return}\ (\lambda x.\mathcal{T}[\![e]\!]) \\
\mathcal{T}[\![\lambda^d x.e]\!] & = \texttt{newVar}\ \texttt{>>=}\ \lambda x.\mathcal{T}[\![e]\!]\ \texttt{>>=}\ \lambda e'.\texttt{return}\ (\texttt{LamExp}\ x\ e') \\
\mathcal{T}[\![e_1 \texttt{@}^s e_2]\!] & = \mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda f.\mathcal{T}[\![e_2]\!]\ \texttt{>>=}\ \lambda v.f\texttt{@}v \\
\mathcal{T}[\![e_1 \texttt{@}^d e_2]\!] & = \mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda e_1'.\mathcal{T}[\![e_2]\!]\ \texttt{>>=}\ \lambda e_2'.\texttt{return}\ (\texttt{AppExp}\ e_1'\ e_2') \\
\mathcal{T}[\![\texttt{let}^s\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!] & = \mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda x.\mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\texttt{let}^d\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!] & = \mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda e_1'.\texttt{newVar}\ \texttt{>>=}\ \lambda x.\texttt{mlet}\ x\ e_1'\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![c]\!] & = \texttt{return}\ c \\
\mathcal{T}[\![\texttt{succ}^s\ e]\!] & = \mathcal{T}[\![e]\!]\ \texttt{>>=}\ \lambda v.\texttt{return}\ (\texttt{succ}\ v) \\
\mathcal{T}[\![\texttt{succ}^d\ e]\!] & = \mathcal{T}[\![e]\!]\ \texttt{>>=}\ \lambda e'.\texttt{return}\ (\texttt{SuccExp}\ e') \\
\mathcal{T}[\![\texttt{if0}^s\ e_1\ e_2\ e_3]\!] & = \mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda v_1.\texttt{if0}\ v_1\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!] \\
\mathcal{T}[\![\texttt{if0}^d\ e_1\ e_2\ e_3]\!] & = \mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda e_1'.\texttt{mif}\ e_1'\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!] \\
\mathcal{T}[\![\texttt{lift}\ e]\!] & = \mathcal{T}[\![e]\!]\ \texttt{>>=}\ \lambda v.\texttt{return}\ (\texttt{ConExp}\ v) \\
\mathcal{T}[\![\texttt{reset}\ e]\!] & = \texttt{mreset}\ \mathcal{T}[\![e]\!]
\end{array}
$$

Fig. 5. Translation into augmented metalanguage

notations. In addition, it ensures that the application of a dynamic (run-time) function does not generate interleaved code.

3 Monadic specification of partial evaluation

Effects have been abstracted and described using monads [10,25]. For that reason, we construct a monadic specification of partial evaluation, using special operators that exert the effects, `mlet` for $l(t)$ and `mif` for $i(t,t)$, or flush them, `mreset`. There is also a `newVar` operator for creating fresh variable names, which is left unspecified to avoid further complication.

We specify the semantics of the two-level calculus by translation into a variant of Moggi's computational metalanguage [21], that is, the call-by-name lambda calculus augmented with the monadic `return` $\cdot$ and bind operations ($\cdot$`>>=`$\cdot$) plus the special `newVar`, `mlet`, `mif`, and `mreset` operators. There are also constructors for generating syntax trees for applied call-by-value lambda calculus.

For concreteness, we specify syntax trees and the operations in the form of Haskell definitions [12]. Here is the datatype for syntax trees.

```
type Var = String
data Exp =
    VarExp Var          -- variable
  | ConExp String       -- constant
  | SuccExp Exp         -- successor
  | LamExp Var Exp      -- lambda
  | AppExp Exp Exp      -- application
  | LetExp Var Exp Exp  -- let
  | IfExp  Exp Exp Exp  -- conditional
```

We capture the further requirements in a type class `PEMonad`, which states the required operations on a monad `m`, suitable for implementing partial evaluation, and their types.

```
class Monad m => PEMonad m where
  mlet   :: Var -> Exp -> m a -> m a
  mif    :: Exp -> m a -> m a -> m a
  mreset :: m Exp -> m Exp
```

The type signatures clearly show that

- `mlet` transforms a computation (its last argument) into another computation;
- `mif` transforms two computations into another computation;
- the type (and hence the binding time) of the body of the `mlet` as well as the type of the branches of the `mif` is *unconstrained*;
- `mreset` `mreset` requires that its argument computation returns a syntax tree (*i.e.*, a dynamic value) and it returns a syntax tree, too.

The translation rules in Fig. 5 are reasonably straightforward. For the static constructs they correspond exactly to the standard monadic translation. As an example for a dynamic construct, we consider the translation of the dynamic lambda:

$$\mathcal{T}[\![\lambda^d x.e]\!] = \texttt{newVar} \mathrel{\texttt{>>=}} \lambda x.\mathcal{T}[\![e]\!] \mathrel{\texttt{>>=}} \lambda e'.\texttt{return}\ (\texttt{LamExp}\ x\ e')$$

The translated term creates a fresh variable name and binds it to x, the bound variable of the translated lambda. In that environment, it specializes e, binds the result to e', and returns the syntax tree `LamExp` x e'. For understanding the rules, it is important to keep in mind that $\mathcal{T}[\![\cdot]\!]$ is a *translation*, that is, the x and e on both sides are *identical*.

To avoid clutter in the translation (and in the proofs below) we assume that the type `Var` is a subset of the type `Exp` with `VarExp` being the injection function. The type-correct right-hand side for the dynamic lambda would be

$$\texttt{newVar} \mathrel{\texttt{>>=}} \lambda x'.\texttt{let}\ x\ \texttt{=}\ \texttt{VarExp}\ x'\ \texttt{in}\ \mathcal{T}[\![e]\!] \mathrel{\texttt{>>=}} \lambda e'.\texttt{return}\ (\texttt{LamExp}\ x'\ e')$$

and similarly in the translation of the dynamic `let`.

4 Monads for partial evaluation

Having constructed the monadic translation for the two-level lambda calculus, the next task is to define a suitable monad that supports the operations, `mlet`, `mif`, and `mreset`.[3] In fact, we will present two such monads. One of them is the continuation monad and its definitions correspond to the usual implementation

[3] Recall that we leave `newVar` unspecified, although it is straightforward to add.

of continuation-based partial evaluation [1,16]. The other monad is, perhaps surprisingly, a simple substitution monad. Nevertheless, as we will show in Sec. 4.3, both monads fulfill the expected laws for the new operations and we will use these laws in Sec. 5 to prove that the results of interpreting a translated two-level term in either monad are equivalent.

4.1 Continuation monad

The type for the continuation monad is standard. Since we are defining a monad for partial evaluation, where the result is always a syntax tree, the type of answers is `Exp`. For technical reasons, Haskell requires the definition of a new data type `C` and a corresponding untagging operation `unC`.

```
data C a = C ((a -> Exp) -> Exp)
unC (C k) = k
```

The definitions for `return` and `>>=` are standard[4].

```
instance Monad C where
  return x   =  C (\ c -> c x)
  ma >>= f   =  C (\ c -> unC ma (\ x -> unC (f x) c))
```

Of course, the essence of continuation-based partial evaluation lies in the defintions of the `PEMonad` operators. The definitions below correspond to code fragments from Bondorf, Danvy, and Lawall [1, 16]. In particular, note how the continuation is duplicated in the `mif`.

```
instance PEMonad C where
  mlet v e ma =
    C (\ c -> LetExp v e (unC ma c))
  mif e m1 m2 =
    C (\ c -> IfExp e (unC m1 c) (unC m2 c))
  mreset me =
    C (\ c -> c (unC me id))
```

4.2 Substitution monad

A computation of type `S a` in the substitution monad is a tree where the leaves hold the computed values of type `a`. Each inner nodes is either an `L` node, specifying an interleaved `let` expression, or an `I` node, specifying an interleaved `if` expression. The following definition captures this intuition.

```
data S a = V a
         | L Var Exp (S a)
         | I Exp (S a) (S a)
```

[4] Recall that `\ x -> e` is a lambda abstraction.

The monadic operations are simple to specify. The `return` operation creates a trivial tree using the `V` constructor. The bind operation m `>>=` f propagates f to each leaf of m, applies it to the value x found there, and pastes the resulting tree $f\ x$ in place of the leaf.

```
instance Monad S where
  return x = V x
  ma >>= f = case ma of
               V x       -> f x
               L v e s   -> L v e (s >>= f)
               I b s1 s2 -> I b (s1 >>= f) (s2 >>= f)
```

The operations of the `PEMonad` class are straightforward to implement, too. The interleaved code can be prepended to the tree since all operations are applied to the values in the leaves of the tree, as specified by the `>>=` operation. The `mreset` operation collects the tree and translates it into an expression.

```
instance PEMonad S where
  mlet v e ma = L v e ma
  mif e m1 m2 = I e m1 m2
  mreset me =
    case me of
      V e        -> return e
      L v e me   -> mreset me >>= \ e' -> return (LetExp v e e')
      I b m1 m2 -> mreset m1 >>= \ e1 -> mreset m2 >>= \ e2 ->
                   return (IfExp b e1 e2)
```

4.3 Laws

Besides fulfilling the usual monad laws [26], a `PEMonad` must obey a number of additional laws. These laws characterize the interaction of `mlet` and `mif` with the `>>=` operator, as well as the interaction of `mlet` and `mif` with `mreset`.

The first two laws state that the `mlet` and `mif` operations are both associative with respect to `>>=`.

```
(MLB)  mlet v e ma >>= f == mlet v e (ma >>= f)
(MIB)  mif e m1 m2 >>= f == mif e (m1 >>= f) (m2 >>= f)
```

The next three laws state that the `mreset` operation absorbs an `mlet` or an `mif` at the price of generating the corresponding code.

```
(MRR)  mreset (return e)      == return e
(MRL)  mreset (mlet v1 e1 me) == mreset me >>= λ e ->
                                 return (LetExp v1 e1 e)
(MRI)  mreset (mif e m1 m2)   == mreset m1 >>= λ e1 ->
                                 mreset m2 >>= λ e2 ->
                                 return (IfExp e e1 e2)
```

Lemma 1. *The laws (MLB), (MIB), (MRR), (MRL), and (MRI) hold in* C *and* S.

Proof. See Appendix A.

In addition, the `mlet` and `mif` operations for a `PEMonad` obey the monadic flattening laws on the level of generated syntax trees. To state these laws, we define two auxiliary operations, `mmlet` and `mmif`, that incorporate the computation of their first expression:

```
mmlet :: Var -> m Exp -> m a -> m a
mmif  :: m Exp -> m a -> m a -> m a

mmlet v me ma = me >>= \ e -> mlet v e ma
mmif me m1 m2 = me >>= \ e -> mif e m1 m2
```

Using these definitions, we can state flattening laws corresponding to the third monad law (associativity of let/bind), on the level of interleaved generated code.

```
mmlet v2 (mmlet v1 m1 m2) m3  ==  mmlet v1 m1 (mmlet v2 m2 m3)
mmlet v (mmif m1 m2 m3) m4    ==  mmif m1 (mmlet v m2 m4) (mmlet v m3 m4)
mmif (mmlet v m1 m2) m3 m4    ==  mmlet v m1 (mmif m2 m3 m4)
mmif (mmif m1 m2 m3) m4 m5    ==  mmif m1 (mmif m2 m4 m5) (mmif m3 m4 m5)
```

They also hold for both, `S` and `C`, and they are proven using `(MLB)` and `(MIB)`.

4.4 New variables

As stated before, we are not going to specify the `newVar` operation completely. However, we have to make some assumptions about it. Its type is

```
newVar :: PEMonad m => m Var
```

and its only effect is the generation of a fresh variable name. That is,

$$\mathtt{newVar} == (\nu \mathtt{x} \in \mathtt{Var})\ \mathtt{return\ x}$$

5 Equivalence of PE monads

It turns out that the laws `(MLB)`, `(MIB)`, `(MRR)`, `(MRL)`, and `(MRI)` completely characterize the computational behavior of the operations of `PEMonad`. To make that formal, let λ_{PE} be the equational theory of the computational metalanguage (beta equivalence plus the three monad laws) augmented by delta rules for successor and `if0` and the mentioned `PEMonad` laws.

We take equivalence of the two `PEMonad`s introduced above to mean that, for each closed two-level lambda term e of type (τ, d), "running" the term $\mathcal{T}[\![e]\!]$ yields the same answer. Here are the definitions for running `C` and `S` computations.

```
runC :: C Exp -> Exp
runC me = unC (mreset me) id

runS :: S Exp -> Exp
runS se = let V x = mreset se in x
```

To (informally) demonstrate equivalence in λ_{PE}, observe that the laws (MLB) and (MIB) allow `mlet` and `mif` operations to float outside of `>>=` operations. That means, each translated term is equal to one where all the `mlet`s and `mif`s, interspersed with `newVar`s, are aggregated either at the top of the term or in the argument of `mreset`s. After application of `mreset` at the toplevel, in `runC` or `runS`, and performing `newVar`s, the laws (MRR), (MRL), and (MRI) enable all `mlet`s and `mif`s to be removed. The resulting term does not contain monadic operators except `return` and `>>=`, hence its computed value is independent of the underlying monad.

6 Soundness

In this section, we connect the type system with the semantics. We shall argue in denotational style by first defining a semantics of a two-level type with effect and then proving type soundness by examining the translated term in the `S` monad. Since the simply-typed lambda calculus, the basis of our two-level calculus, is strongly normalizing, we state the semantics in terms of sets. In addition, we assume a standard semantics mapping $[\![\cdot]\!]$ that maps an expression, e, and a value environment, φ, to a value (see for example [19]). We make this more formal by defining the carriers of an appropriate Henkin model for the type language of the metalanguage.

Definition 1 (Types of the metalanguage).

$$\sigma ::= \texttt{Var} \mid \texttt{Exp} \mid \texttt{M}\ \sigma \mid \sigma \texttt{ ->} \sigma$$

Definition 2 (Semantic values). *Let* $[\![\texttt{Var}]\!]$ *be the set of variable names.*
Let $[\![\texttt{Exp}]\!]$ *be the set of expressions generated by the definition of* **Exp** *in Sec. 4.*
Let $[\![\texttt{Int}]\!]$ *be the set of natural numbers.*
For each type σ*, let*

$$\begin{aligned}[\![\texttt{M}\ \sigma]\!] = &\ \{V\ v \mid v \in [\![\sigma]\!]\}\\ &\cup \{L\ \ x\ \ e\ \ m \mid x \in [\![\texttt{Var}]\!], e \in [\![\texttt{Exp}]\!], m \in [\![\texttt{M}\ \sigma]\!]\}\\ &\cup \{I\ \ e\ \ m_1\ \ m_2 \mid e \in [\![\texttt{Exp}]\!], m_i \in [\![\texttt{M}\ \sigma]\!]\}\end{aligned}$$

For each types σ_2 *and* σ_1*, let*

$$[\![\sigma_2 \texttt{ ->} \sigma_1]\!] = [\![\sigma_1]\!]^{[\![\sigma_2]\!]}$$

that is, the set theoretic functions from $[\![\sigma_2]\!]$ *to* $[\![\sigma_1]\!]$*.*

Clearly, this family of carrier sets with set-theoretic function application satisfies the preconditions for a Henkin model. Now, we can define the semantics of an annotated type in terms of the Henkin model.

Definition 3 (Semantics of annotated types).

$$\begin{array}{ll}
[\![\rho\,!\,\mathbf{1}]\!] & = \{\mathtt{V}\ v \mid v \in [\![\rho]\!]\} \\
[\![\rho\,!\,l(t)]\!] & = \{\mathtt{L}\ \ x\ \ e\ \ v \mid x \in [\![\mathtt{Var}]\!], e \in [\![\mathtt{Exp}]\!], v \in [\![\rho\,!\,t]\!]\} \\
[\![\rho\,!\,i(t_1,t_2)]\!] & = \{\mathtt{I}\ \ e\ \ v_1\ \ v_2 \mid e \in [\![\mathtt{Exp}]\!], v_i \in [\![\rho\,!\,t_i]\!]\} \\
[\![\rho\,!\,t_1 \sqcup t_2]\!] & = [\![\rho\,!\,t_1]\!] \cup [\![\rho\,!\,t_1]\!] \\
[\![(\tau,d)]\!] & = [\![\mathtt{Exp}]\!] \\
[\![(\mathtt{int},s)]\!] & = [\![\mathtt{Int}]\!] \\
[\![(\rho_2 \xrightarrow{t} \rho_1, s)]\!] & = \{f \mid (\forall x \in [\![\rho_2]\!])\ f\ x \in [\![\rho_1\,!\,t]\!]\}
\end{array}$$

It turns out that the translation $\mathcal{T}[\![\cdot]\!]$ is type preserving.

Theorem 1 (Translation soundness). *Suppose that $\Gamma \vdash e : \rho\,!\,t$.*
Then $\mathcal{T}[\![\Gamma]\!] \vdash \mathcal{T}[\![e]\!] : \mathtt{M}\ \mathcal{T}[\![\rho]\!]$ in the simply-typed lambda calculus where

- M *is a* PEMonad *(for instance,* S *or* C*);*
- $\mathcal{T}[\![\Gamma]\!](x) = \mathcal{T}[\![\Gamma(x)]\!]$*, for all* $x \in dom(\Gamma)$*;*
- $\mathcal{T}[\![(\tau,d)]\!] = \mathtt{Exp}$*;*
- $\mathcal{T}[\![(\mathtt{int},s)]\!] = \mathtt{Int}$*;*
- $\mathcal{T}[\![(\rho_2 \xrightarrow{t} \rho_1, s)]\!] = \mathcal{T}[\![\rho_2]\!]\ \mathtt{->M}\ \mathcal{T}[\![\rho_1]\!]$.

In other words, Theorem 1 establishes that each well-typed two-level lambda expression may be translated into a standard functional programming language (*e.g.*, Haskell or ML) and executed there to achieve specialization. Hence, the translation $\mathcal{T}[\![\cdot]\!]$ creates a *generating extension*, *i.e.*, a dedicated specializer for one particular two-level lambda expression. See the author's article [24] for more background information.

To prove type and effect soundness, we need an auxiliary lemma. It establishes that all types inferred by the inference system are well-formed.

Lemma 2. *Suppose that $\Gamma \vdash e : \rho\,!\,t$ and $(\forall x \in dom(\Gamma))\ \vdash_{wft}\ \Gamma(x)$.*
Then $\vdash_{wft}\ \rho$.

Proof. Standard induction on the derivation of $\Gamma \vdash e : \rho\,!\,t$.

Theorem 2 (Type and effect soundness). *Suppose that $\Gamma \vdash e : \rho\,!\,t$ and, for all $x \in dom(\Gamma)$, $\varphi(x) \in [\![\Gamma(x)]\!]$ and $\vdash_{wft}\ \Gamma(x)$.*
Then $[\![\mathcal{T}[\![e]\!]]\!]\varphi \in [\![\rho\,!\,t]\!]$.

Proof. See Appendix B

Adding recursion to our framework would give rise to two complications. First, we would have to base the semantics on domains instead of sets. Second, we would have to add recursion in the form $\mu e.t$ to the language of effects. Both are standard [19, 22] and are elided here for the sake of conciseness.

7 Optimized specification

The two-level calculus presented in Sec. 2 has one unusual feature compared to standard accounts (*e.g.*, in the text book [15] or in the papers about continuation-based partial evaluation [1, 16]). The expression `reset` e is usually not made explicit in the two-level language, but rather hidden in the translation of the dynamic versions of the binding constructs, $\mathtt{let}^d\ x\ \mathtt{=}\ e\ \mathtt{in}\ \ldots$ and $\lambda^d x. \ldots$.

With our formal system, the `reset` e expression plays a clearly defined role. It coerces a dynamic two-level term with arbitrary effect into a two-level term without effect. Instead of integrating `reset` $\cdot$ into the translation of $\lambda^d x.e$, the type system makes sure that the effect of e is $\mathbf{1}$ (that is, empty). This way, the binding-time analysis is free to place `reset` $\cdot$ as early as possible (this avoids code duplication in specialized expressions and hence more efficient specialization) or to avoid placing `reset` $\cdot$ at all, if the type system infers that the effect is $\mathbf{1}$, already.

Here are two examples that illustrate either point.

- Code duplication. Let $\Gamma(f_1) = \Gamma(f_2) = (\rho_2 \xrightarrow{\mathbf{1}} \rho_1, d)$, $\Gamma(x) = (\mathtt{int}, d)$, and $\Gamma(a) = \rho_2$. Then

$$\Gamma \vdash (\mathtt{if0}^d\ x\ f_1\ f_2)\mathtt{@}^d a : \rho_1 \mathbin{!} i(\mathbf{1}, \mathbf{1})$$

  Specializing this code as it is would lead to duplicating the application and its argument a:

$$\mathtt{if0}^d\ x\ (f_1 \mathtt{@}^d a)\ (f_2 \mathtt{@}^d a)$$

  However, since both f_1 and f_2 are dynamic, the binding-time analysis may wrap the conditional into a `reset` $\cdot$ so that the effect of the term vanishes:

$$\Gamma \vdash (\mathtt{reset}\ (\mathtt{if0}^d\ x\ f_1\ f_2))\mathtt{@}^d a : \rho_1 \mathbin{!} \mathbf{1}$$

  Specializing this code avoids the duplication:

$$(\mathtt{if0}\ x\ f_1\ f_2)\mathtt{@}a$$

- Avoiding the placement of `reset` $\cdot$. Suppose that

$$\Gamma, x : \rho_2 \vdash e : \rho_1 \mathbin{!} \mathbf{1}$$

  where $\vdash_{wft} (\rho_2 \xrightarrow{\mathbf{1}} \rho_1, d)$ so that

$$\Gamma \vdash \lambda^d x.e : (\rho_2 \xrightarrow{\mathbf{1}} \rho_1, d) \mathbin{!} \mathbf{1}$$

  Observe that it is not necessary to insert a `reset` $\cdot$ on top of the lambda's body because the body's effect is empty. In constrast, a conventional continuation-based partial evaluator *always* insert a `reset` $\cdot$ in this place.

8 Related work

Consel and Danvy [3] were the first to argue that the binding time of dynamic `let` expressions can be improved converting source programs to continuation-passing style.

Subsequently, Bondorf [1] discovered that the same binding-time improvement can be achieved by writing the partial evaluator in (not-quite) continuation-passing style. In fact, it is essential to deviate from standard continuation-passing style in the treatment of the dynamic `let` and `lambda` expressions.

Danvy and Lawall [16] showed that the non-standard handling of continuations in Bondorf's partial evaluator can be implemented using the control operators `shift` and `reset` [6] that provide compositional continuations. They also generalize the treatment of the dynamic `let` to the dynamic `if`.

It was later seen that the treatment of dynamic `let` expressions (first termed "let-insertion" [2]) enabled by continuation-based partial evaluation was essential to achieve a satisfactory and sound specialization for languages with computational effects [13, 17].

Recently, the work on type-directed partial evaluation for call-by-value languages [5, 11] has spread new interest on the implementation of let-insertion. In addition, work to define powerful, efficient online program generators has lead to the discovery of a state-based implementation of let-insertion [23]. The latter work also proves the equivalence of their let-insertion to the continuation-based method using a type-indexed relation. However, their let-insertion does not scale to if-insertion in an obvious way and our correspondence is much more direct.

9 Conclusion

This work has two major contributions.

1. It presents a new and simple formalization and implementation of continuation-based partial evaluation. This formalization is pleasingly easy to reason about and it is proven equivalent to the traditional one with continuations.
2. It presents a new type and effect system for two-level lambda calculus. This type system can form the basis for a binding-time analysis that constructs optimized generating extensions.

In future work, we would like to examine the connection to the state-based formulation of continuation-based partial evaluation and we would like to design and implement an actual binding-time analysis based on the type system presented in this work. It would also be interesting to compare the effiency of our substitution-based implementation with the ones using continuations and state.

References

1. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, California, USA, June 1992.

2. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.
3. Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, number 523 in Lecture Notes in Computer Science, pages 496–519, Cambridge, MA, 1991. Springer-Verlag.
4. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
5. Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 1996 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg, Fla., January 1996. ACM Press.
6. Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.
7. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, November 1996.
8. Dirk Dussart, Eddy Bevers, and Karel De Vlaminck. Polyvariant constructor specialization. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 54–63, La Jolla, CA, June 1995. ACM Press.
9. Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proceedings of the 1995 International Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 118–136, Glasgow, Scotland, September 1995. Springer-Verlag.
10. Andrzej Filinski. Representing layered monads. In Alexander Aiken, editor, *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, USA, January 1999. ACM Press.
11. Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Krakow, Poland, 2001. Springer-Verlag.
12. Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition`, December 1998.
13. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.
14. John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, number 1706 in Lecture Notes in Computer Science, Copenhagen, Denmark, 1999. Springer-Verlag.
15. Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
16. Julia Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM Press.
17. Julia Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proceedings of the Theoretical Aspects of Computer Software*, number 1281 in Lecture Notes in Computer Science, pages 165–190, Sendai, Japan, September 1997. Springer-Verlag.

18. John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, January 1988. ACM Press.
19. John Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
20. Torben Æ. Mogensen. Inherited limits. In Hatcliff et al. [14], pages 189–202.
21. Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.
22. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
23. Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.
24. Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, September 1999.
25. Philip Wadler and Peter Thiemann. The marriage of monads and effects. *ACM Transactions on Computational Logic*, 4(1):1–32, January 2003.
26. Philip L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, 1990. ACM Press.

APPENDIX

A Proof of Lemma 1

The proofs are type checked Haskell scripts using the operator `=:=` for equality.

(MLB)

```
-- bind_let for S
ls_bind_let_S v e ma f =
  mlet v e ma >>= f =:=
  L v e ma >>= f =:=
  L v e (ma >>= f) =:=
  mlet v e (ma >>= f)

-- bind_let for C
ls_bind_let_C v e ma f =
  mlet v e ma >>= f =:=
  C (\ c -> unC (mlet v e ma) (\ x -> unC (f x) c)) =:=
  C (\ c -> unC (C (\ c -> LetExp v e (unC ma c))) (\ x -> unC (f x) c)) =:=
  C (\ c -> (\ c -> LetExp v e (unC ma c)) (\ x -> unC (f x) c)) =:=
  C (\ c -> LetExp v e (unC ma (\ x -> unC (f x) c))) =:=
  C (\ c -> LetExp v e ((\ c -> unC ma (\ x -> unC (f x) c)) c)) =:=
  C (\ c -> LetExp v e (unC (C (\ c -> unC ma (\ x -> unC (f x) c))) c)) =:=
  mlet v e (C (\ c -> unC ma (\ x -> unC (f x) c))) =:=
  mlet v e (ma >>= f)
```

(MIB) Analogous to `(MLB)`

(MRR)

```
-- mreset (return e) == return e
mreset_return_S e =
  mreset (return e) =:=                               -- def return
  mreset (V e) =:=                                    -- def sreset
  V e =:=                                             -- def return
  return e

mreset_return_C e =
  mreset (return e) =:=                               -- def return
  mreset (C (\ c -> c e)) =:=                         -- def creset
  C (\ c -> c (unC (C (\ c -> c e)) id)) =:=          -- unC/C
  C (\ c -> c ((\ c -> c e) id)) =:=                  -- beta
  C (\ c -> c e) =:=                                  -- def return
  return e
```

(MRL)

```
-- mreset (mlet v1 e1 me) == mreset me >>= \ e -> return (LetExp v1 e1 e)
mreset_mlet_S v1 e1 me =
  mreset (mlet v1 e1 me) =:=                          -- def slet
  mreset (L v1 e1 me) =:=                             -- def sreset
```

```
  mreset me >>= \ e' -> return (LetExp v1 e1 e')

mreset_mlet_C v1 e1 me =
  mreset (mlet v1 e1 me) =:=                               -- def clet
  mreset (C (\ c -> LetExp v1 e1 (unC me c))) =:=          -- def creset
  C (\ c -> c (unC (C (\ c -> LetExp v1 e1 (unC me c))) id)) =:= -- simplify
  C (\ c -> c ((\ c -> LetExp v1 e1 (unC me c)) id)) =:=    -- simplify
  C (\ c -> c (LetExp v1 e1 (unC me id))) =:=              -- simplify
  C (\ c -> (\ x -> c (LetExp v1 e1 x)) (unC me id))  =:=   -- simplify
  C (\ c -> (\ c -> c (unC me id)) (\ x -> c (LetExp v1 e1 x)))  =:= -- def >>=
  C (\ c -> unC (C (\ c -> c (unC me id))) (\ x -> unC ((\ e -> C (\ c -> c (LetExp v1 e1 e))) x) c))  =:= -- def >>=
  C (\ c -> c (unC me id)) >>= \ e -> C (\ c -> c (LetExp v1 e1 e)) =:= -- def creset
  mreset me >>= \ e -> C (\ c -> c (LetExp v1 e1 e)) =:=           -- def return
  mreset me >>= \ e -> return (LetExp v1 e1 e)
```

(MRI)

```
-- mreset (mif e m1 m2) ==
-- mreset m1 >>= \ e1 -> mreset m2 >>= \ e2 -> return (IfExp e e1 e2)
mreset_mif_S e m1 m2 =
  mreset (mif e m1 m2) =:=                                 -- def sif
  mreset (I e m1 m2) =:=                                           -- def sreset
  mreset m1 >>= \ e1 -> mreset m2 >>= \ e2 -> return (IfExp e e1 e2)

mreset_mif_C e m1 m2 =
  mreset (mif e m1 m2) =:=                                 -- def cif
  mreset (C (\ c -> IfExp e (unC m1 c) (unC m2 c))) =:=     -- def creset
  C (\ c -> c (unC (C (\ c -> IfExp e (unC m1 c) (unC m2 c))) id)) =:= -- simplify
  C (\ c -> c ((\ c -> IfExp e (unC m1 c) (unC m2 c)) id)) =:= -- simplify
  C (\ c -> c (IfExp e (unC m1 id) (unC m2 id))) =:=        -- beta
  C (\ c -> (\ e1 -> (\ e2 -> c (IfExp e e1 e2)) (unC m2 id)) (unC m1 id)) =:= -- beta
  C (\ c -> (\ c -> c (unC m1 id)) (\ e1 -> (\ c -> (\ c -> c (unC m2 id)) (\ e2 -> (\ c -> c (IfExp e e1 e2)) c)) c)) =:= -- unC/C
  C (\ c -> (\ c -> c (unC m1 id)) (\ e1 -> unC (C (\ c -> (\ c -> c (unC m2 id)) (\ e2 -> unC (C (\ c -> c (IfExp e e1 e2))) c))) c)) =:= -- beta
  C (\ c -> (\ c -> c (unC m1 id)) (\ e1 -> unC ((\ e1 -> C (\ c -> (\ c -> c (unC m2 id)) (\ e2 -> unC ((\ e2 -> C (\ c -> c (IfExp e e1 e2))) e2) c))) e1) c)) =:= -- alpha
  C (\ c -> (\ c -> c (unC m1 id)) (\ x -> unC ((\ e1 -> C (\ c -> (\ c -> c (unC m2 id)) (\ x -> unC ((\ e2 -> C (\ c -> c (IfExp e e1 e2))) x) c))) x) c)) =:= -- simplify u
  C (\ c -> unC (C (\ c -> c (unC m1 id))) (\ e1 -> unC ((\ e1 -> C (\ c -> unC (C (\ c -> c (unC m2 id))) (\ e2 -> unC ((\ e2 -> C (\ c -> c (IfExp e e1 e2))) e2) c))) e1) c
  C (\ c -> unC (C (\ c -> c (unC m1 id))) (\ e1 -> unC ((\ e1 -> C (\ c -> c (unC m2 id)) >>= \ e2 -> C (\ c -> c (IfExp e e1 e2))) e1) c)) =:= -- def >>=
  C (\ c -> c (unC m1 id)) >>= \ e1 -> C (\ c -> c (unC m2 id)) >>= \ e2 -> C (\ c -> c (IfExp e e1 e2)) =:= -- creset, return
  mreset m1 >>= \ e1 -> mreset m2 >>= \ e2 -> return (IfExp e e1 e2)
```

B Proof of Theorem 2

To avoid clutter in the proof, we assume that variable names are also expressions, that is, the type `Var` is a subset of the type `Exp`. This avoids wrapping variable names into the `VarExp` constructor and also extracting them from an expression. Also, we argue rather informally about the semantics, mostly using rewriting on the expression level. For the whole proof, we assume that the translated expression is interpreted in the `S` monad, which we have shown to be equivalent to the `C` monad.

Proof. By induction on the derivation of $\Gamma \vdash e : \rho \,!\, t$.

Case $\Gamma, x : \rho \vdash x : \rho \,!\, \mathbf{1}$. It holds that $[\![\texttt{return}\ x]\!]\varphi = \texttt{V}\ \varphi(x) \in [\![\rho \,!\, \mathbf{1}]\!]$ by assumption and definition of $[\![\cdot]\!]$.

Case $\Gamma \vdash c : (\texttt{int}, s) \,!\, \mathbf{1}$. It holds that $[\![\texttt{return}\ c]\!]\varphi = \texttt{V}\ c \in [\![(\texttt{int}, s) \,!\, \mathbf{1}]\!]$ since $c \in [\![(\texttt{int}, s)]\!]$.

Case $\dfrac{\Gamma \vdash e : (\texttt{int}, b) \,!\, t}{\Gamma \vdash \texttt{succ}^b\ e : (\texttt{int}, b) \,!\, t}$. There are two cases, $b = s$ and $b = d$.

For $b = s$, we need to evaluate $[\![\mathcal{T}[\![\texttt{succ}^s\ e]\!]]\!]\varphi = [\![\mathcal{T}[\![e]\!]\texttt{>>=}\lambda v.\texttt{return}\ (\texttt{succ}\ v)]\!]\varphi$. By induction, $[\![\mathcal{T}[\![e]\!]]\!]\varphi \in [\![(\texttt{int}, s) \,!\, t]\!]$. An auxiliary induction on t shows that $[\![\mathcal{T}[\![e]\!]\ \texttt{>>=}\ \lambda v.\texttt{return}\ (\texttt{succ}\ v)]\!]\varphi \in [\![(\texttt{int}, s) \,!\, t]\!]$, too:

- Case $\mathbf{1}$. In this case, $[\![\mathcal{T}[\![e]\!]]\!]\varphi = \texttt{V}\ v$, for some v, and hence $[\![\texttt{V}\ v\ \texttt{>>=}\ \lambda v.\texttt{return}\ (\texttt{succ}\ v)]\!]\varphi = [\![\texttt{return}\ (\texttt{succ}\ v)]\!]\varphi[v \mapsto v] = \texttt{V}\ v$.

– Case $l(t')$. In this case, $[\![\mathcal{T}[\![e]\!]]\!]\varphi = \texttt{L x e } v$, for some $v \in [\![(\texttt{int}, s)\,!\,t']\!]$. Hence, by (MLB), $[\![\texttt{L x e } v \texttt{>>=} \lambda v.\texttt{return (succ } v)]\!]\varphi = [\![\texttt{L x e } (v \texttt{>>=} \lambda v.\texttt{return (succ } v))]\!]\varphi = \texttt{L x e } [\![(v \texttt{ >>= } \lambda v.\texttt{return (succ } v))]\!]\varphi$. By the auxiliary induction, $[\![(v \texttt{ >>= } \lambda v.\texttt{return (succ } v))]\!]\varphi \in [\![(\texttt{int}, s)!t']\!]$, so that $[\![\texttt{L x e } v \texttt{>>=} \lambda v.\texttt{return (succ } v)]\!]\varphi \in [\![(\texttt{int}, s)\,!\,l(t')]\!]$.

– Case $i(t', t'')$ is similar.

– Case $t' \sqcup t''$ is immediate by appeal to the inductive hypothesis.

For $b = d$, we have to show that $[\![\mathcal{T}[\![\texttt{succ}^d\ e]\!]]\!]\varphi = [\![\mathcal{T}[\![e]\!] \texttt{>>=} \lambda e'.\texttt{return (succ}^r\ e')]\!]\varphi \in [\![(\texttt{int}, d)\,!\,t]\!]$. The proof is analogous.

Case $\texttt{lift}\ \ e$ is also analogous.

Case $\dfrac{\Gamma, x : \rho_2 \vdash e : \rho_1\,!\,t \qquad \rho = (\rho_2 \xrightarrow{t} \rho_1, b) \quad \vdash_{wft} \rho}{\Gamma \vdash \lambda^b x.e : \rho\,!\,\mathbf{1}}$. There are two subcases.

Subcase $b = s$. Consider $[\![\mathcal{T}[\![\lambda^s x.e]\!]]\!]\varphi = [\![\texttt{return } (\lambda x.\mathcal{T}[\![e]\!]) = \texttt{V } [\![\lambda x.\mathcal{T}[\![e]\!]]\!]]\!]\varphi$. Clearly, this result matches the effect $\mathbf{1}$. Hence, it remains to show that $[\![\lambda x.\mathcal{T}[\![e]\!]]\!]\varphi \in [\![(\rho_2 \xrightarrow{t} \rho_1, s)]\!]$. That is, for all $v \in [\![\rho_2]\!]$, it must be that $[\![\mathcal{T}[\![e]\!]]\!]\varphi[x := v] \in [\![\rho_1\,!\,t]\!]$. But this is exactly the induction hypothesis.

Subcase $b = d$. Consider $[\![\mathcal{T}[\![\lambda^d x.e]\!]]\!]\varphi = \texttt{newVar>>=}\lambda x.\mathcal{T}[\![e]\!] \texttt{ >>= } \lambda e'.\texttt{return } (\lambda^r x.e')$. Since $[\![\texttt{newVar}]\!]\varphi = \texttt{V x}$, for some $\texttt{x} \in [\![\texttt{Var}]\!]$, we can apply the induction hypothesis to find that $[\![\mathcal{T}[\![e]\!]]\!]\varphi[x \mapsto \texttt{VarExp x}] \in [\![\rho_1\,!\,t]\!]$. Since $\vdash_{wft} \rho$ is assumed and $b = d$, we know that $t = \mathbf{1}$ and $\rho_1 = (\tau_1, d)$ and $\rho_2 = (\tau_2, d)$ (and fortunately, $\texttt{VarExp x} \in [\![(\tau_2, d)]\!] = [\![\texttt{Exp}]\!]$), so that $[\![\rho_1\,!\,t]\!] = \{\texttt{V e} \mid \texttt{e} \in [\![\texttt{Exp}]\!]\}$. Again, by definition of >>=, the result of $[\![\mathcal{T}[\![\lambda^d x.e]\!]]\!]\varphi$ is $\texttt{V } (\lambda^r \texttt{x}.e')$, for some $e' \in [\![\texttt{Exp}]\!]$. That is, it is an element of $[\![\rho\,!\,\mathbf{1}]\!]$.

Case $\dfrac{\Gamma \vdash e_1 : (\rho_2 \xrightarrow{t} \rho_1, b)\,!\,t_1 \qquad \Gamma \vdash e_2 : \rho_2\,!\,t_2}{\Gamma \vdash e_1 \texttt{@}^b e_2 : \rho_1\,!\,t_1 \gg t_2 \gg t}$. Again, there are two subcases.

Subcase $b = s$. In this case, $[\![\mathcal{T}[\![e_1 \texttt{@}^s e_2]\!]]\!]\varphi = [\![\mathcal{T}[\![e_1]\!] \texttt{>>=} \lambda f.\mathcal{T}[\![e_2]\!] \texttt{ >>= } \lambda v.f \texttt{@} v]\!]\varphi$. By induction $[\![\mathcal{T}[\![e_1]\!]]\!]\varphi \in [\![(\rho_2 \xrightarrow{t} \rho_1, s)\,!\,t_1]\!]$ so that $f \in [\![(\rho_2 \xrightarrow{t} \rho_1, s)]\!] = \{f \mid (\forall v \in [\![\rho_2]\!])\ f\ v \in [\![\rho_1\,!\,t]\!]\}$. Also, by induction, $[\![\mathcal{T}[\![e_2]\!]]\!]\varphi \in [\![\rho_2\,!\,t_2]\!]$ so that $v \in [\![\rho_2]\!]$ by definition of >>=. Hence, $[\![f \texttt{@} v]\!]\varphi[f \ldots, v \ldots] \in [\![\rho_1\,!\,t]\!]$. An auxiliary induction on t_2 shows that $[\![\mathcal{T}[\![e_2]\!] \texttt{>>=} \lambda v.f \texttt{@} v]\!]\varphi[f \ldots] \in [\![\rho_1\,!\,t_2 \gg t]\!]$ and another auxiliary induction on t_1 shows the claim $[\![\mathcal{T}[\![e_1]\!] \texttt{ >>= } \lambda f.\mathcal{T}[\![e_2]\!] \texttt{ >>= } \lambda v.f \texttt{@} v]\!]\varphi \in [\![\rho_1\,!\,t_1 \gg t_2 \gg t]\!]$.

Subcase $b = d$. In this case, $[\![\mathcal{T}[\![e_1 \texttt{@}^d e_2]\!]]\!]\varphi = [\![\mathcal{T}[\![e_1]\!] \texttt{>>=} \lambda e_1'.\mathcal{T}[\![e_2]\!] \texttt{ >>= } \lambda e_2'.\texttt{return } (e_1' \texttt{@}^r e_2')]\!]\varphi$. By Lemma 2, $\vdash_{wft} (\rho_2 \xrightarrow{t} \rho_1, d)$ so that $t = \mathbf{1}$, $\rho_2 = (\tau_2, d)$, and $\rho_1 = (\tau_1, d)$. By induction, $[\![\mathcal{T}[\![e_1]\!]]\!]\varphi \in [\![(\rho_2 \xrightarrow{t} \rho_1, d)!t_1]\!]$, which means (by definition of >>=) that $e_1' \in [\![\texttt{Exp}]\!]$. Also by induction, $[\![\mathcal{T}[\![e_2]\!]]\!]\varphi \in [\![\rho_2\,!\,t_2]\!]$ which means (by definition of >>= and since $\rho_2 = (\tau_2, d)$) that $e_2' \in [\![\texttt{Exp}]\!]$. Hence, $[\![\texttt{return } (e_1' \texttt{@}^r e_2')]\!]\varphi[e_1' \ldots e_2' \ldots] \in [\![\rho_1\,!\,\mathbf{1}]\!] = [\![\rho_1\,!\,t]\!]$ and two auxiliary inductions on t_2 and t_1 yield $[\![\mathcal{T}[\![e_1]\!] \texttt{ >>= } \lambda e_1'.\mathcal{T}[\![e_2]\!] \texttt{ >>= } \lambda e_2'.\texttt{return } (e_1' \texttt{@}^r e_2')]\!]\varphi \in [\![\rho_1\,!\,t_1 \gg t_2 \gg t]\!]$, as required.

Case $\dfrac{\Gamma \vdash e_1 : (\tau, s) \,!\, t_1 \quad \Gamma, x : (\tau, s) \vdash e_2 : \rho_2 \,!\, t_2}{\Gamma \vdash \texttt{let}^s\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \rho_2 \,!\, t_1 \gg t_2}$. Analogous to considering $(\lambda^s x.e_2)\texttt{@}^s e_1$.

Case $\dfrac{\Gamma \vdash e_1 : (\tau, d) \,!\, t_1 \quad \Gamma, x : (\tau, d) \vdash e_2 : \rho_2 \,!\, t_2}{\Gamma \vdash \texttt{let}^d\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \rho_2 \,!\, t_1 \gg l(t_2)}$. Consider $[\![\mathcal{T}[\![\texttt{let}^d\ x\ \texttt{=}\ e_1\ \texttt{in}\, e_2]\!]]\!]\varphi = [\![\mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda e_1'.\texttt{newVar}\ \texttt{>>=}\ \lambda x.\texttt{mlet}\ x\ e_1'\ \mathcal{T}[\![e_2]\!]]\!]\varphi$. By induction, $[\![\mathcal{T}[\![e_1]\!]]\!]\varphi \in [\![(\tau, d) \,!\, t_1]\!]$ so that (by def of $\texttt{>>=}$) $e_1' \in [\![\texttt{Exp}]\!]$. By assumption, $\texttt{newVar}$ is equal to $\texttt{return x}$, for some fresh variable $\texttt{x}$. Since $\texttt{VarExp x} \in [\![(\tau, d)]\!]$, induction yields that $[\![\mathcal{T}[\![e_2]\!]]\!]\varphi[e_1' \ldots, x \mapsto \texttt{VarExp x}] \in [\![\rho_2 \,!\, t_2]\!]$. By definition of $\texttt{mlet}$, $[\![\texttt{mlet}\ x\ e_1'\ \mathcal{T}[\![e_2]\!]]\!]\varphi[e_1' \ldots, x \ldots] = \texttt{L x e}_1'\ v$, for some $v \in [\![\rho_2 \,!\, t_2]\!]$. By definition that means $\texttt{L x e}_1'\ v \in [\![\rho_2 \,!\, l(t_2)]\!]$. An auxiliary induction on t_1 establishes that $[\![\mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda e_1'.\texttt{newVar}\ \texttt{>>=}\ \lambda x.\texttt{mlet}\ x\ e_1'\ \mathcal{T}[\![e_2]\!]]\!]\varphi \in [\![\rho_2 \,!\, t_1 \gg l(t_2)]\!]$.

Case $\dfrac{\Gamma \vdash e_1 : (\texttt{int}, s) \,!\, t_1 \quad \Gamma \vdash e_2 : \rho \,!\, t_2 \quad \Gamma \vdash e_3 : \rho \,!\, t_3}{\Gamma \vdash \texttt{if0}^s\ e_1\ e_2\ e_3 : \rho \,!\, t_1 \gg (t_2 \sqcup t_3)}$. Consider $[\![\mathcal{T}[\![\texttt{if0}^s\, e_1\, e_2\, e_3]\!]]\!]\varphi = [\![\mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda v_1.\texttt{if0}\ v_1\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!]]\!]\varphi$. By induction $[\![\mathcal{T}[\![e_1]\!]]\!]\varphi \in [\![(\texttt{int}, s) \,!\, t_1]\!]$ so that $v_1 \in [\![(\texttt{int}, s)]\!]$.

Subcase $v_1 = 0$. In that case $[\![\texttt{if0}\ v_1\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!]]\!]\varphi[x \mapsto 0] = [\![\mathcal{T}[\![e_2]\!]]\!]\varphi$. By induction, $[\![\mathcal{T}[\![e_2]\!]]\!]\varphi \in [\![\rho!t_2]\!]$, hence so is $[\![\texttt{if0}\ v_1\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!]]\!]\varphi[x \mapsto 0] \in [\![\rho!t_2]\!]$. Since $[\![\rho!t_2]\!] \subseteq [\![\rho!t_2 \sqcup t_3]\!]$, it holds that $[\![\texttt{if0}\ v_1\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!]]\!]\varphi[x \mapsto 0] \in [\![\rho!t_2 \sqcup t_3]\!]$, too. As usual, an auxiliary inducition on t_1 and the definition of $\texttt{>>=}$ establish that $[\![\mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda v_1.\texttt{if0}\ v_1\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!]]\!]\varphi \in [\![\rho \,!\, t_1 \gg (t_2 \sqcup t_3)]\!]$.

Subcase $v_1 = v_1' + 1$. Analogous.

Case $\dfrac{\Gamma \vdash e_1 : (\tau, d) \,!\, t_1 \quad \Gamma \vdash e_2 : \rho \,!\, t_2 \quad \Gamma \vdash e_3 : \rho \,!\, t_3}{\Gamma \vdash \texttt{if0}^d\ e_1\ e_2\ e_3 : \rho \,!\, t_1 \gg i(t_2, t_3)}$. Consider $[\![\mathcal{T}[\![\texttt{if0}^d\, e_1\, e_2\, e_3]\!]]\!]\varphi = [\![\mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda e_1'.\texttt{mif}\ e_1'\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!]]\!]\varphi$. By induction, $[\![\mathcal{T}[\![e_1]\!]]\!]\varphi \in [\![(\tau, d) \,!\, t_1]\!]$ so that $e_1' \in [\![(\tau, d)]\!]$. Further induction yields $[\![\mathcal{T}[\![e_2]\!]]\!]\varphi \in [\![\rho \,!\, t_2]\!]$ and $[\![\mathcal{T}[\![e_3]\!]]\!]\varphi \in [\![\rho \,!\, t_3]\!]$. Hence $[\![\texttt{mif}\ e_1'\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!]]\!]\varphi[e_1' \ldots] = \texttt{I}e_1'\ v_2\ v_3$ where $v_2 \in [\![\rho \,!\, t_2]\!]$ and $v_2 \in [\![\rho \,!\, t_3]\!]$. Clearly, $\texttt{I}e_1'\ v_2\ v_3 \in [\![\rho \,!\, i(t_2, t_3)]\!]$. Finally, an auxiliary induction on t_1 yields that $[\![\mathcal{T}[\![e_1]\!]\ \texttt{>>=}\ \lambda e_1'.\texttt{mif}\ e_1'\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![e_3]\!]]\!]\varphi \in [\![\rho \,!\, t_1 \gg i(t_2, t_3)]\!]$.

Case $\dfrac{\Gamma \vdash e : (\tau, d) \,!\, t}{\Gamma \vdash \texttt{reset}\ e : (\tau, d) \,!\, \mathbf{1}}$. Consider $[\![\mathcal{T}[\![\texttt{reset}\ e]\!]]\!]\varphi = [\![\texttt{mreset}\ \mathcal{T}[\![e]\!]]\!]\varphi$. By induction, $[\![\mathcal{T}[\![e]\!]]\!]\varphi \in [\![(\tau, d) \,!\, t]\!]$. An auxiliary induction on t establishes the claim: Let $m \in [\![(\tau, d) \,!\, t]\!]$, then $\texttt{mreset}\ m \in [\![(\tau, d) \,!\, \mathbf{1}]\!]$.

Subcase $t = \mathbf{1}$. In this case, $m = \texttt{V e}$, for some $\texttt{e} \in [\![\texttt{Exp}]\!]$. By definition of $\texttt{mreset}$, $\texttt{mreset}\ m = \texttt{V e} \in [\![(\tau, d) \,!\, \mathbf{1}]\!]$.

Subcase $t = l(t')$. In this case, $m = \texttt{L x e}\ m'$, for some $m' \in [\![(\tau, d)!t']\!]$. Hence, $\texttt{mreset}\ m = \texttt{mreset (L x e}\ m') = \texttt{mreset}\ m'\ \texttt{>>=}\ \lambda e'.\texttt{return (LetExp x e}\ e')$. By induction, $\texttt{mreset}\ m' \in [\![(\tau, d) \,!\, \mathbf{1}]\!]$ and hence it holds that (by def of $\texttt{>>=}$) $\texttt{mreset}\ m'\ \texttt{>>=}\ \lambda e'.\texttt{return (LetExp x e}\ e') \in [\![(\tau, d) \,!\, \mathbf{1}]\!]$.

Subcase $t = i(t', t'')$. In this case, $m = \texttt{I e}\ m'\ m''$, for some $m' \in [\![(\tau, d) \,!\, t']\!]$ and $m'' \in [\![(\tau, d)\,!\,t'']\!]$. Hence, $\texttt{mreset}\ m = \texttt{mreset (I e}\ m'\ m'') = \texttt{mreset}\ m'\ \texttt{>>=}\ \lambda e'.\texttt{mreset}\ m''\ \texttt{>>=}\ \lambda e''.\texttt{return (IfExp e}\ e'\ e'')$. By induction, $\texttt{mreset}\ m' \in [\![(\tau, d) \,!\, \mathbf{1}]\!]$ and $\texttt{mreset}\ m'' \in [\![(\tau, d) \,!\, \mathbf{1}]\!]$ and hence it holds that (by def of $\texttt{>>=}$) $\texttt{mreset}\ m'\ \texttt{>>=}\ \lambda e'.\texttt{mreset}\ m''\ \texttt{>>=}\ \lambda e''.\texttt{return (IfExp e}\ e'\ e'') \in [\![(\tau, d) \,!\, \mathbf{1}]\!]$.

Subcase $t = t' \sqcup t''$. In this case, $m \in [\![(\tau, d) \,!\, t']\!]$ or $m \in [\![(\tau, d) \,!\, t'']\!]$. In both cases, induction is applicable and yields that `mreset` $m \in [\![(\tau, d) \,!\, \mathbf{1}]\!]$.