Typing, Representing, and Abstracting Control

Functional Pearl

Philipp Schuster
University of Tübingen
Germany

Jonathan Immanuel Brachthäuser
University of Tübingen
Germany

Abstract

A well known technique to implement programming languages with delimited control operators `shift` and `reset` is to translate programs into continuation passing style (CPS). We can iterate the CPS translation to obtain the CPS hierarchy and to implement a family of control operators $\texttt{shift}_i$ and $\texttt{reset}_i$. This functional pearl retells the story of a family of delimited control operators and their translation to lambda calculus via the CPS hierarchy. Prior work on the CPS hierarchy fixes a level of n control operators for the entire program upfront, but we allow different parts of the program to live at different levels. It turns out that taking `shift0` rather than `shift` as the basis for the family of control operators is essential for this. Our source language is a typed embedding in the dependently typed language Idris. Our target language is a HOAS embedding in Idris. The translation avoids administrative beta- and eta-redexes at all levels of the CPS hierarchy, by iterating well-known techniques for the non-iterated CPS translation.

CCS Concepts • **Software and its engineering → Control structures**; • **Theory of computation** → *Type structures*;

Keywords Delimited Control, Control Effects, Continuation Passing Style, CPS Hierarchy, Compilation

ACM Reference Format:
Philipp Schuster and Jonathan Immanuel Brachthäuser. 2018. Typing, Representing, and Abstracting Control: Functional Pearl. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '18), September 27, 2018, St. Louis, MO, USA*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3240719.3241788

TyDe '18, September 27, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5825-5/18/09…$15.00
https://doi.org/10.1145/3240719.3241788

1 Introduction

Control operators `shift` and `reset` are useful to model complex control flow. A well known technique to implement a programming language with delimited control operators `shift` and `reset` is to translate programs into continuation passing style (CPS) [Danvy and Filinski 1992]. By iterating the CPS translation n times we can implement a family of n control operators $\texttt{shift}_i$ and $\texttt{reset}_i$. While the original introduction of the CPS hierarchy by Danvy and Filinski [1990] works with an untyped language, in this paper we present an implementation of the CPS hierarchy as a typed embedding in a dependently typed language. We index effectful terms by the list of answer types. Each type in this list corresponds to one level of the CPS hierarchy. We also use types to distinguish between two stages: static and dynamic.

By the end of the paper we will be able to write programs in a typed language using a family of control operators and generate code in a language that supports first-class functions. We extend our source language with primitives, `ifThenElse` and `letrec` and show how they interact nicely with our embedding. To get there, we step-wise reconstruct well known techniques and translate them to Idris and the typed setting. The individual steps are simple and guided by the types.

2 Examples

In this section, we motivate programming with control operators in the CPS hierarchy and give an overview over our source language as an embedding into the dependently typed programming language Idris [Brady 2013].

2.1 Example: Non-Deterministic Programming

We start with an example implementation of the non-deterministic programming statements *fail* and *flip* in terms of the control operator `shift0` in Figure 1, adopted from Danvy and Filinski [1990]. It might be instructive to compare with the original presentation. Both implementations capture the current continuation k with `shift0`. In the implementation of *fail* we never resume it and immediately return the string `"no"` instead. In the implementation of *flip* we resume the continuation with both *True* and *False*. We write our programs in do-notation to sequence effectful statements: Each line after the 'do' is a statement. We explicitly resume continuations with `resume`, which we will

```
fail : Stm (String :: rs) a
fail = shift0 (λk ⇒ do
  pure "no")
```

```
flip : Stm (r :: rs) Bool
flip = shift0 (λk ⇒ do
  resume k True
  resume k False)
```

```
emit : a → Stm (List a :: rs) ()
emit a = shift0 (λk ⇒ do
  as ← resume k ()
  pure (a :: as))
```

Figure 1. Function *fail* aborts the computation, *flip* models non-deterministic choice, and *emit* yields values to the context.

introduce later. The blue overbar and red underbar are staging annotations and can safely be ignored for now – we will explain them later.

The types of both *flip* and *fail* show that they are (effectful) statements Stm. Statements are parametrized by a list of answer types and their immediate result. The list of answer types intuitively corresponds to the computational context that needs to be provided to execute the statement. Every element in the list marks a position of the runtime stack with the expected type. We will use these positions as targets to transfer the control-flow to. The immediate result of *fail* is *a* for all *a*, expressing that *fail* never returns anything (it could be *Void*). To allow aborting the computation with the string `"no"`, *fail* requires the list of answer types to start with *String*. In contrast, *flip* is both polymorphic in its (top-most) answer type *r* and the rest of the answer types *rs* (for "results"). It captures the continuation using shift0 and resumes it twice, discarding the first result of type *r*, only executing both alternatives for the side effects. The immediate result of *flip* is *Bool*, hence the continuation *k* takes a boolean value to resume execution.

Because shift0 is a *delimited* control operator it will capture the continuation only up to the closest delimiter reset0. In consequence, the following example will return the string `"Answer was: no"` and *not* terminate the execution as a whole to return with the string `"no"`.

```
delimitFail : Stm rs String
delimitFail = do
  a ← reset0 fail
  pure ("Answer was: " ++ a)
```

Here we delimit *fail* with reset0. Thus, *fail* will not discard the entire continuation but only up to the closest delimiter reset0. Since reset0 immediately surrounds *fail*, the captured continuation happens to be empty. The variable *a* will be bound to the string `"no"` which is the immediate result of the delimited computation reset0 *fail*. The type of the immediate result fits with the top-most answer type of *fail* whose list of answer types is *String* :: *rs*.

In contrast, the following example will not type check:

```
perhapsFail : Stm (String :: rs) Int
perhapsFail = do
  ifThenElse 18 ≤ 0
    fail
    (pure 9)
```

```
wontTypecheck : Stm rs String
wontTypecheck = do
  a ← reset0 perhapsFail
  pure ("Answer was: " ++ a)
```

It is not clear (to the compiler) whether the computation *perhapsFail* delimited by reset0 will abort with string `"no"` or return normally with integer 9. When we delimit a statement with reset0 the answer type and the immediate result type have to agree.

As another example, consider the effectful function *emit* in Figure 1 that yields a value to the surrounding context. It takes a value to emit as its argument *a*. It uses shift0 to capture the current continuation *k* and resumes *k* with the unit value to get a list of results *as*. Finally, it prepends its argument *a* to the front of the other results *as*. Since we expect resuming the continuation to return a list, the effectful function *emit* only works when the top-most answer type is a list. To collect the list of emitted values we define the function *collect*:

```
collect : Stm (List a :: rs) b → Stm rs (List a)
collect m = reset0 (do
  _ ← m
  pure [])
```

The function *collect* runs a computation that has List *a* as its top-most answer type and an arbitrary result type *b*. It runs the computation, ignores its result and then returns the empty list. It also acts as a delimiter for all calls to shift0 in *m* such as the one in *emit*. In consequence, emit will suspend and resume the computation at the surrounding call to *collect*, prepending the emitted values to the empty list after the delimited continuation returns.

2.2 Example: Collecting Triples

We now present a bigger example also adopted from Danvy and Filinski [1990]. The task is to generate all distinct positive integers *i*, *j* and *k* less than or equal to given integer *n* that sum to a given integer *s*. Our strategy is to use the non-deterministic choice statement *flip* to generate candidate triples and the abortive statement *fail* to filter those that do not have the desired property.

Equipped with the operators *flip* and *fail*, in Figure 2, we implement a recursive function *choice* that non-deterministically chooses an integer between 1 and a given integer *n*. We then use *choice* to implement a function *triple* that

```
choice : Int → Stm (String :: rs) Int
choice = letrec (λrecurse ⇒ λn ⇒ do
  ifThenElse (n < 1)
    fail
    (do
      b ← flip
      ifThenElse b
        (recurse (n − 1))
        (pure n)))
```

```
triple : Int → Int → Stm (String :: rs) (Int, Int, Int)
triple n s = do
  i ← choice n
  j ← choice (i − 1)
  k ← choice (j − 1)
  ifThenElse ((i + (j + k)) == s)
    (pure (i , j , k))
    fail
```

Figure 2. Effectful functions *choice* for non-deterministically choosing an integer between 1 and *n*, and *triple* filtering triples that sum up to *s*.

chooses integers *i*, *j* and *k* and fails if they do not sum up to the given integer *s*. To be able to use *fail*, the types of *choice* and *triple* express that they use control effects with top-most answer type *String*, but are polymorphic in the remaining answer types *rs*.

To gather all triples produced by *triple* into a list, we might want to use *emit* and *collect*. However, doing so, we already notice on the type-level, that the two effects interfere. We cannot have a top-most answer type of both *String* and of List (*Int*, *Int*, *Int*) at the same time. The solution is to introduce a second level of control and use the *emit* operation on that level. We could rewrite *emit* to use a new control operator shift0_1 where it previously used shift0. Instead, here we choose to use the function lift that lifts a computation from one level to the next. We will define it later.

```
emitTriples : Stm (String :: List (Int, Int, Int) :: rs) String
emitTriples = do
  res ← triple 9 15
  lift (emit res)
  pure "done"

emittedTriples : Stm [ ] (List (Int, Int, Int))
emittedTriples = collect (reset0 (emitTriples {rs = [ ]}))
```

In *emittedTriples*, we explicitly choose the list of answer types *rs* to be the empty list [ ]. As a result, *emittedTriples* cannot have any further control effects itself – making it a pure value. The effects of *emitTriples* (that is, those of *choice* and *fail*) are delimited by reset0 and the effects of *emit* are delimited by *collect*.

An alternative to collecting all triples is to abort the computation early and only get the first triple. To this end, we implement an effectful function *first* that, when called with an argument *a*, gets the current continuation, but never resumes it. Instead it immediately returns *Just a*.

```
first : a → Stm (Maybe a :: rs) ()
first a = shift0 (λk ⇒ do
  pure (Just a))
```

Running *triple* with *first* instead of *emit* follows the same pattern as before.

```
firstOfTriples : Stm (String :: Maybe (Int, Int, Int) :: rs) String
firstOfTriples = do
  res ← triple 9 15
  lift (first res)
  pure "done"

firstTriple : Stm [ ] (Maybe (Int, Int, Int))
firstTriple = reset0 (do
  reset0 firstOfTriples
  pure Nothing)
```

The effectful function *first* requires an answer type of *Maybe a*. We can use it on the result of *triple* by lifting it, just like we did before with *emit*. However, we now discard the result of reset0 *firstTriples* and return *Nothing* whenever *triple* fails to emit a value.

Figure 3 shows the result of pretty printing the program generated by *emittedTriples*. All control operators are eliminated and evaluating the expression results in a pure list of triples. Internally, it is specialized to use two levels of CPS internally. The CPS transformation only introduced beta-redexes in between recursive definitions. The resulting program is free of administrative beta- and eta-redexes.

We have shown how to use our embedded language to compose programs with control operators. Since our embedding is typed, we were able to prevent some errors and for instance reject the function *wontTypecheck*. We have shown and discussed the code in CPS that we generate. In the next section we will start building up to this goal with an easy first step.

3 Basics: Continuation Passing Style

Our goal is to embed a language with control operators into the dependently typed language Idris. In this section, we start with the translation of control operators shift0 and reset0 into CPS and show how to enable do-notation for terms in CPS.

```
(let choice1 n = (λk1 ⇒ (λk2 ⇒
  (if (n < 1)
    then k2 "no"
    else choice1 (n − 1) k1 (λx4 ⇒ k1 n k2)))) in choice1) 9 (λx0 ⇒ (λk3 ⇒
(let choice2 n = (λk1 ⇒ (λk2 ⇒
  (if (n < 1)
    then k2 "no"
    else choice2 (n − 1) k1 (λx6 ⇒ k1 n k2)))) in choice2) (x0 − 1) (λx1 ⇒ (λk4 ⇒
(let choice3 n = (λk1 ⇒ (λk2 ⇒
  (if (n < 1)
    then k2 "no"
    else choice3 (n − 1) k1 (λx8 ⇒ k1 n k2)))) in choice3) (x1 − 1) (λx2 ⇒ (λk5 ⇒
    (if ((x0 + (x1 + x2)) ≡ 15)
      then ((x0, x1, x2) :: (k5 "done"))
      else k5 "no"))) k4)) k3)) (λx0 ⇒ [ ])
```

Figure 3. Result of pretty printing the expression `emittedTriples`.

We introduce the following type alias to represent terms in CPS with type `a` and answer type `r`.

```
Cps : Type → Type → Type
Cps r a = (a → r) → r
```

We can also view a term of type `Cps r a` as potentially having control effects up to a delimiter that expects type `r`. In this section, a context from type `a` to type `r` is a function `(a → r)`. We will use the terms context and continuation interchangeably.

Following Materzok and Biernacki [2011], we define the control operator `shift0`. Given a body, it returns a term in CPS. The body takes the captured continuation from immediate result type `a` to answer type `r` and returns `r`.

```
shift0 : ((a → r) → r) → Cps r a
shift0 = id
```

We implement `shift0` as the identity function - it is just a shift in perspective so to say. It is important to note that `shift0` removes the corresponding `reset0` delimiter and that both the continuation and the body of `shift0` have to be pure. Neither of them can have any control effects as those would be undelimited. This is also reflected in the type of the body and the continuation: both return a pure value of type `r`.

The control operator `shift0` also has an inverse that we will call `run0`. It is similar to the `$`-operator from Kiselyov and Shan [2007] but with its arguments swapped. Given a term in CPS and a context, it runs the term in the context, delimiting any control effects the term might have.

```
run0 : Cps r a → (a → r) → r
run0 = id
```

The result of `run0` is a pure value of type `r`, it does not have any control effects. We will later see how `shift0` and `run0` make it very natural to walk up and down the CPS hierarchy. To recover the classical delimiter `reset0` we run the computation in the empty context, which is the identity function.

```
reset0 : Cps r r → r
reset0 m = run0 m id
```

The immediate result type and the answer type `r` need to agree. Again, the type makes it clear that `reset0` delimits all control effects. The result is a pure value.

To translate pure values into CPS we define the function `pure` and to compose terms in CPS we define the function `bind`. The reader might recognize the continuation monad. And indeed, these are the two combinators that allow us to use do-notation for our embedded language.

```
pure : a → Cps r a
pure a = λk ⇒ k a

push : (a → Cps r b) → (b → r) → (a → r)
push f k = λa ⇒ f a k

bind : Cps r a → (a → Cps r b) → Cps r b
bind m f = λk ⇒ m (push f k)
```

The function `pure` calls the current continuation with the given value. We define an auxiliary function `push` to push an effectful function `(a → Cps r b)` onto a context `(b → r)` to get a new context `(a → r)`. We call it "push" to emphasize the analogy between computational contexts and the runtime stack. In `bind` we run the given term in CPS with the given effectful function pushed onto the current continuation.

Let us summarize what we have so far in a small example:

```
example : Int
example = 1 + reset0 (do
  x ← shift0 (λk ⇒ k (k 100))
  pure (10 + x))
```

The example evaluates to `121`. With `reset0` we delimit the effectful term to get a value of type *Int* and add the value `1`. In the argument of `reset0` we use our control operator `shift0` to capture the current continuation as `k` and apply it twice to the value `100`. Like all the continuations that we capture with `shift0`, `k` is pure. We bind the result of `shift0` to `x` and return this result after adding `10`. Again, the type of *example* tells us that it is pure i.e. does not have any control effects observable from the outside. All side effects have been encapsulated.

4 Representing Control – Staging CPS Expressions

In the previous section, we embedded the control operators `shift0` and `run0` into our meta language Idris. Now we want to reify terms in the embedded language into our target language: a typed embedding of lambda calculus. In section 7, we will then extend the target language with primitive operations, `letrec` and `ifThenElse`. The data type $\underline{\cdot}$ represents an expression in lambda calculus in HOAS [Pfenning and Elliot 1988] which simplifies the implementation considerably. This was observed before in a similar setting by Thiemann [1996].

```
data _ : Type → Type  where
  λ : (a → b) → a → b
  @ : a → b → a → b
```

We write type applications of data type $\underline{\cdot}$ with a red underbar, as in $\underline{a}$, to represent a target language expression of type a. Additionally, we write the one-argument constructor $\underline{\lambda}$ prefix and the two-argument constructor $\underline{@}$ infix. For example, a term in the target language with function type has type $\underline{a \to b}$, but a function in the meta language between expressions in the target language has type $\underline{a} \to \underline{b}$.

We follow Danvy and Filinski [1992] and add staging annotations to the type of terms in CPS, that we have introduced in section 3:

```
Cps : Type → Type → Type
Cps r a = (a → r) → r
```

We have two stages: present and future. We also call the present stage *static* and the future stage *dynamic* and use types to distinguish terms in different stages. Present-stage values of type `a` have type `a` and future stages values of type `a` have type $\underline{a}$.

Notational Conventions The naming convention of writing a type in blue and with an overbar (like $\overline{\text{Cps}}$) indicates that we statically know the control flow of a term. In contrast, a red type with an underbar is the type of an expression in the target language, which means that the term will only be known dynamically. A black type is a type in the meta language Idris. Similarly, on the term level, we have a naming convention where we use red with an underbar for constructors of target language expressions and black for terms in the meta language Idris.

We are ready to define staged variants of `shift0`, `run0` and `reset0`. The definitions are *exactly* like in section 3, however, using staging annotations we can give them more specific types. The types express the fact that we are composing (dynamic) target language expressions.

```
shift0 : ((a → r) → r) → Cps r a
shift0 = id

run0 : Cps r a → (a → r) → r
run0 = id

reset0 : Cps a a → a
reset0 m = run0 m id
```

To compose programs in CPS, we define staged variants of `pure` and `bind` with auxiliary function `push` just like in section 3. Again, on the term level they are exactly the same, but we give them more specific types.

```
pure : a → Cps r a
pure a = λk ⇒ k a

push : (a → Cps r b) → (b → r) → (a → r)
push f k = λa ⇒ f a k

bind : Cps r a → (a → Cps r b) → Cps r b
bind m f = λk ⇒ m (push f k)
```

Expressions that are built using the above functions are meta level functions over terms of the target language. Eventually, we want to completely reify such expressions to one expression in the target language. To this end, we define the two symmetric functions `reify` and `reflect`:

```
reify : Cps r a → Cps r a
reify m   = λ λk ⇒ m (λa ⇒ k @ a)

reflect : Cps r a → Cps r a
reflect m = λk ⇒ m @ (λ λa ⇒ k a)
```

We naturally avoid any administrative beta redexes that would have to be post-reduced as explained by Danvy and Filinski [1992]. To translate a term in a dynamic context `k`, the function `reify` takes a term where the control flow is known statically but values are only known dynamically and produces a term in the target language. The function `reflect` takes a term in the target language and makes it possible to use it in the source language.

Given the previous definitions, we can now embed application and abstraction. Application takes a term in CPS whose result is an effectful function and a term in CPS whose result is a value of type `a` and applies the function to the value. This makes it necessary to first evaluate both the function and the value and finally reflect the result of the application. To embed lambda abstractions we take a function (again

in HOAS) and return an abstraction in the target language where we reify the body of the function applied to the argument.

```
app : Cps r (a → Cps r b) → Cps r a → Cps r b
app mf ma = do
  f ← mf
  a ← ma
  reflect (f @ a)

lam : (a → Cps r b) → Cps r (a → Cps r b)
lam f = pure (λ λa ⇒ reify (f a))
```

These definitions coincide with the ones by Danvy and Filinski [1992]. All we did was to inline the translation meta-function and factor parts into reusable combinators.

Let's consider the example term from section 3, but with the types presented in this section.

```
example : Int
example = 1 + reset0 (do
  x ← shift0 (λk ⇒ k (k 100))
  pure 10 + x)
```

When we pretty print the expression `example` we get the following:

```
(1 + (10 + (10 + 100)))
```

To make this example more interesting, let's abstract the term with the call to `shift0` into its own function `resumeTwice`.

```
resumeTwice : Cps Int (Int → Cps Int Int)
resumeTwice = lam (λn ⇒ shift0 (λk ⇒ k (k n)))

example' : Int
example' = 1 + reset0 (do
  x ← app resumeTwice (pure 100)
  pure (10 + x))
```

Pretty printing this term now yields:

```
(1 + (λn ⇒ (λk ⇒ (k (k n)))) 100 (λx ⇒ (10 + x)))
```

With `lam` we reify an effectful function into the target language. To apply it in the source language, we have to reflect it. This introduces beta-redexes.

In this section we refined the definitions of section 3 by giving more specific types and thereby adding staging annotations. We reify terms written in CPS in the source language into terms in CPS in the target language. The generated lambda expressions are free of administrative beta-redexes unless we specifically ask for a term to be reified and reflect it later. In the next section, we will again refine the basic definitions of section 3, but this time in a different way.

5 Abstracting Control – The CPS Hierarchy

In this section, we will explore a second variation of the basic definitions of section 3. Instead of adding staging annotations, this time we follow Danvy and Filinski [1990] and iterate the CPS translation to obtain a hierarchy of control operators. Later in section 6, we will combine the extensions of this and the previous section. The present section only uses terms in the meta language and therefore you will not see any colors.

The CPS hierarchy [Biernacka et al. 2011; Danvy and Filinski 1990; Kameyama 2004; Materzok and Biernacki 2012] allows us to use different control effects in the same program. We can obtain the CPS hierarchy by iterating the CPS transformation. Since the CPS transformation transforms both types and terms, we will have to iterate it on both types and terms. As a consequence, we will have multiple answer types, one for each iteration of the CPS transformation. Concretely this means that the answer type of the first CPS transformation is again a term in CPS, whose answer type is then again in CPS and so on. For example using our definition of `Cps` from section 3, the type of a term of type `a` in CPS whose answer types are `p`, `q` and `r` would have type `Cps (Cps (Cps r q) p) a`. As a shorthand, Figure 4c defines the type of statements `Stm rs a` with a list of answer types `rs` and an immediate result of type `a`. Here we can see the power of using dependent types. We index statements by a list and thus statically track all intermediate answer types on the type level. We type members of the CPS hierarchy of level `n` at type `Stm rs a` for a list of answer types with length `n`. A statement with an empty list of answer types cannot have any control effects and is a pure value. Because different parts of a program can have different lists of answer types this will allow us to avoid CPS transforming sub-programs when it is unnecessary.

Again, Figure 4d implements `shift0`, `run0` and `push` *exactly* like in section 3. We just give them more specific types, replacing `r` by `Stm rs r`. However, for `pure` and `bind` (Figure 4a) we now have two cases to consider: one where we have an empty list of answer types and one where we have a non-empty list of answer types. For an empty list of answer types i.e. pure expressions, in `pure` we directly return the given value and in `bind` we apply the second argument to the first. For a non-empty list of answer types the implementation is exactly the one in section 3. Furthermore, `reset0` does not use the identity function as empty context, but instead resets with `pure` which corresponds to θ by Danvy and Filinski [1990].

The definitions given so far only work with the top-most answer type, but we want a hierarchy of control operators. While the CPS hierarchy is usually used to implement a family of control operators based on `shift` and `reset`, in this paper we use it to construct a family of control operators

```
pure : a → Stm rs a
pure[] a = a
pure_{r::rs} a = λk ⇒ k a
push : (a → Stm (r :: rs) b) → (b → Stm rs r) → (a → Stm rs r)
push f k = λa ⇒ f a k
bind : Stm rs a → (a → Stm rs b) → Stm rs b
bind[] m f = f m
bind_{r::rs} m f = λk ⇒ m (push f k)
```

(a) Iterated variant of monadic operations.

```
lift : Stm rs a → Stm (r :: rs) a
lift = bind
```

(b) Lift operation to move between layers of the hierarchy.

```
Stm : List Type → Type → Type
Stm [] a = a
Stm (r :: rs) a = Cps (Stm rs r) a
```

(c) Type of effectful statements, indexed by intermediate answer types.

```
shift0 : ((a → Stm rs r) → Stm rs r) → Stm (r :: rs) a
shift0 = id
run0 : Stm (r :: rs) a → (a → Stm rs r) → Stm rs r
run0 = id
reset0 : Stm (a :: rs) a → Stm rs a
reset0 m = run0 m pure
```

(d) Iterated variant of control operations

Figure 4. Iterated variant of a language with control operators.

$\mathsf{shift0}_0$, $\mathsf{shift0}_1$, $\mathsf{shift0}_2$, etc. based on shift0 and reset0. Rather than defining them directly we will first define a useful function lift (Figure 4b) that lifts any statement with answer types rs into a larger context with one more answer type $r :: rs$. Its implementation is just bind and the reader might be surprised that the types just happen to match.

We obtain a family of shifts by iterating the lifting:

```
shift0_0 : ((a → Stm rs r) → Stm rs r) → Stm (r :: rs) a
shift0_0 = shift0
shift0_1 : ((a → Stm rs r) → Stm rs r) → Stm (q :: r :: rs) a
shift0_1 = lift ∘ shift0
shift0_2 : ((a → Stm rs r) → Stm rs r) → Stm (p :: q :: r :: rs) a
shift0_2 = lift ∘ lift ∘ shift0
```

The body of each $\mathsf{shift0}_i$ has the same type, since the control operator $\mathsf{shift0}_i$ removes $i + 1$ delimiters. It thus can only make use of control effects outside of the $i + 1$ th delimiter. This also becomes visible in the result type: the answer type r has to match the answer type at the corresponding level of the outer computation. For example in $\mathsf{shift0}_1$, the answer type r occurs in the second position in the list of answer types $q :: r :: rs$.

Similarly, we can iterate reset0 to obtain a family of resets that delimits multiple levels of control effects at once.

```
reset0_1 : Stm (a :: a :: rs) a → Stm rs a
reset0_1 = reset0 ∘ reset0
```

All answer types have to agree.

Equipped with lift and reset0 we can recover the classical shift.

```
shift : ((a → Stm (r::rs) r) → Stm (r::rs) r) → Stm (r::rs) a
shift body = shift0 (λk ⇒ reset0 (body (lift ∘ k)))
```

The difference shows in the type signature for shift where the body and continuation live at the same level of the hierarchy as the rest. To implement shift, we capture the continuation with shift0, but delimit the body with reset0. Since the body now can have the same control effects, we also lift the continuation before passing it to the body.

An example of using multiple levels of control effects is the *partition* function.Given an integer a, it partitions a list of integers into two lists: one containing all integers less than a and on containing all integers greater or equal to a. We use *emit* on two levels of the hierarchy to emit values to the respective partition.

```
partition : Int → List Int → Stm [List Int, List Int] ()
partition a l = case l of
  [] ⇒ do
    pure ()
  (h :: t) ⇒ if (a < h)
    then (do
      emit h
      partition a t)
    else (do
      lift (emit h)
      partition a t)
```

Having seen how to walk down the hierarchy with reset0 and walk it up with lift, in the next section we will combine section 4 and section 5 in order to reify terms in the CPS hierarchy.

6 Representing and Abstracting Control

Combining the two variations of the previous two sections, we add staging annotations to the CPS hierarchy. We use the same definition of expressions as before. Our type of statements with staging annotations now marks the immediate result as well as all intermediate answer types as dynamic by wrapping them in $\cdot$ (Figure 5c). When compared to section 5, for the monadic operations (Figure 5a), control operators (Figure 5d) and lift (Figure 5b) we only change the types to

```
pure : a → Stm rs a
pure[] a = a
pure_{r::rs} a = λk ⇒ k a

push : (a → Stm (r :: rs) b) → (b → Stm rs r) → (a → Stm rs r)
push f k = λa ⇒ f a k

bind : Stm rs a → (a → Stm rs b) → Stm rs b
bind[] m f = f m
bind_{r::rs} m f = λk ⇒ m (push f k)
```

(a) Iterated and staged variant of monadic operations.

```
lift : Stm rs a → Stm (r :: rs) a
lift = bind
```

(b) Lift operation to move between layers of the hierarchy.

```
Stm : List Type → Type → Type
Stm [] a = a
Stm (r :: rs) a = (a → Stm rs r) → Stm rs r
```

(c) Type of effectful statements, indexed by intermediate answer types.

```
shift0 : ((a → Stm rs r) → Stm rs r) → Stm (r :: rs) a
shift0 = id

run0 : Stm (r :: rs) a → (a → Stm rs r) → Stm rs r
run0 = id

reset0 : Stm (a :: rs) a → Stm rs a
reset0 m = run0 m pure
```

(d) Iterated and staged variant of control operations

Figure 5. Iterated and staged variant of a language with control operators.

be more specific – the implementation is exactly the same. All abstractions and applications that we introduce are on the meta level which means in using these functions we do not generate any beta redexes in the target language. The control flow in the CPS hierarchy is completely static.

Just like in section 4, we want to reify terms. The only difference is, that the source level terms live in the CPS hierarchy at an arbitrary level.

```
mutual
  reify : Stm rs a → Stm rs a
  reify[] m = m
  reify_{q::qs} m = λ λk ⇒ reify (m (λa ⇒ reflect (k @ a)))

  reflect : Stm rs a → Stm rs a
  reflect[] m = m
  reflect_{q::qs} m = λk ⇒ reflect (m @ (λ λa ⇒ reify (k a)))
```

The functions `reify` and `reflect` are mutually recursive. To reify a pure statement we don't need to do anything. It already is an expression in the target language. If we reify a statement with at least one answer type, we build an abstraction for the continuation and recursively reify the body after passing the reflected continuation to it. The body is one level lower in the hierarchy. Symmetrically, to reflect a pure expression into a statement without any answer types we don't have to do anything. If the expression has at least one answer type, we abstract the continuation and reflect the body after passing the reified continuation to it.

For example, let's define a function that emits a value on two levels of control.

```
emitTwice : Int → Stm (List Int :: List Int :: rs) ()
emitTwice a = do
  emit a
  lift (emit a)
```

We have made `emitTwice` polymorphic in the rest of the list of answer types `rs`. In choosing `rs` before reification we choose the level of the CPS hierarchy the reified term is in. For example, when we choose `rs` to be the empty list `[]`, reify and pretty print the function `emitTwice` we get:

```
(λa ⇒ (λk1 ⇒ (λk2 ⇒
  (a :: (k1 () (λas ⇒ (k2 (a :: as))))))))
```

If we choose `rs` to be the singleton list containing type unit `[()]`, we obtain one more level of CPS:

```
(λa ⇒ (λk1 ⇒ (λk2 ⇒ (λk3 ⇒
  (k1 () (λas ⇒ (λk4 ⇒
    (k2 (a :: as) (λx1 ⇒ k4 x1)))) (λas ⇒
      (k3 (a :: as)))))))))
```

With the abstraction `(λx1 ⇒ k4 x1)`, we have introduced an eta-redex that when reduced exposes another eta-redex. While we could post-reduce those eta-redexes, it is advisable to not generate them in the first place.

We have shown how to combine the treatments of section 4 and section 5 to take terms in our source language that use control operators at different levels and generate terms in our target language in the CPS hierarchy. These generated terms might contain many eta-redexes. This is particularly severe, since the size of fully expanded types in the CPS hierarchy is exponential in the number of levels. In the next section we will show how to avoid these extra eta-redexes.

7 Preventing Eta-Redexes

In the previous section, we generated code for terms in the CPS hierarchy. But the code generated for statements with multiple answer types contains eta-redexes as observed by Danvy and Filinski [1992]. Danvy and Filinski also propose a solution to this problem: they distinguish whether the context is dynamic or static, avoiding unnecessary reflection

and later reification of an already dynamic context. We can easily translate their solution to the iterated setting by distinguishing for every context in the CPS hierarchy whether it is static or dynamic. This will be our final version of the language.

In Figure 6c, we introduce a type of contexts `Ctx rs a b` that corresponds to an effectful function `a → Stm rs b` from `a` to `b` with answer types `rs`. The two variants of type `Ctx` let us statically distinguish between static and dynamic contexts. A static context is an effectful function, like in section 6. A dynamic context is an expression in the target language of type `a → Stm rs b` where `Stm` is from section 5. A statement now takes a context instead of an effectful function as its continuation. This makes our type of statements and our type of contexts mutually recursive.

With the distinction between static and dynamic context, `reify` and `reflect` are more complicated and use an auxiliary function `reifyContext`. Their definition is translated from [Danvy and Filinski 1992], but where they have two mutually recursive functions that do the translation, we have two cases in `reifyContext`: one for static and one for dynamic continuations.

```
mutual
  reify : Stm rs a → Stm rs a
  reify[]    m = m
  reify(q::qs) m = λ λk ⇒ reify (m (Dynamic k))

  reflect : Stm rs a → Stm rs a
  reflect[]    m = m
  reflect(q::qs) m = λk ⇒ reflect (m @ (reifyContext k))

  reifyContext : Ctx rs a r → a → Stm rs r
  reifyContext (Static k)  = λ λa ⇒ reify (k a)
  reifyContext (Dynamic k) = k
```

While in section 6, in `reify` we reflected the context before passing it to the statement `m`, we now wrap it in the `Dynamic` constructor. In `reflect`, where we previously generated a lambda term to reify the context, we now call the function `reifyContext` instead. This will avoid generating a lambda abstraction when the context is dynamic.

In previous sections, continuations were functions and therefore we could apply them to a value to run them. Now we need a function `resume` to run a continuation with a given value.

```
resume : Ctx rs a r → (a → Stm rs r)
resume (Static k) = k
resume (Dynamic k) = λa ⇒ reflect (k @ a)
```

Symmetrical to `reifyContext`, in `resume` we reflect the context when it is dynamic but do nothing if it is static.

Once again, the definitions for `shift0` and `run0` do not change (Figure 6d). Their types express that they now operate on a contexts. In `pure`, `push` and `bind` (Figure 6a) we now have to resume the continuation with `resume` instead of directly calling it. In `reset0`, we reset the computation into a context that is statically known to be `pure`. Similarly, `push` creates a static context.

With this definition of `bind` we can't use do-notation anymore. So we define `(>>=)` to call bind with its second argument wrapped in the `Static` constructor.

```
(>>=) : Stm rs a → (a → Stm rs b) → Stm rs b
m >>= f = bind m (Static f)
```

Monadic composition in do-notation uses static contexts and still does not produce beta-redexes. In `reify` and `reflect` we take advantage of our ability to also pass dynamic contexts to statements to avoid eta-redexes.

7.1 Primitives, Branching and Recursion

Using this final version of our language, we now show how to add primitives, `ifThenElse` and `letrec`. In section 4, we defined `lam` and `app` in the same way as Danvy and Filinski [1992]. However, it turns out that those definitions don't fit nicely with the rest of the language. We rather propose the following more symmetrical definitions.

```
app : a → Stm rs b → (a → Stm rs b)
app f = λa ⇒ reflect (f @ a)

lam : (a → Stm rs b) → a → Stm rs b
lam f = λ λa ⇒ reify (f a)
```

Here, the definition of `app` does not take statements that evaluate to the function and the argument, respectively. Instead, it assumes they are already values and simply reflects the application. Likewise, the definition of `lam` does not return a statement that then evaluates to the created lambda abstraction. Instead, the result of `lam` is immediately the reified lambda abstration.

In similar spirit, we do not CPS transform pure primitives like `· + ·`. While such a translation is possible as the following shows, we rather ask the user to explicitly use do-notation and bind.

```
add : Stm rs Int → Stm rs Int → Stm rs Int
add mx my = do
  x ← mx
  y ← my
  pure (x + y)
```

This helps us to exploit the type-level distinction between pure expressions and effectful statements.

Assuming the target language has a primitive `Ite` of type `Bool → a → a → a`, we define `ifThenElse` for statements with an arbitrary list of answer types.

```
ifThenElse : Bool → Stm rs a → Stm rs a → Stm rs a
ifThenElse[]    b t e = Ite b t e
ifThenElse(q::qs) b t e = λk ⇒ ifThenElse b (t k) (e k)
```

```
pure : a → Stm rs a
pure[] a = a
pure_{r::rs} a = λk ⇒ resume k a

push : Ctx (r :: rs) a b → Ctx rs b r → Ctx rs a r
push f k = Static (λa ⇒ resume f a k)

bind : Stm rs a → Ctx rs a b → Stm rs b
bind[] m f = resume f m
bind_{r::rs} m f = λk ⇒ m (push f k)
```

(a) Iterated and staged variant of monadic operations.

```
lift : Stm rs a → Stm (r :: rs) a
lift = bind
```

(b) Lift operation to move between layers of the hierarchy.

```
mutual
  Stm : List Type → Type → Type
  Stm [] a = a
  Stm (r :: rs) a = Ctx rs a r → Stm rs r

  data Ctx : List Type → Type → Type → Type where
    Static :    (a → Stm rs b) → Ctx rs a b
    Dynamic : a → Stm rs b     → Ctx rs a b
```

(c) Mutually recursive types of statements and contexts.

```
shift0 : (Ctx rs a r → Stm rs r) → Stm (r :: rs) a
shift0 = id

run0 : Stm (r :: rs) a → Ctx rs a r → Stm rs r
run0 = id

reset0 : Stm (a :: rs) a → Stm rs a
reset0 m = run0 m (Static pure)
```

(d) Iterated and staged variant of control operations with opt. eta-redexes.

Figure 6. Iterated and staged variant of a language with control operators avoiding eta-redexes.

In the case where the list of answer types, we are already operating on values and can directly use the target language's `Ite`. In the other case, we abstract over the current context and recurse, passing the context to both the then-branch and the else-branch. This duplicates the static context and might result in a blow up of the size of generated code. To avoid this duplication we can reify the context, give it a name and pass the reflected name to the two branches. Also, to generate the code in Figure 3 we performed a bit of partial evaluation (not shown here) to avoid generating a call to `ifThenElse`, when the dynamic condition happens to be statically known to be `True` or `False`.

To implement `letrec`, we assume the target language has a `Rec` primitive of type $(a \to b \to a \to b) \to a \to b$.

```
letrec : ((a → Stm rs b) → a → Stm rs b) → a → Stm rs b
letrec body = λa ⇒ reflect (Rec (λf ⇒ λx ⇒
  reify (body (λy ⇒ reflect (f @ y)) x)) @ a)
```

Granted, the implementation of `letrec` is rather complicated, but luckily we have the types to guide us with the insertion of staging annotations. The outermost call to `reflect` allows us to use `letrec` with statically known control flow. We necessarily need to reify the body in order to pass it to `Rec`. Finally, the innermost `reflect` call reflects the recursive application to be used by `body`.

To summarize, in this section we have introduced a new type of contexts that allows us to statically distinguish between static and dynamic contexts. This helps us to avoid eta-redexes in the generated code. We have shown how to add primitives, `ifThenElse` and `letrec` to the source language, given that the target language has corresponding features.

8 Related Work

The amount of work on delimited continuations and on two-stage lambda calculus is vast, therefore we only compare the most closely related publications.

We base our basic control operator `shift0` and its CPS translation on prior work by Materzok and Biernacki [2011] who introduce a type system with subtyping, type inference and implicit coercions. In contrast, we require users to explicitly use `lift`. Additionally, they support answer type modification while we do not. Indexing effectful terms by a list of answer types is not sufficient to support answer type modification. Moving from the list index to a tree structure similar to the annotations by Materzok and Biernacki might remedy this problem.

We implement the CPS hierarchy first presented by Danvy and Filinski [1990]. However, our family of control operators is based on `shift0` instead of `shift`. Danvy and Filinski translate the entire program at a fixed level `n` of the CPS hierarchy, while we allow for different subterms of a program to live on different levels. While they give types to ease understanding, their source language and meta language are untyped. We have types in the source language, use a typed meta language, and show how the two type systems cooperate.

This paper is explicitly based on the classical work by Danvy and Filinski [1992]. The authors explain how to avoid administrative beta- and eta-redexes during a CPS transformation, by writing the translation itself in CPS. They also show the importance of distinguishing static and dynamic application and abstraction. In this paper, we embrace many important insights from their work and extend them to iterated CPS.

9 Limitations and Future Work

We discuss some limitations of our work and how some of them could be addressed in the future.

Our language does not allow for answer type modification, which means that our types rule out for example the `prefixes` function from [Biernacka et al. 2011]. It is possible to rewrite some examples of answer type modification, including this one, to use multiple answer types instead. But we strongly suspect that this is not always possible while statically ruling out runtime errors.

In comparison to type systems that allow for answer type modification and multiple levels of control, for example by Materzok and Biernacki [2012], we have a less complicated type system. In our implementation, answer types form a list instead of a tree. To allow for answer type modification, we would add another type variable to our type of terms in CPS from section 3 and obtain the indexed continuation monad $Cps\ r\ o\ a = (a \to o) \to r$. It is possible that the iterating and staging development from this paper would then still work, but we leave this to future work.

We restrict ourselves to the static (in the control operator sense) control operator `shift0`. Our types make sure that we never capture a continuation when there is no delimiter. Naturally this rules out some programs that we could express in a language or library with *dynamic* delimited control [Dyvbig et al. 2007].

We use polymorphism in the host language Idris to make our statements polymorphic in the tail of the list of answer types `rs`. It would be desirable to reify this polymorphism into the target language. One approach could be to reify the functions that do the lifting.

The size of the types of our statements, when fully expanded, is exponential in the number of levels. If we want to target a typed language we might have to share types with some kind of type-level let-insertion.

We could try to reify some levels but not others to implement a language with control effects, and also use control effects on the meta level. We imagine that our type of statements would have two lists, one for the static part and one for the dynamic part and we could move individual levels from static to dynamic or back.

10 Conclusion

We have reimplemented and combined `shift0` [Materzok and Biernacki 2011], abstracting control [Danvy and Filinski 1990], and representing control [Danvy and Filinski 1992] in a typed language combining different techniques and tradeoffs to ease the implementation. We hope to inspire the design and implementation of libraries and languages with control effects.

References

Malgorzata Biernacka, Dariusz Biernacki, and Sergueï Lenglet. 2011. Typing Control Operators in the CPS Hierarchy. In *Proceedings of the Conference on Principles and Practice of Declarative Programming*. ACM, New York, NY, USA, 12.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA.

Oliver Danvy and Andrzex Filinski. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391.

R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of Functional Programming* 17, 6 (2007), 687–730.

Yukiyoshi Kameyama. 2004. Axioms for delimited continuations in the CPS hierarchy. In *International Workshop on Computer Science Logic*. Springer, 442–457.

Oleg Kiselyov and Chung-chieh Shan. 2007. A Substructural Type System for Delimited Continuations. In *Typed Lambda Calculi and Applications*, Simona Ronchi Della Rocca (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 223–239.

Marek Materzok and Dariusz Biernacki. 2011. Subtyping Delimited Continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 81–93. https://doi.org/10.1145/2034773.2034786

Marek Materzok and Dariusz Biernacki. 2012. A dynamic interpretation of the CPS hierarchy. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 296–311.

Frank Pfenning and Conal Elliot. 1988. Higher-Order Abstract Syntax. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 199–208. https://doi.org/10.1145/53990.54010

Peter J. Thiemann. 1996. Cogen in six Lines. In *Proceedings of the International Conference on Functional Programming*. ACM, 180–189.