Imperial College London

Department of Computing

Exception Handling in Haskell

by
William S. Fisher

supervised by
Steffen van Bakel

Submitted in partial fulfillment of the requirements for the MSc
degree in Computing Science of Imperial College London

September 2016

Acknowledgements

Thanks me

Contents

CHAPTER 1

Introduction

Hel

CHAPTER 2

Background

This chapter explores what *formal systems* are and what they are useful for. It looks at a number of related formal systems and their relation to computation. It outlines context ontop of which the rest of this project is built.

Formal Systems

λ-Calculus

Definition 0.1 (Grammar for untyped λ-calculus)
λ-variables are denoted by $x, y, \ldots$

$$M, N \quad ::= \quad x \mid \lambda x.M \mid M\ N$$

Definition 0.2 (Reduction rules for λ-calculus)

$$\begin{array}{rcl} x & \to & x \\ \lambda x.M & \to & \lambda x.M \\ (\lambda x.M)N & \to & M[N/x] \end{array}$$

Logic, Types, and their Computation Interpretation

Continuations.

Delimited-Continuations.

Haskell

Monads.

$\lambda\mu$-Calculus

Definition 0.3 (Grammar for $\lambda\mu$-calculus)
λ-variables are denoted by $x, y, \ldots$ and μ-variables are denoted by $\alpha, \beta, \ldots$

$$M, N \quad ::= \quad x \mid \lambda x.M \mid M\ N \mid \mu\alpha.[\beta]M$$

Definition 0.4 (Reduction rules for $\lambda\mu$-calculus)

$$
\begin{array}{rcl}
x & \to & x \\
\lambda x.M & \to & \lambda x.M \\
\mu\alpha.[\beta]M & \to & \mu\alpha.[\beta]M \\
(\lambda x.M)N & \to & M[N/x] \\
(\mu\alpha.[\beta]M)N & \to & (\mu\alpha.[\beta]M[[\gamma]M'N/[\alpha]M'])
\end{array}
$$

The terse reduction rule at the end simple states that the application of a $\lambda\mu$-abstraction $\mu\alpha.M$ to a term N applies all the sub-terms of M labelled $[\alpha]$ to N and relabels them with a fresh μ variable.

λ^{try}-Calculus

Delimited-Continuation Calculus

Simon Peyton-Jones *et al.* extended the λ-calculus with additional operators in order create a framework for implementing delimited continuations [**?**]. This calculus will be referred to as the delimited-continuation calculus or DCC. Many calculi have been devised with control mechanisms. Like the $\lambda\mu$-calculus, these control mechanisms are all specific instances of delimited and undelimited continuations. DCC provides a set of operations that are capable of expressing many of these other common control mechanisms.

The grammar of DCC is an extension of the standard λ-calculus:

Definition 0.5 (Grammar for DCC)

$$
\begin{array}{llcl}
\text{(Variables)} & x, y, \ldots & & \\
\text{(Expressions)} & e & ::= & x \mid \lambda x.e \mid e\ e' \\
 & & \mid & \mathit{newPrompt} \mid \mathit{pushPrompt}\ e\ e \\
 & & \mid & \mathit{withSubCont}\ e\ e \mid \mathit{pushSubCont}\ e\ e
\end{array}
$$

The operational semantics can be understood through an abstract machine that transforms tuple of the form $\langle e,\ D,\ E\,q\rangle$:

The additional terms behave as follows:

- *newPrompt* returns a new and distinct prompt.
- *pushPrompt*'s first argument is a prompt which is pushed onto the continuation stack before evaluating its second argument.
- *withSubCont* captures the subcontinuation from the most recent occurrence of the first argument (a prompt) on the excution stack to the current point of execution. Aborts this continuation and applies the second argument (a λ-abstraction) to the captured continuation.
- *pushSubCont* pushes the current continuation and then its first argument (a subcontinuation) onto the continuation stack before evaluating its second argument.

Definition 0.6 (Operational semantics for DCC)

$$
\begin{array}{lcll}
\langle e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[\Box\ e'], E, q\rangle & \text{e non-value}\\
\langle v\ e, D, E, q\rangle & \Rightarrow & \langle e, D[v\ \Box], E, q\rangle & \text{e non-value}\\
\langle pushPrompt\ e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[pushPrompt\ \Box\ e'], E, q\rangle & \text{e non-value}\\
\langle withSubCont\ e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[withSubCont\ \Box\ e'], E, q\rangle & \text{e non-value}\\
\langle withSubCont\ p\ e, D, E, q\rangle & \Rightarrow & \langle e, D[withSubCont\ p\ \Box], E, q\rangle & \text{e non-value}\\
\langle pushSubCont\ e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[pushSubCont\ \Box\ e'], E, q\rangle & \text{e non-value}\\
 & & & \\
\langle (\lambda x.e)\ v, D, E, q\rangle & \Rightarrow & \langle e[v/x], D, E, q\rangle & \\
\langle newPrompt, D, E, q\rangle & \Rightarrow & \langle q, D, E, q+1\rangle & \\
\langle pushPrompt\ p\ e, D, E, q\rangle & \Rightarrow & \langle e, \Box, p : D : E, q\rangle & \\
\langle withSubCont\ p\ v, D, E, q\rangle & \Rightarrow & \langle v(D : E\overset{p}{\uparrow}, \Box, E\overset{p}{\downarrow}, q\rangle & \\
\langle pushSubContE'\ e, D, E, q\rangle & \Rightarrow & \langle e, \Box, E' + +(D : E), q\rangle & \\
 & & & \\
\langle v, D, E, q\rangle & \Rightarrow & \langle D[v], \Box, E, q\rangle & \\
\langle v, \Box, p : E, q\rangle & \Rightarrow & \langle v, \Box, E, q\rangle & \\
\langle v, \Box, D : E, q\rangle & \Rightarrow & \langle v, D, E, q\rangle &
\end{array}
$$

CHAPTER 3

DCC Interpreter

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

Interpreter

Although Peyton-Jones *et al.* implement a language-level module for DCC, we are interested in the intermediate term transformations. Examining transformation steps in full allows us to derive proofs of soundness and completeness for the translations from the λ and $\lambda\mu$ calculi into DCC. For this reason, the interpreter was implemented as a term-rewriting program.

Whereas the original grammar for the DCC abstract machine presents sequences as values, the original exposition leaves the semantics for transforming sequences into useable expressions implicit. These semantics are unpacked in the implementation details. To capture the correct behaviour in this interpreter, we must formalize these semantics as a syntax-transformation. Sequences are therefore presented as expressions with the following explicit reduction rule:

Definition 0.7 (Semantics of a sequence of continuations)
Let D_i denote some term with a hole and $D_i[v]$ denote the term D_i with the hole filled by v:

$$\langle (D_1 : D_2 : \cdots : D_n), D', E, q\rangle \quad \Rightarrow \quad \langle \lambda x.D_n[D_{n-1}[\ \ldots D_1[x] \ldots]], D', E, q\rangle$$

A sequence of contexts evaluates to an abstraction that, when applied to a value v, returns v to the first context which returns its value to the second context and so on through the whole sequence.

Implementation

Data structures. There are two data types for representing DCC terms, `Value` and `Expr`:

```
data Value = Var Char
  | Abs Char Expr
  | Prompt Int

data Expr = Val Value
  | App Expr Expr
  | Hole
  | PushPrompt Expr Expr
```

```
  | PushSubCont Expr Expr
  | WithSubCont Expr Expr
  | NewPrompt
  | Seq [Expr]
  | Sub Expr Expr Char
```

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 $\langle e,\ D,\ E,\ q \rangle$:

```
data State = State Expr Expr [Expr] Value
```

Utility Functions. Some utility functions are defined to help readability. See Figure 3 for implementations:

- `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.
- `contextToAbs :: Expr -> Expr` takes a term with a hole and returns an abstraction that fills the hole with an expression when applied to it.
- `seqToAbs :: [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. It then turns this large context into an abstraction using `contextToAbs`.
- `promptMatch :: Int -> Expr -> Bool` returns true if the second argument is a Prompt and has the same value as the first argument
- `splitBefore :: [Expr] -> Int -> [Expr]`
- `splitAfter :: [Expr] -> Int -> [Expr]`
- `sub :: [Expr] -> Int -> [Expr]`

Reduction Rules. The heavy lifting is done by the function `eval :: State -> State`. `eval` is defined inductively on the structure of the current expression. Each case of `eval` corresponds directly to at least one of the reduction rules of the DCC operational semantics. The full source can be found in the appendix:

The first case deals with applications of the form `e e'`. 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 a redex 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 first. In effect, an application first ensures the left-hand term has been evaluated fully before evaluating the right-hand term.

```
eval (State (App e e') d es q) = case e of
  Val v -> case e' of
    Val _ -> case v of (Abs x e) -> State (Sub e e' x) d es q
    otherwise -> State e' (ret d (App e Hole)) es q
  otherwise -> State e (ret d (App Hole e')) es q
```

```
contextToAbs e = (Val (Abs fresh body))
  where fresh = 'x'  -- TODO: generate truly fresh var
        body = ret e (Val (Var fresh))

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

seqToAbs es = contextToAbs $ foldr ret Hole $ reverse es

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 i p = case p of
  (Val (Prompt p')) -> i == p'
  otherwise -> False

splitBefore p es = takeWhile (not . promptMatch p) es

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

Figure 1: Utility functions for DCC interpreter

This implements the following three reduction rules:

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

The following reduction rules for pushPrompt are implemented to ensure the first expression has been evaluated to a prompt:

$$\begin{array}{lcl} \langle pushPrompt\ e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[pushPrompt\ \Box\ e'], E, q\rangle \\ \langle pushPrompt\ p\ e, D, E, q\rangle & \Rightarrow & \langle e, \Box, p : D : E, q\rangle \end{array}$$

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

The reduction rules for `WithSubCont` 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.

$$\begin{array}{lcl}
\langle withSubCont\ e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[withSubCont\ \square\ e'], E, q\rangle \\
\langle withSubCont\ p\ e, D, E, q\rangle & \Rightarrow & \langle e, D[withSubCont\ p\ \square], E, q\rangle \\
\langle withSubCont\ p\ v, D, E, q\rangle & \Rightarrow & \langle v(D : E\overset{p}{\uparrow}, \square, E\overset{p}{\downarrow}, q\rangle
\end{array}$$

```
eval (State (WithSubCont e e') d es q) = case e of
  Val v -> case e' of
    Val _ -> case v of (Prompt p) -> State (App e' (Seq (d:beforeP)))
                                                Hole afterP q
                                         where beforeP = splitBefore p es
                                               afterP = splitAfter p es
    otherwise -> State e' (ret d (WithSubCont e Hole)) es q
  otherwise -> State e (ret d (WithSubCont Hole e')) es q
```

Reducing `PushSubCont` ensures that the first argument is a sequence, pushes the current continuation onto the stack, and then pushes the abstraction that represents the sequence onto the stack. The abstraction is first applied to a Hole. This is a hack to reverse the conversion of context-sequences into abstractions. This is necessary because context-sequences need to be abstractions when being applied but need to be sequences when being composed with other sequences of contexts.

```
eval (State (PushSubCont e e') d es q) = case e of
  Val v -> State e' Hole ([App (Val v) Hole]++(d:es)) q
  otherwise -> State e (ret d (PushSubCont Hole e')) es q
```

The reduction of `Sub` states is defined inductively on the structure of the first argument of dominant term. The base case replaces matching variables with the second term. The other cases ensure that substitution is propogated to the subterms.

```
eval (State (Sub e y x) d es q) = State e' d es q
  where e' = case e of
          Val (Var m) -> if m == x then y else (Val (Var m))
          Val (Abs h m) -> Val (Abs h (sub m y x))
          App m n -> App (sub m y x) (sub n y x)
          Val (Prompt p) -> Val (Prompt p)
          NewPrompt -> NewPrompt
          PushPrompt e1 e2 -> PushPrompt (sub e1 y x) (sub e2 y x)
          WithSubCont e1 e2 -> WithSubCont (sub e1 y x) (sub e2 y x)
          PushSubCont e1 e2 -> PushSubCont (sub e1 y x) (sub e2 y x)
```

Evaluating a `Seq` transforms the sequence into an abstraction using `seqToAbs`. This corresponds to the reduction rule we introduced in Figure 3:

```
eval (State (Seq s) d es q) = State (seqToAbs s) d es q
```

Evaluated a value returns the value to the current continuation if there is one or pulls a continuation off the stack if there is not. If the stack is empty, nothing happens.

```
eval (State (Val v) d es q) = case d of
  Hole -> case es of
    (e:es') -> case e of
      (Val (Prompt p)) -> State (Val v) Hole es' q
      otherwise -> State (Val v) e es' q
    otherwise -> State (Val v) d es q
  otherwise -> State (ret d (Val v)) Hole es q
```

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

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

CHAPTER 4

Translations

λ^{try}-to-$\lambda\mu$

$\lambda\mu$-to-DCC

λ^{try}-to-DCC

CHAPTER 5

Conclusion

Evaluation

Conclusion

Future Work