1 — CDC Interpreter

This chapter explores the implementation of an interpreter for CDC. Portions of source code are examined in detail although the full source can be found in the appendix.

1.1 Implementation

Although Peyton-Jones *et al.* implement a language-level module for CDC, we are interested in the intermediate term transformations. Using the step-by-step transformations produced by this interpreter, we can construct and verify the implementations of λ^{try} and $\lambda\mu$ into CDC. Examining transformation steps in full also allows us to derive proofs of soundness and completeness for these translations. For this reason, the interpreter was implemented as a term-rewriting program.

We follow the operational semantics of the system to provide an implementation. This is not necessary and results in an ineffecient implementation. Despite this, it is the simplest approach to implementation and efficiency is not central to producing proofs.

1.1.1 Data structures

There are two data types for representing CDC terms, `Value` and `Expr` (Figure 1.1.1). Values are not evaluated: when a term has been reduced to a value, it has terminated on that value. An expression (`Expr`) is a term that can be evaluated to another term. The only exception is a `Hole` which can take any position an expression can. For this reason, it must be a data constructor for expression types.

The core of the abstract machine is a function from one state to the next. A state is its own data type which corresponds to the tuple from the specification of the semantics of the abstract machine $\langle e,\ D,\ E,\ q\rangle$.

1.1.2 Utility Functions

Some utility functions are simplify the implementation. Informally, these functions behave as follows (see Figure 1.1.2 for implementations details):

```
1   data Value = Var Char
2     | Abs Char Expr
3     | Prompt Int
4     | Seq [Expr]
5     deriving (Show, Eq)
6
7   data Expr = Val Value
8     | App Expr Expr
9     | Hole
10    | PushPrompt Expr Expr
11    | PushSubCont Expr Expr
12    | WithSubCont Expr Expr
13    | NewPrompt
14
15    | Sub Expr Expr Char
16    deriving (Show, Eq)
17
18  data State = State Expr Expr [Expr] Value
19    deriving (Show, Eq)
```

Figure 1.1: Data structures for the CDC interpreter

- `prettify :: Expr -> String` is defined inductively for pretty-printing terms.
- `ret :: Expr -> Expr -> Expr` returns the first expression with any holes filled in by the second expression.
- `composeContexts :: [Expr] -> Expr` takes a sequence of expressions and, starting from the end, fills the hole of each expression with the previous expression. This in effect joins the output of each context with the input of the next context.
- `promptMatch :: Int -> Expr -> Bool` returns true if the second argument is a Prompt with the same value as the first argument
- `splitBefore :: [Expr] -> Int -> [Expr]` returns the sequence of expressions up until (but not including) the prompt matching the second argument.
- `splitAfter :: [Expr] -> Int -> [Expr]` returns the sequence of expressions from (but not including) the prompt matching the second argument.
- `sub :: Expr -> Expr -> Char -> Expr` returns the first expression with all occurences of the third expression replaced by the second

expression. If we name the arguments `sub M V x` then this corresponds to the result of evaluating the substitution notation $M[v/x]$.

1.1.3 Reduction Rules

The heavy lifting of the interpreter is done by the function `eval :: State -> State`. `eval` is defined inductively on the structure of CDC terms. Using pattern-matching, each case of `eval` corresponds directly to at least one of the reduction rules of the CDC abstract machine.

Application

$$\begin{array}{lcll} \langle e\ e', D, E, q\rangle & \rightarrow & \langle e, D[\square\ e'], E, q\rangle & \text{e non-value} \\ \langle v\ e, D, E, q\rangle & \rightarrow & \langle e, D[v\ \square], E, q\rangle & \text{e non-value} \\ \langle (\lambda x.e)\ v, D, E, q\rangle & \rightarrow & \langle e[v/x], D, E, q\rangle & \end{array}$$

The `App e e'` case deals with applications: if both terms are values and the first term is an abstraction of the form λ`x.m`, the dominant term becomes a substitution of `e'` for `x` in `m`. Otherwise, the term that is not a value is made the dominant term and the remainder of the application is added to the current context. If both terms are redexes, the left-most is made the dominant term first. In effect, an application first ensures the left-hand term has been evaluated fully before evaluating the right-hand term.

```
1   eval (State (App e e') d es q) = case e of
2     Val v -> case e' of
3       Val _ -> case v of
4         Abs x m -> State (Sub m e' x) d es q
5         Seq es' -> State (ret (composeContexts es') e') d es q
6         otherwise -> State (App e e') d es q
7       otherwise -> State e' (ret d (App e Hole)) es q
8     otherwise -> State e (ret d (App Hole e')) es q
```

PushPrompt

$$\begin{array}{lcll} \langle pushPrompt\ e\ e', D, E, q\rangle & \rightarrow & \langle e, D[pushPrompt\ \square\ e'], E, q\rangle & \text{e non-value} \\ \langle pushPrompt\ p\ e, D, E, q\rangle & \rightarrow & \langle e, \square, p : D : E, q\rangle & \end{array}$$

The `PushPrompt e e'` case ensures the left term is a value. It then pushes the first argument (a prompt) and the current context onto the stack and makes the second argument the dominant term.

```
1   eval (State (PushPrompt e e') d es q) = case e of
2     Val _ -> State e' Hole (e:d:es) q
3     otherwise -> case d of
4       Hole -> State e (PushPrompt Hole e') es q
5       otherwise -> State e (ret d (PushPrompt Hole e')) es q
```

```
ret :: Expr -> Expr -> Expr
ret d e = case d of
  Hole -> e
  App m n -> App (ret m e) (ret n e)
  Val (Abs x m) -> Val $ Abs x (ret m e)
  PushPrompt m n -> PushPrompt (ret m e) (ret n e)
  WithSubCont m n -> WithSubCont (ret m e) (ret n e)
  PushSubCont m n -> PushSubCont (ret m e) (ret n e)
  otherwise -> d

composeContexts :: [Expr] -> Expr
composeContexts = foldr ret Hole . reverse

sub :: Expr -> Expr -> Char -> Expr
sub m v x = case m of
  Val (Var n) -> if n == x then v else m
  Val (Abs y e) -> Val (Abs y $ sub e v x)
  Val (Prompt p) -> Val (Prompt p)
  App e e' -> App (sub e v x) (sub e' v x)
  NewPrompt -> NewPrompt
  PushPrompt e e' -> PushPrompt (sub e v x) (sub e' v x)
  WithSubCont e e' -> WithSubCont (sub e v x) (sub e' v x)
  PushSubCont e e' -> PushSubCont (sub e v x) (sub e' v x)

promptMatch :: Int -> Expr -> Bool
promptMatch i p = case p of
  (Val (Prompt p')) -> i == p'
  otherwise -> False

splitBefore :: [Expr] -> [Expr]
splitBefore p es = takeWhile (not . promptMatch p) es

splitAfter :: [Expr] -> [Expr]
splitAfter  p es = case length es of
  0 -> []
  otherwise -> tail list
  where list = dropWhile (not . promptMatch p) es
```

Figure 1.2: Utility functions for CDC interpreter

WithSubCont

$$\begin{array}{lcll}
\langle withSubCont\ e\ e', D, E, q\rangle & \rightarrow & \langle e, D[withSubCont\ \square\ e'], E, q\rangle & \text{e non-value} \\
\langle withSubCont\ p\ e, D, E, q\rangle & \rightarrow & \langle e, D[withSubCont\ p\ \square], E, q\rangle & \text{e non-value} \\
\langle withSubCont\ p\ v, D, E, q\rangle & \rightarrow & \langle v(D : E^p_{\uparrow}, \square, E^p_{\downarrow}, q\rangle &
\end{array}$$

The reduction rules for `WithSubCont e e'` ensure that the first argument has been evaluated to a prompt `p` and then that the second argument has been evaluated to an abstraction. Finally, it appends the current continuation to the sequence yielded by splitting the continuation stack at `p`, and creates an application of the second argument to this sequence.

```
1    eval (State (WithSubCont e e') d es q) =
2      case e of
3        Val v -> case e' of
4          Val _ -> case v of (Prompt p) ->
5            State (App e' (seq' (d:beforeP))) Hole afterP q
6              where beforeP = splitBefore p es
7                    afterP = splitAfter p es
8          otherwise -> State e' (ret d (WithSubCont e Hole)) es q
9        otherwise -> State e (ret d (WithSubCont Hole e')) es q
```

PushSubCont

$$\begin{array}{lcll}
\langle pushSubCont\ e\ e', D, E, q\rangle & \rightarrow & \langle e, D[pushSubCont\ \square\ e'], E, q\rangle & \text{e non-value} \\
\langle pushSubContE'\ e, D, E, q\rangle & \rightarrow & \langle e, \square, E' {+\!+} (D : E), q\rangle &
\end{array}$$

Reducing `PushSubCont e e'` ensures that the first argument is a sequence. Then it pushes the current continuation, followed by this sequence, onto the stack. The second argument is promoted to be the dominant term. This has the effect of evaluating the dominant term and return the result to the sequence.

```
1    eval (State (PushSubCont e e') d es q) =
2      case e of
3        Val (Seq es') -> State e' Hole (es'++(d:es)) q
4        otherwise -> State e (ret d (PushSubCont Hole e')) es q
```

Substitution

The reduction of `Sub e y x` uses `sub` to recursively substitute the third argument with the second in the first.

```
1    eval (State (Sub e y x) d es q) =
2      State (sub e y x) d es q
```

NewPrompt

$$\langle newPrompt, D, E, q\rangle \quad \rightarrow \quad \langle q, D, E, q+1\rangle$$

Evaluating `NewPrompt` places the value of the current prompt as the dominant term and increments the global prompt counter:

```
1    eval (State NewPrompt d es (Prompt p)) =
2      State (Val (Prompt p)) d es (Prompt $ p+1)
```

Bibliography

[1] Philippe de Groote. A cps-translation of the lambda-μ-calculus. In *Trees in Algebra and Programming - CAAP'94, 19th International Colloquium, Edinburgh, U.K., April 11-13, 1994, Proceedings*, pages 85–99, 1994.

[2] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730, 2007.

[3] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[4] Michel Parigot. Lambda-my-calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, pages 190–201, 1992.

[5] Steffen van Bakel. λ^{try}: exception handling with failure and recovery, 2015. Unpublished paper on formally modelling exception handling in the λ-calculus.

[6] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.