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.

What are exceptions? How are they typed? What have approaches been before?

van Bakel and the λ^{try}-calculus is different approach. λ^{try}already compared to the 'classical- cal...

1.1 Solution

1.2 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.

2.1 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.

2.1.1 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.

$$\begin{array}{lr}
t & \text{by (1)} \\
t\ o\ f & \text{by (2 \& 4)} \\
t\ o\ f\ a\ t & \text{by (1 \& 3)}
\end{array}$$

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

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

2.1.2 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 2.1.1, we add derivation rules:

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

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.

2.1.3 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
f	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.

2.1.4 Derivation Strategies

When applying derivation rules to compound terms, we can imagine the compound term being decomposed into simpler terms until one of the derivation rules applies. When we decompose a term, we separate it into a dominant term and a context:

$$\begin{array}{ccc} & t\ o\ f\ a\ t & \\ t\ o\ f & & \Box\ a\ t \end{array}$$

The left-hand side is the dominant term and the right-hand side is the context. The $\Box$ in the context denotes a hole that needs to be filled to create a full term.

Once we have a term that we can apply a derivation rule to, we apply the derivation rule and recombine the context with the resulting term:

$$\begin{array}{rll} & t\ o\ f & \Box\ a\ t \\ \rightarrow & t & \Box\ a\ t \\ \textit{recombine} & t\ a\ t & \end{array}$$

The term $t\ o\ f\ a\ t$ can be decomposed in two ways:

$$\begin{array}{lll} 1. & t\ o\ f & \Box\ a\ t \\ 2. & f\ a\ t & t\ o\ \Box \end{array}$$

When we have more than one way derivation rules can be applied to a term, we can use *derivation strategies* to determine which rule we apply. The derivation strategy we use decides how our compound terms are decomposed. In our tf-system, there are two obvious derivation strategies: either apply derivations starting from the left or starting from the right.

$$\begin{array}{rcl} \textbf{(left)} & t\ o\ f\ a\ t & \\ \rightarrow & t\ o\ f & \Box\ a\ t \\ \rightarrow & t & \Box\ a\ t \\ \rightarrow & t\ a\ t & \end{array}$$

$$\begin{array}{rcl} \textbf{(right)} & t\ o\ f\ a\ t & \\ \rightarrow & f\ a\ t & t\ o\ \Box \\ \rightarrow & f & t\ o\ \Box \\ \textit{recombine} & t\ o\ f & \end{array}$$

Whereas derivation rules are defined in the system, derivation strategies are methods of choosing which derivation rule to apply when given a choice.

2.2 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 effectivelycomputible 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.

2.2.1 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 2.2.1 (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:

$$\begin{array}{l} \lambda x.x \\ \lambda x.xy \\ \lambda x.(\lambda y.xy) \end{array}$$

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:

2.2.2 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

[1] General recursive functions and Turing machines, respectively

$$\begin{array}{l} xy \\ xyz \\ x(yz) \\ (\lambda x.x)y \end{array}$$

Figure 2.3: Examples of valid applications

is defined inductively as:

Definition 2.2.2 (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 2.2.3 (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 2.2.4 (α-REDUCTION FOR λ-CALCULUS)

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

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

2.2.3 Head Reduction

Head reduction is a reduction strategy for the λ-calculus. In head reduction, only the head variables are reduced. The head variable hv of a term is defined inductively:

Definition 2.2.5 HEAD VARIABLE hv FOR λ-TERMS

$$\begin{array}{rcl} hv(\lambda x.M) & = & hv(M) \\ hv(xM_1 \dots M_n) & = & x \end{array}$$

By only allowing reduction on head variables, β-reduction is restricted to the following contexts:

Definition 2.2.6 HEAD REDUCTION FOR λ-CALCULUS

$$\begin{array}{lll} M \rightarrow N & \Rightarrow & \{ \\ & & \quad \begin{array}{rcl} \lambda x.M & \rightarrow_H & \lambda x.N \\ M\overline{L} & \rightarrow_H & N\overline{L} \end{array} \\ & & \} \end{array}$$

These restricted rules define head reduction for the λ-calculus. When a term can no longer be β-reduced, it is in β-normal form. Similarly, if a term cannot be head-reduced any further, it is in βH-normal form. We can define βH-normal form on λ-terms with the following BNF grammar:

Definition 2.2.7 Grammar for λ-terms in βH-normal form.

$$\begin{array}{rcl} N & ::= & \lambda x.N \\ & | & xM_1 \ldots M_2 \end{array}$$

βH-normal form only allows the reduction of the outermost, left-most redex. We define $\to_{\beta H}^{nf}$ as the reduction of a term to βH-normal form. Similarly we use $\to_{\beta}^{nf}$ to denote the reduction to β-normal form.

Example 2.2.8 (λ-terms in β- and βH-normal forms)

$$(\lambda v.\lambda y.z(vy))(\lambda x.x) \begin{array}{ll} \to_{\beta}^{nf} & \lambda y.zy \\ \\ \to_{\beta H}^{nf} & \lambda y.z((\lambda x.x)y) \end{array}$$

2.3 Logic and Types

There are many formal systems for describing logic. These systems attempt to describe the relationship between logical statements. Like all formal systems, they allow us to derive new logical statements by following transformation rules.

2.3.1 Implicative Intuitionistic Logic

Implicative Intuitionistic Logic (IIL) is a subset of Gerhard Gentzen's system of natural deduction. It is restricted to only the $\to \mathcal{I}$ and $\to \mathcal{E}$ rules. Following Brouwer, it eschews the law of excluded middle. [2]

In natural deduction, the → symbol represents implication. To say $A \to B$ is to say that whenever A is true, we know B is true. The statement should be read "A implies B".

Logical *inference rules* describe that if some statement A is true, we can take for granted that some other statement B is true. The notation for this is:

$$\frac{A}{B}$$

Using this notation, we can now describe the inference rules for IIL:

$$(\to \mathcal{E}) \quad \frac{A \to B \qquad A}{B} \qquad\qquad (\to \mathcal{I}) \quad \frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B}$$

[2] The law of excluded middle says $\vdash P \vee \neg P$

The $\rightarrow \mathcal{E}$ rule says that given the statement $A \rightarrow B$ and the statement A, we can conclude B. For instance, consider $A =$ “It is raining” and $B =$ “It is wet outside”. The statement $A \rightarrow B$ becomes “If (it is raining) then (it is wet outside)”. jk If we know that “If (it is raining) then (it is wet outside)” and we are told “It is raining”, we can take for granted “It is wet outside”.

The $\rightarrow \mathcal{I}$ rule says that if we assume A and from that assumption we deduce B, we can conclude that $A \rightarrow B$. Let us assume that Alan Turing did not see Alonzo Church’s work. From that assumption, we deduce that Alan Turing could not have stolen Alonzo Church’s work. Therefore we can conclude that “If Alan Turing did not see Alonzo Church’s work, he could not have stolen it”.

2.3.2 Classical Logic

First, we introduce the notation for a *sequent*:

$$\Gamma \vdash T$$

The Γ is a sequence of statements called the *antecedent*. The right-hand side of the $\vdash$ is also a sequence of statements called the *succedent*. The whole sequent denotes that the conjunction of the statements in the antecedent imply the disjunction of the statements in the succedent. That is to say, all the statements on the left taken together imply that at least one of the statements on the right is correct.

$$\begin{gathered} A_1, A_2, \ldots, A_n \vdash S_1, S_2, \ldots, S_m \\ \text{denotes} \\ A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow S_1 \vee S_2 \vee \cdots \vee S_m \end{gathered}$$

2.3.3 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:

Definition 2.3.1 (Grammar for constructing types)

Type variables are represented by the lower-case greek alphabet $\alpha, \beta, \gamma, ...$

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

2.3.4 Typed λ-Calculus

Typed systems have rules for assigning types for terms. Type assignment rules for the *typed* λ-calculus are:

$$(\to \mathcal{E})\ \frac{\Gamma \vdash M : A \to B \qquad N : A}{\Gamma \vdash MN : B}$$

$$(\to \mathcal{I})\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

The first rule states that the application of a term of type $A \to B$ to a term of type A has type B. The $\to \mathcal{I}$ rule says a term of type B in a context where $x : A$ is the same as an abstraction of type $A \to B$ in a context without $x : A$. These rules add restrictions on what constitutes a well-formed term.

Example 2.3.2 (Type-assignment restricts set of valid terms)

$$\begin{array}{l} (\lambda x.xx)(\lambda x.xx) \\ xx : B \\ x : A \to B \\ x : A \end{array}$$

The variable x is applied to a term so it must have type $A \to B$. The term it is applied to must have type A. However x is applied to itself so it must have type A and $A \to B$. This means the term $\lambda x.xx$ is untypable: it is disallowed by the rules of the typed λ-calculus.

2.3.5 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.

Looking again at the rules of the typed λ-calculus and IIL, there is a clear correspondence:

λ-calculus	IIL
$\frac{\Gamma \vdash M : A \to B \qquad N : A}{\Gamma \vdash MN : B}$	$\frac{A \to B \quad A}{B}$
$\frac{\Gamma \vdash \lambda x.M : A \to B}{\Gamma, x : A \vdash M : B}$	$\frac{A \to B}{[A]}$ $\vdots$ B

2.4 Haskell

2.4.1 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 Integers.

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.

2.4.2 Type Level/Value Level

Haskell distinguishes between terms on the type level and terms on the value level. This is the same as the separate layer of terms and types in the λ-calculus. Types in Haskell 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.

2.4.3 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
```

2.5 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 $M \to_\beta M'$

$$\begin{array}{lccc}
\textit{(Compound term)} & & MN & \\
\textit{(Decompose)} & M & & \Box N \\
\textit{(Beta-reduce dominant term)} & M' & & \Box N \\
\textit{(Refill hole of context)} & & M'N &
\end{array}$$

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*.

2.5.1 Undelimited Continuations

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

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

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**.

2.5.2 Delimited-Continuations

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

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

Figure 2.6: Decomposing a term into multiple contexts

Here, when a dominant term has been reduced, the reduct is returned to the continuation at the top of the stack. This newly joined term then becomes the dominant term. 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 us to use parts of the stack selectively for instance taking just top N contexts. The increased granularity of control means we can manipulate not just the entire remaining continuation but sections of it. Continuations of this kind are called **delimited continuations**.

2.5.3 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 2.5.1 (Translation of standard λ-terms into CPS)

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

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 $\lambda x.\textsc{display}\ x$ to display the results of the program.

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

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

The translation of standard λ-terms into CPS similarly transforms the *types* of λ-terms. For example, a term $x : A$ becomes $\lambda k.kx : (A \to B) \to B$. 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 $(A \to B) \to B$ again, it is clear that A is the type of the term passed to the continuation of a CPS-term:

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

2.5.4 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[3]:

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

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 `>>=`:

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

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

$$\mathit{return} : A \to \mathit{Comp}\ A$$

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 `>>=`.

2.6 $\lambda\mu$-Calculus

Michel Parigot wrote the $\lambda\mu$-calculus as a system with an isomorphism to classical logic. It is an extension of the λ-calculus. This means that the grammar and reduction rules of the $\lambda\mu$-calculus are a superset of those of the λ-calculus.

2.6.1 Syntax

Just as λ introduces λ-abstractions, $\lambda\mu$ introduces $\lambda\mu$-abstractions. 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.

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

Definition 2.6.1 (Grammar for $\lambda\mu$-calculus)

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

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

2.6.2 Reduction Rules

Definition 2.6.2 (Reduction rules for $\lambda\mu$-calculus)

$$\begin{array}{lrcll} i) & (\lambda x.M)N & \rightarrow_\beta & M[N/x] & \\ ii) & (\mu\alpha.[\beta]M)N & \rightarrow\mu & (\mu\alpha.[\beta]M[[\gamma]M'N/[\alpha]M']) & \\ iii) & \mu\alpha.[\alpha]M & \rightarrow\mu & M & (\alpha \notin fn(M)) \\ iv) & \mu\delta.[\beta](\mu\gamma.[\alpha]M) & \rightarrow\mu & \mu\delta.[\alpha]M[\beta/\gamma] & \end{array}$$

The terse reduction rule of *iii*) simply 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. It can be thought of as a λ-abstraction that can be applied to any number of variables. If we knew how many variables the term is applied to, we could replace

$$\mu\alpha \ldots [\alpha]M$$

with

$$\lambda x_1 \ldots \lambda x_n \ldots M x_1 \ldots x_n$$

2.6.3 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 $[\alpha]M$ 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 $\square N$. Informally, we can imagine that the $\lambda\mu$-variable α now maps to this context $\{\alpha \Rightarrow \square N\}$:

Example 2.6.3

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

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 $\mu \circ .[\alpha](\lambda s.fs)$: [4]

Example 2.6.4

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 2.6.5

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)

2.6.4 Head Reduction

The idea of head-reduction can be extended to the $\lambda\mu$-calculus. First, we extend the definition of head variables and add the definition of *head names*:

Definition 2.6.6 Head variables and head names hn for the $\lambda\mu$-calculus

$$\begin{array}{lcl} hv(\lambda x.M) & = & hv(M) \\ hv(xM_1 \ldots M_n) & = & x \\ hv(\mu\alpha.[\beta]M) & = & hv(M) \\ & & \\ hn(\mu\alpha.[\beta]M) & = & \beta \end{array}$$

Reduction rules can only be applied to the head variable. Additionally, μ-reduction rules are restricted to terms labelled with the head name:

[4] Following van Bakel, we use $\circ$ to denote a μ-variable that does not occur in the body of the $\lambda\mu$-abstraction.

Definition 2.6.7 Head reduction for $\lambda\mu$-calculus

$$
\begin{array}{lll}
M \to N & \Rightarrow & \{ \\
 & & \begin{array}{rcl}
\lambda x.M & \to_H & \lambda x.N \\
M\overline{L} & \to_H & N\overline{L} \\
\mu\alpha.[\beta]M & \to_H & \mu\alpha.[\beta]N \\
M[[\alpha]M'L/[\alpha]M'] & \to_H & N[[\alpha]M'L/[\alpha]M']
\end{array} \\
 & & \}
\end{array}
$$

These restricted rules define head reduction for the $\lambda\mu$-calculus. μH-normal form is defined as terms that cannot be μH-reduced any further. Terms in μH-normal form are those producible by the following grammar:

Definition 2.6.8 Grammar for $\lambda\mu$-terms in $\beta\mu H$-normal form.

$$
\begin{array}{lcll}
N & ::= & \lambda x.N & \\
 & | & xM_1 \dots M_2 & \\
 & | & \mu\alpha.[\beta]N & (\alpha \neq \beta \vee \alpha \in N, N \neq \mu\gamma.[\delta]N') \\
 & | & N[[\alpha]ML/[\alpha]M] & (hn(N) \neq \alpha)
\end{array}
$$

The head reduction of $\lambda\mu$-terms ensures only the left-most and outermost term is reduced. We define $\to_{\mu H}^{nf}$ as the reduction to normal form with respect to $\to_{\mu H}$.

2.7 Delimited-Continuation Calculus

Simon Peyton-Jones *et al.* extended the λ-calculus with additional operators in order create a framework for implementing delimited continuations [1]. 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.

2.7.1 Syntax

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

Definition 2.7.1 (Grammar for DCC)

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

2.7.2 Reduction Rules

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

Definition 2.7.2 (Operational semantics for DCC)

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

2.7.3 Significance

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.

The abstract machine defined in Figure 2.7.2 also encodes the reduction strategy. The first block of rules define in what order redexes are reducted.

2.8 λ^{try}-Calculus

Steffen van Bakel extended the λ-calculus with operators for modelling exceptions. Unlike previous systems, the λ^{try}-calculus uses named exceptions.

2.8.1 Exceptions

Exceptions in programming languages are indications that control flow cannot continue. For example, if you attempt to open a non-existent file, the operation might throw an exception. If an exception occurs without being caught, a program will exit with an error. Exceptions can be caught and attempts at recovery can be made by exception handlers.

The common syntax for introducing exception handlers is in try-catch blocks. An exception that occurs in a try-catch block will be handled by a corresponding handler. For example, see the Javascript syntax for this:

```
try {
  /* possibly throw exception */
} catch (e) {
  /* recover thrown exception */
}
```

The `catch (e) { ... }` introduces a single exception handler. This exception handler will be called if an exception is thrown inside the try block. If we want to introduce multiple exception handlers, we need a mechanism for deciding which handler will be called. Java solves this by registering different exception handlers based on their type:

```
try {
  /* possibly throw exception */
} catch (IOException e) {
  /* recover from IOException */
} catch (FileNotFoundException e) {
  /* recover from FileNotFoundException */
}
```

Here, which handler is called depends on the type of the exception thrown.

2.8.2 Syntax

The grammar of the λ^{try}-calculus is as follows:

Definition 2.8.1 (Grammar of the λ^{try}-calculus)

$$\begin{array}{rcl} C & ::= & \text{catch }\ \mathrm{n}_1(x) = M_1\ ; \ldots ; \text{catch } \mathrm{n}_i(x) = M_i \quad (i \geq 1) \\ M, N & ::= & x \mid \lambda x.M \mid MN \mid \text{try } M;\ C \mid \text{throw } \mathrm{n}(M) \end{array}$$

The grammar for C describes a catch block as series of one or more catch statements. For convenience, we will use the notation

$$\overline{\text{catch } \mathrm{n}_i(x) = M_i} i > 1$$

to describe a catch block with more than one catch statement.
The λ^{try}-calculus adds three new syntactic constructs:

- **throw n(**M**)** denotes the throwing of an exception with name n passing it the value M.
- **catch n(**x**)** $=$ M registers the exception handler M to the name n with parameter x.
- **try** M**;** C attempts to run term M in an environment with the exception handlers in catch block C registered.

2.8.3 Reduction Rules

In conjunction with the additional syntactic constructions, the λ^{try}-calculus introduces some reduction rules:

Definition 2.8.2 (λ^{try} REDUCTION RULES)

$$\begin{array}{rlcl}
(\beta)\text{:} & (\lambda x.M)N & \to & M[N/x] \\
(\text{throw})\text{:} & (\text{throw } \mathrm{n}(M))N & \to & \text{throw } \mathrm{n}(M) \\
(\text{try-throw})\text{:} & \text{try throw } \mathrm{n}_l(N);\ \overline{\text{catch } \mathrm{n}_i(x) = M_i} & \to & M_l[N/x] \\
(\text{try-value})\text{:} & \text{try } V;\ \overline{\text{catch } \mathrm{n}_i(x) = M_i} & \to & V
\end{array}$$

The β reduction rule is familiar from the λ-calculus. A **throw** term applied to any term discards the second term. A **try** term that contains a throw reduces to the handler that corresponds to the name of the exception thrown with all occurrences of the parameter replaced by the value thrown. For instance a throw n(N) inside a **try** will reduce to $M[N/x]$ if there is a catch $\mathrm{n}(x) = M$ in the catch block. A **try** term that contains a value reduces to just that value.

2.8.4 Significance

The occurrence of an exception aborts the current computation. λ^{try} models this by discarding terms that a **throw** is applied to. The **try-catch** statements mirror the syntax of try-catch statements in programming languages in the C-syntax family.

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.

3.1 Interpreter

Although Peyton-Jones *et al.* implement a language-level module for DCC, 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 DCC. 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.

3.2 Implementation

3.2.1 Data structures

There are two data types for representing DCC terms, `Value` and `Expr` (Figure 3.2.1). 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 $\langle e,\ D,\ E,\ q\rangle$:

3.2.2 Utility Functions

Some utility functions are simplify the implementation. Informally, these functions behave as follows (see Figure 3.2.2 for implementations details):

```
data Value = Var Char
  | Abs Char Expr
  | Prompt Int
  | Seq [Expr]
  deriving (Show, Eq)

data Expr = Val Value
  | App Expr Expr
  | Hole
  | PushPrompt Expr Expr
  | PushSubCont Expr Expr
  | WithSubCont Expr Expr
  | NewPrompt

  | Sub Expr Expr Char
  deriving (Show, Eq)

data State = State Expr Expr [Expr] Value
  deriving (Show, Eq)
```

- `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 $M[v/x]$.

3.2.3 Reduction Rules

The heavy lifting of the interpreter is done by the function `eval :: State -> State`. `eval` is defined inductively on the structure of DCC terms. Using pattern-matching, each case of `eval` corresponds directly to at least one of the reduction rules of the DCC abstract machine. For the full implementation of `eval`, see Figure 3.2.3

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.

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.

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.

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.

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

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

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.

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

```
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 3.1: Utility functions for DCC interpreter

```
eval :: State -> State
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
      Seq es' -> State (ret (composeContexts es') e') d es q
      otherwise -> State (App e e') d es q
    otherwise -> State e' (ret d (App e Hole)) es q
  otherwise -> State e (ret d (App Hole e')) es q

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

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

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

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

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

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
```

4 — Translations

In this chapter, we develop a interpretation of $\lambda\mu$ in DCC. We prove some properties of this interpretation, including *soundness.* We concatenate this interpretation with van Bakel's interpretation of λ^{try} in $\lambda\mu$. This concatenation yields an interpretation of λ^{try} in DCC. This will then be used as a basis for the implementation of λ^{try} in Haskell.

4.1 Interpreting λ^{try} in $\lambda\mu$

Steffen van Bakel describes the interpretation of λ^{try} to $\lambda\mu$:

$$
\begin{array}{rcl}
\llbracket x \rrbracket & \triangleq & x \\
\llbracket \lambda x.M \rrbracket & \triangleq & \lambda x.\llbracket M \rrbracket \\
\llbracket MN \rrbracket & \triangleq & \llbracket M \rrbracket \llbracket N \rrbracket \\
\llbracket \text{try } M;\ \overline{\text{catch } \mathrm{n}_i(x) = M_i};\ \text{catch } \mathrm{m}(x) = L \rrbracket & & \\
& \triangleq & \\
& & (\lambda c_m.\mu \mathrm{m}.[\mathrm{m}]\llbracket \text{try } M;\ \overline{\text{catch } \mathrm{n}_i(x) = M_i} \rrbracket)(\lambda x.\llbracket L \rrbracket) \\
\llbracket \text{try } M;\ \text{catch } \mathrm{m}(x) = L \rrbracket & \triangleq & (\lambda c_m.\mu \mathrm{m}.[\mathrm{m}]\llbracket M \rrbracket)(\lambda x.\llbracket L \rrbracket) \\
\llbracket \text{throw } \mathrm{n}(M) \rrbracket & \triangleq & \lambda \circ .[\mathrm{n}]c_n \llbracket M \rrbracket
\end{array}
$$

throw n(M) terms are modelled using $\lambda\mu$-abstractions of non-occurring names. This has the effect of removing all terms it is applied to:

$$(\mu \circ .M)NOP \to (\mu \circ .M)OP \to (\mu \circ .M)P \to \mu \circ .M$$

The contents of the $\lambda\mu$-abstraction calls c_n. This λ-variable is bound by the translation of **try** terms. This binding means that the exception handlers, represented by $\lambda x.\llbracket L \rrbracket$, are in scope for the reduction of the body of the try M.

4.2 Interpreting $\lambda\mu$ in DCC

The translation of $\lambda\mu$-terms into DCC assumes that there is a single global prompt P_0. It also assumes that this prompt has already been pushed onto the stack. This means that the translation of a full $\lambda\mu$-program M in DCC is:

Definition 4.2.1 (Initialization of stack for running M in DCC)

$$(\lambda\textsc{p}_0.\textsc{pp}\ \textsc{p}_0\ [\![M]\!])\ \textsc{np}$$

This creates a new prompt $\textsc{p}_0$ which is in scope for all terms in M. It also prepares the stack by pushing $\textsc{p}_0$ immediately. With the stack prepared, the interpretation of $\lambda\mu$ terms into DCC proceeds as follows:

Definition 4.2.2 (Interpretation of $\lambda\mu$ into DCC)

$$\begin{array}{lcl}
[\![x]\!] & \triangleq & x \\
[\![\lambda x.M]\!] & \triangleq & \lambda x.[\![M]\!] \\
[\![MN]\!] & \triangleq & [\![M]\!][\![N]\!] \\
[\![\mu\alpha.M]\!] & \triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ [\![M]\!] \\
[\![[\beta]M]\!] & \triangleq & \textsc{psc}\ \beta\ [\![M]\!]
\end{array}$$

To implement $\lambda\mu$-abstractions, we capture the subcontinuation until the last occurrence of $\textsc{p}_0$ on the stack. This subcontinuation is bound to α which ensures the subcontinuation is distributed to all occurrences of α in M. $\textsc{p}_0$ is then pushed back onto the stack before the evaluation of M.

To implement named-terms, the subcontinuation β is pushed into the stack before evaluating M. This means the reduct of M will be returned to this subcontinuation. In effect, this reduces M and passes the result to β.

Example 4.2.3 $[\![\mu\alpha.[\alpha](\lambda x.x)]\!] \to [\![\lambda x.x]\!]$

$$\begin{array}{llll}
 & [\![\mu\alpha.[\alpha](\lambda x.x)]\!] & & \\
\triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ \lambda x.x, & \Box, & \textsc{p}_0 : [] \\
\to_\beta & (\lambda\alpha.\textsc{pp}\ \textsc{p}_0\ \lambda x.x)(\Box), & \Box, & [] \\
\to_\beta & (\textsc{pp}\ \textsc{p}_0\ \lambda x.x)[\Box/\alpha], & \Box, & [] \\
\to_\beta & \textsc{pp}\ \textsc{p}_0\ \lambda x.x, & \Box, & [] \\
\to_\beta & \lambda x.x, & \Box, & \textsc{p}_0 : []
\end{array}$$

The final state has restored the initial state of the stack by pushing $\textsc{p}_0$ back on.

Theorem 4.2.4 (Soundness of $[\![\bullet]\!]$) *If $M \to_\mu N$ then $[\![M]\!] \to_{DCC} [\![N]\!]$*

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

$$(\lambda x.M)N \to M[N/x] :$$

$$\begin{array}{llll}
 & [\![(\lambda x.M)N]\!] & & \\
\triangleq & [\![(\lambda x.M)]\!][\![N]\!] & & \\
\triangleq & (\lambda x.[\![M]\!])[\![N]\!], & \Box, & \textsc{p}_0 : [] \\
\to_\beta & [\![M]\!][[\![N]\!]/x], & \Box, & \textsc{p}_0 : [] \\
\triangleq & [\![M[[\![N]\!]/x]]\!], & \Box, & \textsc{p}_0 : []
\end{array}$$

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

$$
\begin{array}{rlrr}
 & \llbracket(\mu\alpha.[\beta]M)N\rrbracket & & \\
\triangleq & \llbracket(\mu\alpha.[\beta M)\rrbracket\llbracket N\rrbracket & & \\
\triangleq & (\textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket))\llbracket N\rrbracket, & \square, & \mathrm{P}_0 : [] \\
\to_{DCC} & \textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket), & \square\llbracket N\rrbracket, & \mathrm{P}_0 : [] \\
\to_{DCC} & (\lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket))(\square\llbracket N\rrbracket), & \square, & [] \\
\to_{\beta} & (\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket))[\square\llbracket N\rrbracket/\alpha], & \square, & [] \\
\to_{\beta} & \textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ (\llbracket M\rrbracket[\square\llbracket N\rrbracket/\alpha])), & \square, & [] \\
\to_{DCC} & \textsc{psc}\ \beta\ (\llbracket M\rrbracket[\square\llbracket N\rrbracket/\alpha]), & \square, & \mathrm{P}_0 : [] \\
\triangleq & \llbracket\mu\alpha.([\beta]M)[[\alpha]M'N/[\alpha]M']\rrbracket & &
\end{array}
$$

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

$$
\begin{array}{rlrr}
 & \llbracket\mu\alpha.[\alpha]M\rrbracket & & \\
\triangleq & \textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M\rrbracket), & \square, & \mathrm{P}_0 : [] \\
\to_{DCC} & \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M\rrbracket)(\square), & \square, & [] \\
\to_{\beta} & \textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M\rrbracket)[\square/\alpha], & \square, & [] \\
\to_{\beta} & \textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \square\ (\llbracket M\rrbracket[\square/\alpha]), & \square, & [] \\
\to_{DCC} & \textsc{psc}\ \square\ (\llbracket M\rrbracket[\square/\alpha], & \square, & \mathrm{P}_0 : [] \\
\to_{DCC} & \llbracket M\rrbracket[\square/\alpha], & \square, & \mathrm{P}_0 : [] \\
\triangleq & \llbracket M\rrbracket & &
\end{array}
$$

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

$$
\begin{array}{rlrr}
\triangleq & \textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ \llbracket[\beta]\mu\gamma.[\delta]M\rrbracket, & \square, & \mathrm{P}_0 : [] \\
\to_{DCC} & \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ \llbracket[\beta]\mu\gamma.[\delta]M\rrbracket(\square), & \square, & [] \\
\to_{\beta} & (\textsc{pp}\ \mathrm{P}_0\ \llbracket[\beta]\mu\gamma.[\delta]M\rrbracket)[\square/\alpha], & \square, & [] \\
\to_{\beta} & \textsc{pp}\ \mathrm{P}_0\ \llbracket[\beta]\mu\gamma.[\delta]M\rrbracket[\square/\alpha], & \square, & [] \\
\to_{\beta} & \llbracket[\beta]\mu\gamma.[\delta]M\rrbracket[\square/\alpha], & \square, & \mathrm{P}_0 : [] \\
\triangleq & (psc\beta\llbracket\mu\gamma.[\delta]M\rrbracket)[\square/\alpha], & \square, & \mathrm{P}_0 : [] \\
\triangleq & \llbracket\mu\gamma.[\delta]M\rrbracket)[\square/\alpha], & \square, & \beta : \mathrm{P}_0 : [] \\
\triangleq & (\textsc{wsc}\ \mathrm{P}_0\ \lambda\gamma.\llbracket[\delta]M\rrbracket)[\square/\alpha], & \square, & \beta : \mathrm{P}_0 : [] \\
\triangleq & \textsc{wsc}\ \mathrm{P}_0\ \lambda\gamma.\textsc{pp}\ \mathrm{P}_0\ \llbracket[\delta]M\rrbracket[\square/\alpha], & \square, & \beta : \mathrm{P}_0 : [] \\
\triangleq & (\lambda\gamma.\textsc{pp}\ \mathrm{P}_0\ \llbracket[\delta]M\rrbracket[\square/\alpha])(\beta), & \square, & [] \\
\triangleq & (\textsc{pp}\ \mathrm{P}_0\ \llbracket[\delta]M\rrbracket[\square/\alpha])[\beta/\gamma], & \square, & [] \\
\triangleq & \textsc{pp}\ \mathrm{P}_0\ (\llbracket[\delta]M\rrbracket[\square/\alpha])[\beta/\gamma], & \square, & [] \\
\triangleq & \llbracket[\delta]M\rrbracket[\square/\alpha])[\beta/\gamma], & \square, & \mathrm{P}_0 : [] \\
\triangleq & (\textsc{psc}\ \delta\ M[\square/\alpha][\beta/\gamma], & \square, & \mathrm{P}_0 : [] \\
\triangleq & \textsc{psc}\ \delta\ (M[\square/\alpha])[\beta/\gamma], & \square, & \mathrm{P}_0 : [] \\
\triangleq & \llbracket\mu\alpha.[\delta](M[\beta/\gamma])\rrbracket, & \square, & \mathrm{P}_0 : []
\end{array}
$$

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

$$
\begin{array}{rlll}
 & \llbracket \mu\alpha.[\beta]\mu\gamma.[\delta]M \rrbracket & & \\
\triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ \llbracket [\beta]\mu\gamma.[\delta]M \rrbracket, & \Box, & \textsc{p}_0 : [] \\
\to_{DCC} & \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ \llbracket [\beta]\mu\gamma.[\delta]M \rrbracket(\Box), & \Box, & [] \\
\to_{\beta} & (\textsc{pp}\ \textsc{p}_0\ \llbracket [\beta]\mu\gamma.[\delta]M \rrbracket)[\Box/\alpha], & \Box, & [] \\
\to_{\beta} & \textsc{pp}\ \textsc{p}_0\ \llbracket [\beta]\mu\gamma.[\delta]M \rrbracket[\Box/\alpha], & \Box, & [] \\
\to_{\beta} & \llbracket [\beta]\mu\gamma.[\delta]M \rrbracket[\Box/\alpha], & \Box, & \textsc{p}_0 : [] \\
\triangleq & (psc\beta\llbracket \mu\gamma.[\delta]M \rrbracket)[\Box/\alpha], & \Box, & \textsc{p}_0 : [] \\
\triangleq & \llbracket \mu\gamma.[\delta]M \rrbracket)[\Box/\alpha], & \Box, & \beta : \textsc{p}_0 : [] \\
\triangleq & (\textsc{wsc}\ \textsc{p}_0\ \lambda\gamma.\llbracket [\delta]M \rrbracket)[\Box/\alpha], & \Box, & \beta : \textsc{p}_0 : [] \\
\triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\gamma.\textsc{pp}\ \textsc{p}_0\ \llbracket [\delta]M \rrbracket[\Box/\alpha], & \Box, & \beta : \textsc{p}_0 : [] \\
\triangleq & (\lambda\gamma.\textsc{pp}\ \textsc{p}_0\ \llbracket [\delta]M \rrbracket[\Box/\alpha])(\beta), & \Box, & [] \\
\triangleq & (\textsc{pp}\ \textsc{p}_0\ \llbracket [\delta]M \rrbracket[\Box/\alpha])[\beta/\gamma], & \Box, & [] \\
\triangleq & \textsc{pp}\ \textsc{p}_0\ (\llbracket [\delta]M \rrbracket[\Box/\alpha])[\beta/\gamma], & \Box, & [] \\
\triangleq & \llbracket [\delta]M \rrbracket[\Box/\alpha])[\beta/\gamma], & \Box, & \textsc{p}_0 : [] \\
\triangleq & (\textsc{psc}\ \delta\ M[\Box/\alpha][\beta/\gamma], & \Box, & \textsc{p}_0 : [] \\
\triangleq & \textsc{psc}\ \beta\ (M[\Box/\alpha])[\beta/\gamma], & \Box, & \textsc{p}_0 : [] \\
\triangleq & \llbracket \mu\alpha.[\beta](M[\beta/\gamma]) \rrbracket, & \Box, & \textsc{p}_0 : []
\end{array}
$$

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

$$
\begin{array}{rlll}
 & \llbracket (\mu\delta.[\alpha]M)[[\gamma]M'N/[\alpha]M'] \rrbracket & & \\
\triangleq & (\textsc{wsc}\ \textsc{p}_0\ \lambda\delta.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \alpha\ M))[\Box N/\alpha] & \Box, & \textsc{p}_0 : [] \\
\to_{\beta} & \textsc{wsc}\ \textsc{p}_0\ \lambda\delta.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \Box N\ (M[\Box N/\alpha]))) & \Box, & \textsc{p}_0 : [] \\
\triangleq & \llbracket \mu\delta.[\alpha](M[[\alpha]M'N/[\alpha]M'])N \rrbracket & &
\end{array}
$$

$M[[\alpha]M'N/[\alpha]M] \to M \quad (\alpha \notin \mathrm{fn}(M))$:

$$
\begin{array}{rll}
 & \llbracket M[[\alpha]M'N/[\alpha]M] \rrbracket & \\
\triangleq & \llbracket M \rrbracket[\Box N/\alpha] & \\
\to_{\beta} & \llbracket M \rrbracket & (\alpha \notin \mathrm{fv}(M))
\end{array}
$$

□

This means that the translation of M reduces a term Q that relates to the untranslated reduct of N, given $M \to N$. The relation is either that Q reduces to N, is the same as N, or N reduces to Q.

Theorem 4.2.5 (Completeness of $\llbracket \bullet \rrbracket$)

We attempt to prove one of the following properties:

1. $\llbracket M \rrbracket \to Q \Rightarrow \exists N.M \to N \wedge Q \to^* \llbracket N \rrbracket$

2. $[\![M]\!] \rightarrow^{nf} Q \Rightarrow \exists N.M \rightarrow^{*} N \wedge [\![N]\!] = Q$

3. $[\![M]\!] \rightarrow^{nf} Q \Rightarrow \exists N.M \rightarrow^{*} N \wedge [\![N]\!] \rightarrow^{nf} Q$

4.3 Interpreting λ^{try} in DCC

By appending the interpretation of λ^{try} in $\lambda\mu$ with the interpretation of $\lambda\mu$ in DCC, we get a translation from λ^{try} to DCC:

Definition 4.3.1 Translation of λ^{try} into DCC

$$\begin{array}{rcl}
[\![x]\!] & \triangleq & x \\
[\![\lambda x.M]\!] & \triangleq & \lambda x.[\![M]\!] \\
[\![MN]\!] & \triangleq & [\![M]\!][\![N]\!] \\
[\![\text{throw n}(M)]\!] & \triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda \circ .\textsc{pp}\ \textsc{p}_0\ (psc\ n\ (c_n\ [\![M]\!])) \\
[\![\text{try } M;\ \text{catch n}(x) = L]\!] & \triangleq & (\lambda c_n.\textsc{wsc}\ \textsc{p}_0\ \lambda n.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ n\ [\![M]\!]))(\lambda x.[\![L]\!])
\end{array}$$

$$[\![\text{try } M;\ \overline{\text{catch n}_i(x) = M_i};\ \text{catch m}(x) = L]\!]$$

$$\triangleq$$

$$(\lambda c_m.\textsc{wsc}\ \textsc{p}_0\ \lambda m.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ m\ [\![\text{try } M;\ \overline{\text{catch n}_i(x) = M_i}]\!]))(\lambda x.[\![L]\!])$$

5 — Implementation

This chapter explains the implementation of λ^{try} in Haskell. It examines how the translations from the previous chapter and the DCC library published in [1] are transformed into Haskell code. Finally, it explores the possibility of a generalized Haskell library.

5.1 DCC Library

5.1.1 Missing Typeclass Instances

5.2 Single Naive Implementation

Blindly transcribing the translation yields a naive implementation in Haskell. This implementation is a functioning proof-of-concept that the translation works as intended. It does not present a high level of abstraction, requiring programs to be manually translated to Haskell:

```
try :: Prompt ans b
      -> ((t -> CC ans a) -> CC ans a1) -> (t -> CC ans a1)
      -> CC ans a1
try p0 m handler = withSubCont p0 (\n ->
    pushPrompt p0 (pushSubCont n (m $ \x ->
      throw p0 n handler x)))

throw :: Prompt ans b
         -> SubCont ans a1 b -> (t -> CC ans a1) -> t
         -> CC ans a
throw p0 n c m = withSubCont p0 (\_ ->
  pushPrompt p0 (pushSubCont n (c m)))
```

This short implementation: - adds explicit p0 argument required by wsc and pp. - requires program to be run inside:

runCC with

```
example1 = runCC (do
  p <- newPrompt
  pushPrompt p (testy p (\t -> return 5) (\x -> return $ x+1)))

example2 = runCC (do
  p <- newPrompt
  pushPrompt p (testy p (\t -> (t 4)) (\x -> return $ x+1)))
```

5.2.1 Issues

Remark: wrap throw in a function to solve scoping issues

Impedance mismatch between DCC and λ^{try}-calculus. We only need a single prompt. This prompt should be triggered on beginning of "try" statement. Prompt should be reused for setting up all catch handlers statements

Access to each handler requires explicitly passing catch names to solve scoping. This means that we have to define try, try1, try2, etc for number of handlers: no approach for variadic handlers.

Type signature exposes CC, Prompt, etc data constructors. We want to abstract over these so user does not rely on knowing them. Leaky abstractions.

5.3 Improved Implementation

The improved version creates a single prompt for the try block. The throw operation is defined in scope of the prompt, removing the need to add an additional argument.

We are still required to call runCC to return the result. Although handlers can be ordinary functions (see examples), the body must `return` a value if not calling one of the exception handlers. Also, multiple handlers still require multiple function instances:

```
try m handler = do
  p <- newPrompt
  let throw n = \v -> withSubCont p (\_ ->
            pushPrompt p (pushSubCont n $ return v))
  pushPrompt p (withSubCont p (\n ->
    pushPrompt p (pushSubCont n
      (m (thr n . handler)))))

try2 m h1 h2 = do
  p <- newPrompt
  let throw n = \v -> withSubCont p (\_ ->
            pushPrompt p (pushSubCont n $ return v))
  pushPrompt p (withSubCont p (\n ->
    pushPrompt p (pushSubCont n
      (m (thr n . h1) (thr n . h2)))))
```

```
handler1 = (+2)
handler2 = (+4)

tryExample1 = try (\t -> return 1) handler1
tryExample2 = try (\t -> t 1) handler1

try2Example1 = try2 (\t1 -> \t2 -> return 1) handler1 handler2
try2Example2 = try2 (\t1 -> \t2 -> t1 1) handler1 handler2
try2Example3 = try2 (\t1 -> \t2 -> t2 1) handler1 handler2
```

If we look at the type signatures, we can see the same important restrictions from the naive implementation. The handlers must produce values of the same type. This will also be the type of running `runCC` on the returned `CC` value.

```
  try  :: ((a -> CC ans a1) -> CC ans a2) -> (a -> a2) -> CC ans a2
  try2 :: ((a -> CC ans a1) -> (a2 -> CC ans a3) -> CC ans a4)
         -> (a -> a4) -> (a2 -> a4) -> CC ans a4
```

6 — Conclusion

6.1 Evaluation

6.2 Conclusion

6.3 Future Work

- *Type preservation* – check whether types are preserved across the translations
- *Haskell library* – write a general purpose library for named exceptions in Haskell
- *Reimplement DCC without stack* – The implementation of $\lambda\mu$ in DCC only ever uses a single prompt: DCC is overengineered for the purposes of a $\lambda\mu$-translation. Rewrite DCC using a mapping from prompts to contexts.

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.