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

Abstract

Implementing exception handling in Haskell. Unlike other libraries, use named exception handlers. Use the λ^{try}-calculus to formalize and explore a series of translations between multiple calculi to arrive at a translation into Haskell. Explore properties of this translation including soundness and completeness. Publish useable Haskell library.

Acknowledgements

Thanks me

Contents

1 | Introduction

Explanation of problem space: need and motivation demonstrated with examples.

Solution

Contribution

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 the context ontop of which the rest of this project is built.

Formal Systems

Formal systems are a set of rules for writing and manipulating formulae. Formulae are constructed from a set of characters called the *alphabet* by following a some formula-construction rules called the *grammar*. The only formulae considered *well-formed* in a system are those constructed according to the grammar of a system. Formal systems are used to model domains of knowledge to help better and more formally understand those domains.

Syntax and Grammars

The grammar of a formal system describes the system's syntax. Grammars are rules for constructing formulae that are well-formed. Formulae produced according to the grammar of a system are well-formed according to the syntax of that system.

We defined grammars using Backus-Naur Form or BNF:

$$M ::= t \mid f$$

This grammar describes that syntactically-valid constructs are either the letter t or the letter f. Grammars can be recursive which allows much more expressive construction rules:

This grammar describes formulae containing the any number of occurrences of the letters t or f separated by either an a or an o.

t	by (1)
$t\ o\ f$	by (2 & 4)
$t\ o\ f\ a\ t$	by (1 & 3)

$$
\begin{array}{rcll}
M, N & ::= & t & (1) \\
& | & f & (2) \\
& | & M\ a\ N & (3) \\
& | & M\ o\ N & (4)
\end{array}
$$

Figure 2.1: Grammar for producing the letters t or f connected by the letters a or o

Derivation Rules

Whereas a grammar describes the rules for producing well-formed formulae, the derivation rules describe rules for transforming formulae of a particular form into a new formula. Using the grammar from Figure 2, we add derivation rules:

$$
\begin{array}{rcl}
t\ a\ M & \rightarrow & M \\
M\ a\ t & \rightarrow & M \\
f\ a\ f & \rightarrow & f
\end{array}
$$

$$
\begin{array}{rcl}
M\ o\ t & \rightarrow & t \\
t\ o\ M & \rightarrow & t \\
f\ o\ f & \rightarrow & f
\end{array}
$$

These rules describe that if a formula matches the pattern on the left-hand side, where M represents a well-formed formula, it can be replaced by the formula on the right-hand side.

Domain Modelling

The syntax and derivation rules of a formal system are defined to model some domain. This isomorphism between the domain and the formal system means we can attempt to discover truths about the domain through studying the formal system.

For example, take the tf-system described above. Without understanding the domain, we are able to manipulate formulae of the system to create new formulae. The tf-system is isomorphic to Boolean algebra:

tf-system	Boolean algebra
t	1
t	0
a	$\wedge$
o	$\vee$

Using formal systems allows us to understand the domains they model from different perspectives and thereby learn novel truths about them.

λ-Calculus

In response to Hilbert's *Entscheidungsproblem*, Alonzo Church defined the λ-calculus. It is a formal system capable of expressing the set of effectively-computible algorithms. Ontop of this, he built his proof that not all algorithms are decidable. Shortly after, Godel and Turing created their own models of effective computibility. [1] These models were later proved to all be equivalent.

Syntax

λ-variables are represented by $x, y, z, \&c$. Variables denote an arbitrary value: they do not describe what the value is but that any two occurrences of the same variable represent the same value. The grammar for constructing well-formed λ-terms is:

Definition 2.0.1 (Grammar for untyped λ-calculus)

λ-variables are denoted by $x, y, \ldots$

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

λ-abstractions are represented by $\lambda x.M$ where x is a parameter and M is the body of the abstraction. The same idea is expressed by more conventional notation as a mathematical function $f(x) = M$. The λ annotates the beginning of an abstraction and the . separates the parameter from the body of the abstraction. This grammar is recursive meaning the body of an abstraction is just another term constructed according to the grammar. Some examples of abstractions are:

$\lambda x.x$
$\lambda x.xy$
$\lambda x.(\lambda y.xy)$

Figure 2.2: Examples of valid λ-abstractions

Applications are represented by any two terms, constructed according to the grammar, placed alongside one another. Application gives highest precedence to the left-most terms. Bracketing can be introduced to enforce alternative application order for example xyz is implicitly read as $(xy)z$ but an be written as $x(yz)$ to describe that the application of yz should come first. Examples of applications are:

[1] General recursive functions and Turing machines, respectively

xy
xyz
$x(yz)$
$(\lambda x.x)y$

Figure 2.3: Examples of valid applications

Reduction Rules

First we will introduce the substitution notation $M[N/x]$. This denotes the term M with all occurrences of x replaced by N. The substitution notation is defined inductively as:

Definition 2.0.2 (Substitution notation for λ-terms)

$$\begin{array}{rcll} x[y/x] & \rightarrow & y & \\ z[y/x] & \rightarrow & z & (z \neq x) \\ (\lambda z.M)[y/x] & \rightarrow & \lambda z.(M[y/x]) & \\ (MN)[y/x] & \rightarrow & M[y/x]N[y/x] & \end{array}$$

β-reduction

The main derivation rule of the λ-calculus is β-reduction. If term M β-reduces to term N, we write $M \rightarrow_\beta N$ although the β subscript can be omitted if it is clear from context. β-reduction is defined for the application of two terms:

Definition 2.0.3 (β-reduction for λ-calculus)

$$(\lambda x.M)N \quad \rightarrow_\beta \quad M[N/x]$$

λ-variables and λ-abstactions are *values*: they do not reduce to other terms. If a formula is a value, the reduction terminates on that value. Only applications reduce to other terms. This means that an application is a reducible expression or a *redex*. Reducing a redex models the computing of a function.

α-reduction

The λ-calculus defines a reduction rule for renaming variables. Variable names are arbitrary and chosen just to denote identity: all occurrences of x are the same. This can become a problem in the following case:

$$(\lambda y.\lambda x.xy)(\lambda x.x)$$

After the application is reduced, we have the term:

$$\lambda x.x(\lambda x.x)$$

In this case, it is ambiguous which λ-abstraction the right-most x is bound by. When this term is applied to another, will the substitution occur to all occurrences of x? From the initial term, it is clear that this would be incorrect. The λ-calculus introduces α-reduction to solve this:

Definition 2.0.4 (α-REDUCTION FOR λ-CALCULUS)

$$\lambda x.M \to_\alpha \lambda y.M[y/x] \quad (y \notin fv(M))$$

This means we can rename the lead variable of an abstraction M on the conditions that:

1. All variables bound by that abstraction are renamed the same
2. The variable it is changed to is not currently in use in M

Logic and Types

Implicative Intuitionistic Logic

A formal system based on Gerhard Gentzen's natural deduction. Restricted only to $\to \mathcal{I}$ and $\to \mathcal{E}$. Following Brouwer, also leaves out law of excluded middle.

$$(\to \mathcal{I}) \quad \frac{A \to B \quad A}{B} \qquad (\to \mathcal{E}) \quad \frac{[A] \vdash B}{A \to B}$$

Type Assignment

Type assignment introduces additional grammar and restrictions on the reduction rules of a system. These extensions prevent logically inconsistent terms from being constructed. A type assignment has the form:

$$M : \alpha$$

which states that term M has the type α. Like variables, type variables are abstract: they do not describe anything more about a type than its identity. That is to say $x : A$ and $y : A$ have the same type but we cannot say any more about what that type is.

A type is either some uppercase Latin letter or it is two valid types connected by a $\rightarrow$. This is described by the following BNF grammar:

$$A, B ::= \varphi \mid A \rightarrow B$$

Typing rules have their own form that resembles Gentzen's sequent calculus:

$$\Gamma \vdash T$$

This describes that Γ is a set of typed λ-terms and T is a typed λ-term that is derivable from Γ.

Curry-Howard Isomophism

Curry-Howard isomorphism states that their is an isomorphism between the typed of a term and a logical proposition. The term itself is the proof of the proposition.

Haskell

Data Types

New data types can be introduced into Haskell in 3 distinct ways. First, using the `data` keyword:

```
data Animal a = Dog a
  | Cat
```

The `data` keyword begins the definition of a new data type. The word immediately following determines the type constructor for the new type. Following this is a type parameter for the type constructor. There can be any number of type parameters, including zero. The right-hand side of the `=` introduces a `|`-separated list of data constructors.

```
> let hector = Cat
> :t hector
hector :: Animal a

> let topaz = Dog "foo"
> :t topaz
topaz :: Animal String
```

The type parameter is constrained by the type of the value the data constructor was initialized with. In the example above, calling the `Dog` data

constructor with a string makes the type `Animal String` rather than the more general `Animal a`.

The second method for introducing new data types is the `newtype` keyword. The key difference between `data` and `newtype` is that `newtype` can only have one data constructor. Informally, this implies a kind of isomorphism:

```
newtype Foo a = Foo (a -> Integer)
```

The type constructor can take type parameters which will be constrained by the inhabitants of the data constructor. This data type expresses an isomorphism between `Foo a` and functions from `a` to `Integer`s.

Finally, we can introduce type aliases using the `type` keyword:

```
type Name = String
```

Again, we introduce a type constructor `Name` but this time we name another type, in this case `String`, as its inhabitant. This means that the type `Name` is a type alias for `String` and will share the same data constructors.

Type Level/Value Level

Haskell distinguishes between terms on the type level and terms on the value level. Type level terms are descriptions of the types of a value. They provide restrictions on the construction of invalid terms. For instance if we have a function of type `String -> Integer`, we cannot apply it to a term of type `Boolean`. The type-checker will throw an error before any value-level computation is initiated.

The value level is the level on which data is constructed and manipulated. The operation `1+1` occurs on the value level. The value level is where computation takes place and the type level is where static analysis of the program type takes place.

Type Classes

Haskell adds type classes to the type level. Types can have instances of type classes. The most similar concept from Object-Oriented programming is *interfaces*.

```
class Addable a where
  (add) :: a -> a -> a
```

Type classes are introduced using the `class` keyword. Beneath that are the function names and corresponding type-signatures of the functions that an instance of a class must implement.

For example, we can create instances of the `Addable` class:

```
data Number = One | Two | ThreeOrMore

instance Addable Number where
  add One One = Two
  add One Two = ThreeOrMore
  add Two One = ThreeOrMore
```

Continuations

Compound terms can be decomposed into two seperate parts: a dominant term and a context. The dominant term is the term currently being evaluated. The context is a term with a hole that will be filled with the value the dominant term reduces to.

Assume that $M \rightarrow_\beta M'$

(Compound term)	MN	
(Decompose)	M	$\Box N$
(Beta-reduce dominant term)	M'	$\Box N$
(Refill hole of context)	$M'N$	

Figure 2.4: Decomposing a term into a dominant term and a context

When M has β-reduced to a value – M' – then the hole of the context $\Box N$ is filled to form $M'N$. What the dominant term and context are for a given term depends on the reduction rules and strategy. The context is what remains to be reduced at given moment of reduction. Thus a context is also called a *continuation*.

Undelimited Continuations

For more complex terms, the waiting context will grow as the dominant term gets further decomposed:

$(MM')M''$	
(MM')	$\Box M''$
M	$(\Box M')M''$

Figure 2.5: Decomposing a term into multiple contexts

By amalgamating continuatinos into one big continuation we only have two components at point during the reduction: the current dominant term and the *current continuation*.

Assume we have some reduction rules defined for manipulating continuations. This model keeps continuations grouped together which means these hypothetical reduction rules could manipulate only this entire remaining continuation. For this reason, continuations that can only be manipulated in their entirety are **undelimited continuations**.

Delimited-Continuations

Instead, if we maintain a stack of continuations when decomposing complex terms, we can keep continuations separated:

$$\begin{array}{lll} (MM')M'' & & \\ (MM') & \Box M'' & \\ M & \Box M' & \Box M'' \end{array}$$

Figure 2.6: Decomposing a term into multiple contexts

Here, when a dominant term has been reduced, the reduct is returned to its corresponding continuation. This newly joined term then becomes the dominant redex. After this new dominant term has been reduced, it will be returned to the next waiting continuation, and so on. Throughout this process, we maintain each continuation separately.

Assume again that we have reduction rules defined for manipulating continuations. By keeping continuations separate, this model would allow use to use parts of the stack selectively for instance placing delimiters between portions of interest. The increased granularity of control means we can manipulate not just the entire remaining continuation but sections of it. Thus, continuations of this kind are called **delimited continuations**.

Continuation-Passing Style

By rewriting λ-terms, the continuations can be made explicit. All terms must be turned into λ-abstractions of some variable k where k is the continuation of a term. k is then called on the result of the term, triggering the continuation to take control. This style of writing λ-terms is called continuation-passing style or CPS.

Definition 2.0.5 (Translation of standard λ-terms into CPS)

$$\begin{array}{lcl} \underline{x} & = & \lambda k.kx \\ \underline{\lambda x.M} & = & \lambda k.k(\lambda x.\underline{M}) \\ \underline{MN} & = & \lambda k.M(\lambda m.m\underline{N}(\lambda n.mnk) \end{array}$$

The term that a CPS program terminates on will be of the form $\lambda k.kM$. In order to extract the value, a *final continuation* must be provided. Depending on the context, this could be an identity function $\lambda x.x$ or a display operation $\lambda x.\text{display } x$ to display the results of the program.

Figure 2.7: Extracting the final value from a terminated CPS program

$$\begin{array}{ll} & (\lambda k.kM)(\lambda x.x) \\ \rightarrow & (\lambda x.x)M \\ \rightarrow & M \end{array}$$

The translation of standard λ-terms into CPS similarly transforms the *types* of λ-terms. For example, a term $x : A$ becomes $\lambda k.kx : (A \rightarrow B) \rightarrow B$. This type represents a delayed computation: a computation that is waiting for a function to continue execution with. In order to resume the computation, the term must be applied to a continuation.

As an example, take the term M where

$$M = \lambda k.kx$$

To access the value x contained in M, we have to apply M to a continuation function $\lambda m.N$:

$$\begin{array}{l} (\lambda k.kx)(\lambda m.N) \\ (\lambda m.N)x \\ N[x/m] \end{array}$$

Within the body of N, m is bound to the value contained by M. So we can think about M as a suspended computation that, when applied to a continuation, applies the continuation to x. Looking at the type $(A \rightarrow B) \rightarrow B$ again, it is clear that A is the type of the term passed to the continuation of a CPS-term:

$$\begin{array}{l} \lambda k.kx : (A \rightarrow B) \rightarrow B \\ k : (A \rightarrow B) \\ kx : B \\ x : A \end{array}$$

Monads

If we have two suspended computations M and M' and we want to run M and then M', we have to apply M to a continuation to access its value and then do the same to M':

$$M(\lambda m.M'(\lambda m'.N))$$

This is a common operation so we define a utility operator `>>=` that binds the first suspended computation to a continuation which returns another suspended computation[2]:

$$\texttt{>>=} : ((A \to B) \to B) \to (A \to ((B \to C) \to C)) \to ((B \to C) \to C)$$

The type $(A \to B) \to B$ that represents a suspended computation returning a value of type A to its continuation we will call an A-computation or $Comp\ A$. We can rewrite the type signature of `>>=`:

$$\texttt{>>=} : Comp\ A \to (A \to Comp\ B) \to Comp\ B$$

We define another operator, *return*, that takes a value and returns a suspended computation that returns that value:

$$return : A \to Comp\ A$$

The type constructor $Comp\ A$, together with the two utility functions `>>=` and *return*, make up Haskell's Monad type class:

```
class Monad M where
  (>>=) :: M a -> (a -> M b) -> M b
  return :: a -> M a
```

The Monad type class generalizes CPS terms: they represent suspended computations that can be composed using `>>=`. Just like CPS terms, a Monad type `M a` tells us that we have a term that will pass values of type `a` to the continuation it is bound to using `>>=`.

$\lambda\mu$-Calculus

Syntax

Definition 2.0.6 (Grammar for $\lambda\mu$-calculus)

λ-variables are denoted by $x, y, \ldots$ and μ-variables are denoted by $\alpha, \beta, \ldots$

$$\begin{array}{lrcl} \text{(Unnamed term)} & M, N & ::= & x \mid \lambda x.M \mid M\ N \mid \mu\alpha.C \\ \text{(Named term)} & C & ::= & [\alpha]M \end{array}$$

Just as λ introduces a new λ-abstraction, $\lambda\mu$ introduces a new $\lambda\mu$-abstraction. The body of a $\lambda\mu$-abstraction must be a named term. A named term consists of a name of the form $[\alpha]$ followed by an unnamed term.

[2] `(>>=)` is pronounced 'bind'.

Reduction Rules

Definition 2.0.7 (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.

Computational Significance

The additional $\lambda\mu$ reduction rules model context manipulation. $\lambda\mu$-variables map to contexts. When an unnamed term is labelled with a $\lambda\mu$-variable, it is evaluated in that context. For instance the named term $['a]M$ has the effect of evaluating M in the context pointed to by α.

To make this more concrete, consider the compound term $(\mu\alpha.[\beta]M)\ N$. First we decompose the term into a dominant term $(\mu\alpha.[\beta]M)$ and a context $\Box N$. Informally, we can imagine that the $\lambda\mu$-variable α now maps to this context $\{\alpha \Rightarrow \Box N\}$:

Example 2.0.8

Dominant	**Context**
$(\mu\alpha.[\beta]M)N$	
$\mu\alpha.[\beta]M$	$\Box N$

All subterms of M labelled α will now be evaluated in the context $\Box N$ and the context will destroyed. For example, let us replace M with $\mu \circ .[\alpha](\lambda s.fs)$:

Example 2.0.9

Dominant	**Context**	
$\mu\alpha.[\beta]\mu \circ .[\alpha](\lambda s.fs)$	$\Box N$	
$\mu\alpha.[\alpha](\lambda s.fs)$	$\Box N$	
$\mu\gamma.[\gamma](\lambda s.fs)N$		(γ fresh)

After applying the term $[\alpha](\lambda s.fs)$ to N, the context $\Box N$ is consumed and every occurrence of α is replaced with a fresh variable – in this case a γ – to clarify that the new $\lambda\mu$-abstraction points to a new context. This means that $\lambda\mu$-abstractions will pass all of the applicative contexts to the named subterms:

Example 2.0.10

Dominant	**Context**	
$(\mu\alpha.[\alpha](\lambda s.\lambda t.st))MN$		
$(\mu\alpha.[\alpha](\lambda s.\lambda t.st))M$	$\Box N$	
$\mu\alpha.[\alpha](\lambda s.\lambda t.st)$	$\Box M : \Box N$	
$\mu\gamma.[\gamma](\lambda s.\lambda t.st)M$	$\Box N$	(γ fresh)
$\mu\delta.[\delta](\lambda s.\lambda t.st)MN$	$\Box N$	(δ fresh)

Isomorphism & Computational Interpretation

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:

Syntax

Definition 2.0.11 (Grammar for DCC)

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

Reduction Rules

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

Significance

The additional terms behave as follows:

- *newPrompt* returns a new and distinct prompt.

Definition 2.0.12 (Operational semantics for DCC)

$$
\begin{array}{lcll}
\langle e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[\Box\ e'], E, q\rangle & \text{e non-value} \\
\langle v\ e, D, E, q\rangle & \Rightarrow & \langle e, D[v\ \Box], E, q\rangle & \text{e non-value} \\
\langle \mathit{pushPrompt}\ e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[\mathit{pushPrompt}\ \Box\ e'], E, q\rangle & \text{e non-value} \\
\langle \mathit{withSubCont}\ e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[\mathit{withSubCont}\ \Box\ e'], E, q\rangle & \text{e non-value} \\
\langle \mathit{withSubCont}\ p\ e, D, E, q\rangle & \Rightarrow & \langle e, D[\mathit{withSubCont}\ p\ \Box], E, q\rangle & \text{e non-value} \\
\langle \mathit{pushSubCont}\ e\ e', D, E, q\rangle & \Rightarrow & \langle e, D[\mathit{pushSubCont}\ \Box\ e'], E, q\rangle & \text{e non-value}
\end{array}
$$

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

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

- *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.

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 3.0.1 (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 m) -> State (Sub m 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
```

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
```

```
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 3.1: Utility functions for DCC interpreter

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)
```

4 | Translations

$\lambda\mu$-to-DCC

The translation of a full program M in the $\lambda\mu$-calculus to DCC is defined as $[\![M]\!]_p$ where the subscript p denotes the translation is initialized with prompt p. Formally, this means that $[\![M]\!]_p \triangleq (\lambda p.\text{PP}\ p\ [\![M]\!])\text{NP}$

Definition 4.0.1 (Initialization of a translation of M into DCC)

$$[\![M]\!]_p \triangleq (\lambda p.\text{PP}\ p\ [\![M]\!])\text{NP}$$

Definition 4.0.2 (Interpretation of $\lambda\mu$ into DCC)

$$\begin{array}{lcl} [\![x]\!] & \triangleq & x \\ [\![\lambda x.M]\!] & \triangleq & \lambda x.[\![M]\!] \\ [\![MN]\!] & \triangleq & [\![M]\!][\![N]\!] \\ [\![\mu\alpha.M]\!] & \triangleq & \text{WSC}\ p\ \lambda\alpha.\text{PP}\ p\ [\![M]\!] \\ [\![[\beta]M]\!] & \triangleq & \text{PSC}\ \beta\ [\![M]\!] \end{array}$$

Theorem 4.0.3 (Soundness of $[\![\bullet]\!]$) *If $M \to_\mu N$ then $[\![M]\!] \to_{DCC} [\![N]\!]$*

Proof. By induction on the definition of $\to_\mu$

$$\begin{array}{rl} & (\lambda x.M)N \to M[N/x]: \\ & \underline{(\lambda x.M)N} \\ \triangleq & \underline{(\lambda x.M)\underline{N}} \\ \triangleq & (\lambda x.\underline{M})\underline{N} \\ \to_\beta & \underline{M}[\underline{N}/x] \end{array}$$

$(\mu\alpha.[\beta]M)N \to \mu\alpha.([\beta]M)[[\alpha]M'N/[\alpha]M'] :$

$$
\begin{array}{rll}
 & \underline{(\mu\alpha.[\beta]M)N} & \\
\triangleq & \underline{(\mu\alpha.[\beta M)\underline{N}} & \\
\triangleq & \underline{(\textsc{wsc}\ p\ \lambda\alpha.\textsc{pp}\ p\ (\textsc{psc}\ \beta\ \underline{M}))\underline{N}} & p : [] \\
\to_{DCC} & \textsc{wsc}\ p\ \lambda\alpha.\textsc{pp}\ p\ (\textsc{psc}\ \beta\ \underline{M}) & \Box\underline{N} : p : [] \\
\to_{DCC} & (\lambda\alpha.\textsc{pp}\ p\ (\textsc{psc}\ \beta\ \underline{M}))(\Box\underline{N}) & [] \\
\to_{\beta} & (\textsc{pp}\ p\ (\textsc{psc}\ \beta\ \underline{M}))[\Box\underline{N}/\alpha] & [] \\
\to_{\beta} & \textsc{pp}\ p\ (\textsc{psc}\ \beta\ (\underline{M}[\Box\underline{N}/\alpha])) & [] \\
\to_{DCC} & \textsc{psc}\ \beta\ (\underline{M}[\Box\underline{N}/\alpha]) & p : [] \\
\triangleq & \underline{\mu\alpha.([\beta]M)[[\alpha]M'N/[\alpha]M']} &
\end{array}
$$

$\mu\alpha.[\alpha]M \to M :$

$$
\begin{array}{rll}
 & \underline{\mu\alpha.[\alpha]M} & \\
\triangleq & \textsc{wsc}\ p\ \lambda\alpha.\textsc{pp}\ p\ (\textsc{psc}\ \alpha\ \underline{M}) & p : [] \\
\to_{DCC} & \lambda\alpha.\textsc{pp}\ p\ (\textsc{psc}\ \alpha\ \underline{M})(\Box) & [] \\
\to_{\beta} & \textsc{pp}\ p\ (\textsc{psc}\ \alpha\ \underline{M})[\Box/\alpha] & [] \\
\to_{\beta} & \textsc{pp}\ p\ (\textsc{psc}\ \Box\ (\underline{M}[\Box/\alpha]) & [] \\
\to_{DCC} & \textsc{psc}\ \Box\ (\underline{M}[\Box/\alpha] & p : [] \\
\to_{DCC} & \underline{M}[\Box/\alpha] & p : [] \\
\triangleq & \underline{M} &
\end{array}
$$

$\mu\alpha.[\beta]\mu\gamma.[\delta]M \to \mu\alpha.[\delta](M[\beta/\gamma]) :$

$$
\begin{array}{rll}
\triangleq & \textsc{wsc}\ p\ \lambda\alpha.\textsc{pp}\ p\ \underline{[\beta]\mu\gamma.[\delta]M} & p : [] \\
\to_{DCC} & \lambda\alpha.\textsc{pp}\ p\ \underline{[\beta]\mu\gamma.[\delta]M}(\Box) & [] \\
\to_{\beta} & (\textsc{pp}\ p\ \underline{[\beta]\mu\gamma.[\delta]M})[\Box/\alpha] & [] \\
\to_{\beta} & \textsc{pp}\ p\ \underline{[\beta]\mu\gamma.[\delta]M}[\Box/\alpha] & [] \\
\to_{\beta} & \underline{[\beta]\mu\gamma.[\delta]M}[\Box/\alpha] & p : [] \\
\triangleq & (psc\beta\mu\gamma.[\delta]M)[\Box/\alpha] & p : [] \\
\triangleq & \underline{\mu\gamma.[\delta]M)[\Box/\alpha]} & \beta : p : [] \\
\triangleq & \underline{(\textsc{wsc}\ p\ \lambda\gamma.[\delta]M)}[\Box/\alpha] & \beta : p : [] \\
\triangleq & \textsc{wsc}\ p\ \lambda\gamma.\textsc{pp}\ p\ \underline{[\delta]M}[\Box/\alpha] & \beta : p : [] \\
\triangleq & (\lambda\gamma.\textsc{pp}\ p\ \underline{[\delta]M}[\Box/\alpha])(\beta) & [] \\
\triangleq & \underline{(\textsc{pp}\ p\ [\delta]M}[\Box/\alpha])[\beta/\gamma] & [] \\
\triangleq & \textsc{pp}\ p\ (\underline{[\delta]M}[\Box/\alpha])[\beta/\gamma] & [] \\
\triangleq & \underline{[\delta]M}[\Box/\alpha])[\beta/\gamma] & p : [] \\
\triangleq & \underline{(\textsc{psc}\ \delta\ M}[\Box/\alpha][\beta/\gamma] & p : [] \\
\triangleq & \textsc{psc}\ \delta\ (M[\Box/\alpha])[\beta/\gamma] & p : [] \\
\triangleq & \underline{\mu\alpha.[\delta](M[\beta/\gamma])} & p : []
\end{array}
$$

$$
\begin{array}{rll}
\multicolumn{3}{l}{\mu\alpha.[\beta]\mu\gamma.[\gamma]M \to \mu\alpha.[\beta](M[\beta/\gamma]) :} \\
& \underline{\mu\alpha.[\beta]\mu\gamma.[\delta]M} & \\
\triangleq & \textsc{wsc}\ p\ \lambda\alpha.\textsc{pp}\ p\ \underline{[\beta]\mu\gamma.[\delta]M} & p : [] \\
\to_{DCC} & \lambda\alpha.\textsc{pp}\ p\ \underline{[\beta]\mu\gamma.[\delta]M}(\Box) & [] \\
\to_\beta & (\textsc{pp}\ p\ \underline{[\beta]\mu\gamma.[\delta]M})[\Box/\alpha] & [] \\
\to_\beta & \textsc{pp}\ p\ \underline{[\beta]\mu\gamma.[\delta]M}[\Box/\alpha] & [] \\
\to_\beta & \underline{[\beta]\mu\gamma.[\delta]M}[\Box/\alpha] & p : [] \\
\triangleq & (psc\beta\underline{\mu\gamma.[\delta]M})[\Box/\alpha] & p : [] \\
\triangleq & \underline{\mu\gamma.[\delta]M})[\Box/\alpha] & \beta : p : [] \\
\triangleq & \underline{(\textsc{wsc}\ p\ \lambda\gamma.[\delta]M})[\Box/\alpha] & \beta : p : [] \\
\triangleq & \textsc{wsc}\ p\ \lambda\gamma.\textsc{pp}\ p\ \underline{[\delta]M}[\Box/\alpha] & \beta : p : [] \\
\triangleq & (\lambda\gamma.\textsc{pp}\ p\ \underline{[\delta]M}[\Box/\alpha])(\beta) & [] \\
\triangleq & (\textsc{pp}\ p\ \underline{[\delta]M}[\Box/\alpha])[\beta/\gamma] & [] \\
\triangleq & \textsc{pp}\ p\ (\underline{[\delta]M}[\Box/\alpha])[\beta/\gamma] & [] \\
\triangleq & \underline{[\delta]M}[\Box/\alpha])[\beta/\gamma] & p : [] \\
\triangleq & \underline{(\textsc{psc}\ \delta\ M}[\Box/\alpha][\beta/\gamma] & p : [] \\
\triangleq & \textsc{psc}\ \beta\ (M[\Box/\alpha])[\beta/\gamma] & p : [] \\
\triangleq & \underline{\mu\alpha.[\beta](M[\beta/\gamma])} & p : []
\end{array}
$$

$$
\begin{array}{rl}
\multicolumn{2}{l}{(\mu\delta.[\alpha]M)[[\alpha]M'N/[\alpha]M'] \to (\mu\delta.[\alpha](M[[\alpha]M'N/[\alpha]M'])} \\
& \underline{(\mu\delta.[\alpha]M)[[\gamma]M'N/[\alpha]M']} \\
\triangleq & \underline{(\textsc{wsc}\ p\ \lambda\delta.\textsc{pp}\ p\ (\textsc{psc}\ \alpha\ M))[\Box N/\alpha]} \\
\to_\beta & \textsc{wsc}\ p\ \lambda\delta.\textsc{pp}\ p\ (\textsc{psc}\ \Box N\ (M[\Box N/\alpha]))) \\
\triangleq & \underline{\mu\delta.[\alpha](M[[\alpha]M'N/[\alpha]M'])}
\end{array}
$$

$$
\begin{array}{rll}
\multicolumn{3}{l}{M[[\alpha]M'N/[\alpha]M] \to M \quad (\alpha \notin \mathrm{fn}(M)) :} \\
& \underline{M[[\alpha]M'N/[\alpha]M]} & \\
\triangleq & \underline{M}[\Box N/\alpha] & \\
\to_\beta & \underline{M} & (\alpha \notin \mathrm{fv}(M))
\end{array}
$$

□

λ^{try}-to-DCC

If we append the translation of λ^{try} into $\lambda\mu$ with the translating of $\lambda\mu$ to DCC, we get a translation from λ-calculus to DCC:

Definition 4.0.4 TRANSLATION OF λ^{try} INTO DCC

$$
\begin{array}{rcl}
\underline{x} & \triangleq & x \\
\underline{\lambda x.M} & \triangleq & \lambda x.\underline{M} \\
\underline{MN} & \triangleq & \underline{MN} \\
\underline{\text{throw n}(M)} & \triangleq & \text{WSC } p\ \lambda \circ .\text{PP } p\ (psc\ n\ (c\ \underline{M})) \\
\underline{\text{try } M;\ \overbrace{\text{catch n}(x) = L}^{\text{i} > 1}} & \triangleq & (\lambda c.\text{WSC } p\ \lambda n.\text{PP } p\ (\text{PSC } n\ \underline{M}))(\lambda x.\underline{L}) \\
\underline{\text{try } M;\ \overbrace{\text{catch n}_i(x) = M_i}^{\text{i} > 1};\ \text{catch m}(x) = L} & & \\
\triangleq & & \\
(\lambda c.\text{WSC } p\ \lambda m.\text{PP } p\ (\text{PSC } m\ \underline{\text{try } M;\ \overbrace{\text{catch n}_i(x) = M_i}^{\text{i} > 1}}))(\lambda x.\underline{L}) & &
\end{array}
$$

5 | Conclusion

Evaluation

Conclusion

Future Work

Bibliography

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