Imperial College London

Department of Computing

Exception Handling in Haskell Using the λ^{try}-Calculus

by
William S. Fisher
`wsf15@ic.ac.uk`

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

We present a language-level extension to Haskell for exception handling. We outline a series of translations between formal systems, starting from the λ^{try}-calculus. Unlike other approaches in either software or formal systems, the λ^{try}-calculus uses named exception handlers. We define a term-rewriting interpreter in Haskell for interpreting terms from the Calculus of Delimed Continuations. This interpreter produces the reduction steps required to prove the soundness and completeness of our translations. Finally, we explore both a proof-of-concept, library-level implementation and a language-level extension that allow for named exception handling in Haskell, and are based on the λ^{try}-calculus.

Acknowledgements

Firstly, I want to say thank you to Steffen van Bakel for his steady reassurance, for directing my excitement into productive channels of inquiry, and – most importantly – for bringing me in from the cold, into the warm shelter of formal proofs, type theory, and computational calculi. I would like to thank Rafe for getting me excited about maths and the λ-calculus, for wandering through the darkness with me, and for helping me find intuitions when all I could find were headaches. I would like to thank Caitlin for always keeping me happy and sane, and for reminding me to look after myself. Thank you (and sorry) to my parents for their patience with me, I'm sure this has been just as hard for them.

Contents

1 — Introduction

In 1936 Alonzo Church presented the λ-calculus, a formal system for expressing the entire set of computable problems [2].[1] Later, Haskell Curry and William Howard both discover a correspondence between the simply-typed λ-calculus and intuitionistic logic. However the correspondence runs deeper than this: there is actually a true isomorphism between many systems of formal logic and computational calculi.

The correspondence between systems of formal logic and computer programs drove research into both the logical counterparts of programs and the computational counterparts of logical systems. As a part of this, Michel Parigot developed the $\lambda\mu$-calculus, an extension of the λ-calculus, through an isomorphism to classical logic [8]. More recently, Steffen van Bakel modelled exceptions and exception handling through an extension of the λ-calculus called the λ^{try}-calculus [10].[2]

Exceptions are most commonly found in two kinds: systems where there is a single type of exception such as in Javascript, and systems where exceptions are discriminated by *type* as in Java and Haskell. Unlike either of these approaches, the exception handling proposed by the λ^{try}-calculus introduces exceptions discriminated by *name*.

Project

This paper proposes an implementation of named exceptions in Haskell. First we define a translation from the $\lambda\mu$-calculus to a calculus for expressing delimited continuations, CDC [4].

$$\lambda\mu \quad \rightarrow \quad CDC$$

CDC is a calculus which closely models Haskell's syntax and which has been extended with operations for manipulating delimited continuations. The CDC library for Haskell defined by Jones *et al.* in [4] evaluates CDC terms in a single step, returning the final value of the reduction. However, we are interested in producing the entire step-by-step evaluation of CDC terms which we can transliterate into CDC derivations for verifying translations of $\lambda\mu$ into CDC. This motivates us to write a term-rewriting program for evaluating CDC terms. The proofs generated by this program are used to investigate the soundness and completeness of the $\lambda\mu \rightarrow CDC$ translation. The translation is then concatenated with van Bakel's original translation of his λ^{try}-calculus into the $\lambda\mu$-calculus which yields a translation from λ^{try} to CDC:

$$(\lambda^{try} \rightarrow \lambda\mu) \circ (\lambda\mu \rightarrow CDC) = \lambda^{try} \rightarrow CDC$$

We use this final translation to explore implementations of named exceptions directly in Haskell.

[1]The ability of a system to express the entire set of computable problems is now commonly known as 'Turing completeness' although here this description would be anachronistic; Turing's work on computability was published a year after Church's.

[2]Exceptions are a form of error generated by computers to indicate that the flow of control cannot continue as expected and must be aborted.

Outline

Chapter 2 discusses the related context for this project. Following this, this paper makes the following original contributions:

- A Haskell interpreter for the calculus of delimited continuations, CDC (in Chapter 3)
- A translation of $\lambda\mu$ into CDC along with proofs of its soundness and completeness with respect to μ-reduction (in Chapter 4)
- A translation of λ^{try} into CDC (also in Chapter 4)
- A proof of concept implementation of λ^{try} in Haskell, based on this translation (in Chapter 5)
- The specification for a language extension for Haskell with named exceptions, based on λ^{try} (also Chapter 5)

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 on which the rest of this project is built.*

2.1 Formal Systems

Formal systems are sets of rules for writing and manipulating formulæ. Formulæ are constructed from a set of characters called the *alphabet* by following some formulæ-construction rules called the *grammar*. The only formulæ 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 formulæ that are well formed. Formulæ produced according to the grammar of a system are well formed according to the syntax of that system.

Grammars are normally defined by 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:

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

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

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

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

2.1.2 Derivation Rules

Whereas a grammar describes the rules for producing well-formed formulæ, the derivation rules describe rules for transforming formulæ of a particular form into new formulæ. Using the grammar from Figure 2.1.1, we add derivation (rewriting) rules:

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

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 formulæ of the system to create new formulæ. Without going into detail, it is easy to verify that the tf-system is isomorphic to a subset of Boolean algebra:

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

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.

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

The left-hand side is the dominant term and the right-hand side is the context. The $\square$ 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. For example, we start with the term:

$$t\ a\ f\ a\ t$$

This term is decomposed into a dominant term (on the left) and a context (on the right):

$$t\ a\ f \qquad \square\ a\ t$$

Using the derivation rules from above, we can rewrite the dominant term $t\ a\ f$ as f:

$$f \qquad \square\ a\ t$$

Finally, once the dominant term cannot be reduced further, we recombine it with the context to get:

$$f\ a\ t$$

However the the term $t\ a\ f\ a\ t$ can be decomposed in two ways:

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

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. For compound terms, derivation strategies decide which derivation rule to applied to which subterm. In our tf-system, there are two obvious derivation strategies: either apply derivations starting from the left or starting from the right.

$$\begin{array}{rll} \textbf{(left)} & t\ a\ f\ a\ t & \\ \rightarrow & t\ a\ f & \Box\ a\ t \\ \rightarrow & f & \Box\ a\ t \\ \textit{recombine} & f\ a\ t & \end{array}$$

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

Whereas derivation rules are defined in the system, derivation strategies are strategies *about* the system: methods for 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 [1]. It is a formal system capable of expressing the set of effectively-computible algorithms. Shortly after, Gödel and Turing created their own models of effective computibility. [1] These models were later proved to be equivalent.

2.2.1 Syntax

λ-variables are taken from an infinite set and are written x, y, z, etc. Variables are open positions in a term that can be filled by other terms. The grammar for constructing well-formed λ-terms is:

Definition 2.2.1 (Grammar for untyped λ-calculus)

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

$$\begin{array}{rcll} M, N & ::= & x & \text{(Variable)} \\ & | & \lambda x.M & \text{(Abstraction)} \\ & | & (M\ N) & \text{(Application)} \end{array}$$

λ-abstractions are anonymous functions and 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:

[1]General recursive functions and Turing machines, respectively

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

Figure 2.2: Examples of valid λ-abstractions

Applications are represented by any two terms, both constructed according to the grammar, placed alongside one another. We assume the convention that application gives highest precedence to the left-most application. This means we can drop brackets where our convention will prevent ambiguity. Note that this means that if the term is an application, the outermost brackets can always be dropped. For example, we can drop the outermost, leftmost brackets from the term $((xy)z)$ and instead write xyz. To ensure the application of terms in any other order, we must leave the brackets explicit. For example the brackets in the term $x(yz)$ cannot be removed without changing the meaning of the term. Examples of applications are:

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

Figure 2.3: Examples of valid applications

2.2.2 Reduction Rules

The reduction rules for the λ-calculus are like the derivation rules from formal systems. They determine how terms of a particular form can be transformed into new terms.

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 2.2.2 (Substitution notation for λ-terms)

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

β-reduction

The main derivation rule of the λ-calculus is β-reduction. If term M β-reduces to term N, we write $M \to_\beta N$ 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) [10]
i) $\to_\beta$ is defined as the compatible closure of the rule:

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

ii) $\to_\beta^*$ is defined as the transitive closure of $\to_\beta$

λ-variables and λ-abstactions are *values*: they do not reduce to other terms. If a term is a value, the reduction cannot be applied to that term: it has *terminated* on that value. Only applications whose left-hand side is an abstraction can reduce to other terms. This means that an application of this form is called a reducible expression or a *redex.* Reducing a redex models the computing of a function.

α-reduction

The variables in a λ-term are either *bound* or *free.* The bound variables of a term are those introduced by λ-abstractions before their use.

Definition 2.2.4 BOUND VARIABLES bv OF λ-TERMS [10]

$$\begin{array}{lcl} bv(x) & = & \varnothing \\ bv(\lambda x.M) & = & bv(M) \cup \{x\} \\ bv(MN) & = & bv(M) \cup bv(N) \end{array}$$

Definition 2.2.5 FREE VARIABLES fv OF λ-TERMS [10]
The free variables of a term M are variables that occur in M that are not bound.

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 2.2.6 (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 does not already appear in M

Returning to the term above, we can α-reduce the right-hand term to reduce ambiguity before β-reducing the whole term:

$$\begin{array}{ll} & (\lambda y.\lambda x.xy)(\lambda x.x) \\ \rightarrow_\alpha & (\lambda y.\lambda x.xy)(\lambda z.z) \quad (z \neq x \wedge z \neq y) \\ \rightarrow_\beta & \lambda x.x(\lambda z.z) \end{array}$$

Using α-reduction, we can ensure that the binding of every variable in a term is unambiguous.

2.2.3 Reduction Strategies

A term is in normal form for a reduction strategy if, following that reduction strategy, no more reductions can take place. For instance, a term that can no longer be β-reduced is in β-normal form. We use $\rightarrow_{\beta}^{nf}$ to denote a term in β-normal form. Again, we can omit the β subscript if it is clear from context.

Lazy reduction is a reduction strategy that restricts the reduction to the outermost, leftmost redex. Any other subterms that are redexes cannot be reduced.

Definition 2.2.7 Lazy reduction of λ-terms
i) The outermost, leftmost term is the only one that can be reduced
ii) No other reductions can take place, including underneath an abstraction
iii) We use $\rightarrow^{L}$ to denote lazy reduction

For example, let us begin β-reducing $(\lambda x.\lambda y.xy)(\lambda z.z)$:

$$\begin{array}{rl} & (\lambda x.\lambda y.xy)(\lambda z.z) \\ \rightarrow_{\beta} & \lambda y.(\lambda z.z)y \end{array}$$

Whereas β-reduction would continue and reduce the inner redex $(\lambda z.z)y$, lazy reduction does not reduce beneath λ-abstractions. This term is in normal form for lazy reduction.

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.

Logic relates statements together with logical connectives. Under an interpretation, a statement is either true or false. Depending on the truth value of the component statements, a compound statement is either true or false. As far as we are concerned, the logical connectives consist of $\wedge, \vee, \rightarrow$, and $\neg$.

- $M \wedge N$ (read: 'and') is true only when both M and N are true.
- $M \vee N$ (read: 'or') is true only if either M or N are true.
- $M \rightarrow N$ (read: 'implies') is false when M is true but N is false.
- $\neg M$ (read: 'not') is true only when M is false.

The symbol $\rightarrow$ represents implication: whenever M is true, N is true.

2.3.1 Natural Deduction

Natural deduction is a system of logic. In natural deduction, there are *introduction* and *elimination* rules for each logical connective. These are presented as *inference rules.* Logical inference rules describe that if some statement A is true, then it follows that some other statement B is true. This is denoted by:

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

Using this notation, we can now describe the inference rules for logical implication, $\rightarrow$. For our purposes, this is the only logical connective we are interested in. The reason for this will become clear.

Definition 2.3.1 $\rightarrow$ INTRODUCTION AND ELIMINATION RULES

$$\frac{A \rightarrow B \quad A}{B} \rightarrow \mathcal{E} \qquad \frac{\begin{matrix}[A] \\ \vdots \\ B\end{matrix}}{A \rightarrow B} \rightarrow \mathcal{I}$$

The $\rightarrow \mathcal{E}$ rule says that assuming the statement $A \rightarrow B$ and the statement A are true, we can conclude that B is the case. The rule is read as an implication. For instance, consider

$$\begin{array}{lcl} A & = & \textit{It is raining} \\ B & = & \textit{It is wet outside} \\ A \rightarrow B & = & \textit{If (it is raining) then (it is wet outside)} \end{array}$$

If we know that "if (it is raining) then (it is wet outside)" and we are told "it is raining", clearly 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 implies B. Let us assume (where $[A]$ denotes that we assume A is true):

$$[A] = \textit{Turing did not see Church's work}$$

From this assumption, it's clear that that

$$B = \textit{Turing could not have stolen Church's work}$$

The $\rightarrow \mathcal{I}$ rule lets us conclude from these statements that

$$A \rightarrow B = \textit{If Turing did not see Church's work, he could not have stolen it}$$

Note the removal of the assumption ([ ]) from A. By restricting ourselves to these rules, we are working within *Implicative Intuitionistic Logic* (IIL). IIL is a restricted form of logic. For example Pierce's law, which says $((P \rightarrow Q) \rightarrow P) \rightarrow P$, is not applicable to IIL. [2]

2.3.2 Sequent Calculus

Gentzen explored natural deduction through the sequent calculus [12]. We have followed Girard *et al.*'s observation that the syntax of the sequent calculus is overcomplicated for the purpose of natural deduction [7]. However it is useful for both classical natural deduction and type assignment systems (see Sections 2.3.3 and 2.3.4 respectively).

Sequent calculus manipulates *sequents* where a sequent is denoted by:

$$\Gamma \vdash \Delta$$

On the left-hand side, the Γ represents a sequence of zero or more statements called the *antecedent.* On the right-hand side of the $\vdash$, Δ represents a different sequence of zero or more 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, if all the statements on the left are true then at least one of the statements on the right is true:

$$A_1, A_2, \ldots, A_n \vdash S_1, S_2, \ldots, S_m$$
denotes
$$A_1 \wedge A_2 \wedge \cdots \wedge A_n \rightarrow S_1 \vee S_2 \vee \cdots \vee S_m$$

[2]IIL also rejects the law of excluded middle which says for every P, $P \vee \neg P$. According to intuitionistic logic, unless we have a *constructive* proof for which of P or $\neg P$ is true, the statement is false

2.3.3 Classical Natural Deduction

Gentzen's classical natural deduction used the sequent calculus and introduced a set structural rules for manipulating sequents [3].

Definition 2.3.2 Structural rules of classical natural deduction

$$\frac{\Gamma, C, D, \Gamma' \vdash \Delta}{\Gamma, D, C, \Gamma' \vdash \Delta}\,\mathcal{L}X \qquad \frac{\Gamma \vdash \Delta, C, D, \Delta'}{\Gamma \vdash \Delta, D, C, \Delta'}\,\mathcal{R}X$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, C \vdash \Delta}\,\mathcal{L}W \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, C}\,\mathcal{R}W$$

$$\frac{\Gamma, C, C \vdash \Delta}{\Gamma, C \vdash \Delta}\,\mathcal{L}C \qquad \frac{\Gamma \vdash C, C, \Delta}{\Gamma \vdash C, \Delta}\,\mathcal{R}C$$

$$\frac{\Gamma \vdash \Delta, A \quad A, \Gamma' \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathbf{Cut}$$

The first set of rules, $\mathcal{LX}$ and $\mathcal{RX}$, are left and right exchange rules. They express the commutativity of statements in the left and right sequences. The weakening rules, $\mathcal{LW}$ and $\mathcal{RW}$, allow the introduction of new formulæ on either the left or right sequences. The contraction rules, $\mathcal{LC}$ and $\mathcal{RC}$, allow the contraction of multiple occurrences of a formula into a single occurrence. The cut rule allows us to replace assumptions of formulæ with their concrete proofs, if we have proved them somewhere else.

In addition to these, the system has a set of logical rules which relate closely to natural deduction. Although rules for all the logical connectives exist, we again choose to highlight just the cases important to this project:

Definition 2.3.3 Logical rules of classical natural deduction [3]

$$\frac{\Gamma \vdash C, \Delta \quad \Gamma', D \vdash \Delta'}{\Gamma, \Gamma', C \to D \vdash \Delta, \Delta'}\,\mathcal{L}\to \qquad \frac{\Gamma, C \vdash D, \Delta}{\Gamma \vdash C \to D, \Delta}\,\mathcal{R}\to$$

$$\frac{\Gamma \vdash C, \Delta}{\Gamma, \neg C \vdash \Delta}\,\mathcal{L}\neg \qquad \frac{\Gamma, C \vdash \Delta}{\Gamma \vdash \neg C, \Delta}\,\mathcal{R}\neg$$

Moving a statement from the left-hand sequent to the right negates that statement. The same is true moving a statement from the right-hand sequent to the left. The implication rules allow the introduction of $\to$ in either the left or right sequent.

2.3.4 Type Assignment

There are many type assignment different systems. Systems of type assignment introduce additional grammar and restrictions on the reduction rules of a formal 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 $\to$. This is described by the following BNF grammar:

Definition 2.3.4 (Grammar for constructing types)
Type variables are represented by the lower-case greek alphabet $\alpha, \beta, \gamma, ...$

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

2.3.5 Typed λ-Calculus

The typed λ-calculus is an extension of the λ-calculus with types assigned to λ-terms. λ-abstractions have arrow types: $A \to B$. This describes that the abstraction can only be applied to a value of type A and returns one of type B. Type assignment rules for the *typed* λ-calculus are:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \qquad N : A}{\Gamma \vdash MN : B} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

Similarly to the sequent calculus, the letter Γ represents a sequence of type assignment statements. In Γ, each variable can only have one type. The form of these rules denote that the typing judgement on the right-hand side of the $\vdash$ are a consequence of the sequence of type assignment statements in Γ. 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 second 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. These restrictions prevent the formation of terms with undesirable properties.

Example 2.3.5 (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 according to this simple type assignment system: the conflicting types for x mean the term is disallowed by the rules of the typed λ-calculus.

2.3.6 Curry-Howard Isomophism

The Curry-Howard isomorphism states that there is a true isomorphism between the type of a term and a logical proposition. The type of a term is a logical proposition and the term itself its a proof of that proposition. The simplification of a proof maps to the evaluation of the corresponding program [12].

Looking again at the type assignment rules of the typed λ-calculus and the inference rules of IIL, the correspondence is clear:

	λ-calculus	**IIL**
$\to\mathcal{E}$	$\dfrac{\Gamma \vdash M : A \to B \qquad N : A}{\Gamma \vdash MN : B}$	$\dfrac{A \to B \quad A}{B}$
$\to\mathcal{I}$	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$	$\dfrac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \to B}$

The correspondence between logic and programs is not limited to IIL and the typed λ-calculus. There are many features of computer programs that have counter-parts in logical systems.

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 and this data constructor can only have a single argument 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 place of a data constructor, in this case `String`. 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 typed 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 _ _     = ThreeOrMore
```

To declare an instance of a type class, we must supply the bodies for functions in the class specification. The `Addable` class, for instance, requires that the body of the `add` function is defined. When declaring an instance, we have to ensure the type specification of each function is respected.

2.5 Continuations

As in the example in Figure 2.1.4, compound λ-terms can be evaluated into a dominant term and a context:

Assume that $M \to_\beta^* M'$

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

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

Contexts are partial terms that remain to be reduced after the reduction of the dominant term. After the dominant term has reduced to a value, that value will fill the hole ($\Box$) of the context to form a new term. For example, consider the term

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

According to our bracketing convention, the leftmost redex is reduced first:

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

After this term has reduced, the reduct will be applied to t:

$$\Box t$$

In this way, the context $\Box t$ holds information about the *rest* of the reduction steps. The context of a term is the future computation of that term: what will happen to the term after it has been reduced. For this reason, the 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 continuations into one big continuation we only have two components at any point during the reduction: the current dominant term and the *current continuation.*

By adding additional operators to a language, the continuations of terms can be exposed. This provides programmers with the ability to control continuations. Control operators that only allow manipulation of the entire remaining continuation are **undelimited continuations**. For example, Scheme's `call/cc` operator aborts the entire remaining continuation.

2.5.2 Delimited Continuations

Instead, if we maintain continuations in a stack when decomposing complex terms, we can keep the 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 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.

If the control operators of a language allow manipulation of portions of the continuation stack, the continuations are **delimited**. To manipulate portions of the stack, these operators need a *control delimiter.* A control delimiter marks a point on the stack that delimits how far the operators can control. These delimiters commonly called *prompts*, after Felleisen first introduced them as such [6]. By allowing prompts to be pushed onto the stack, we can recall portions of the stack *up until* a prompt. This gives the operators a finer grain of control over continuations.

2.5.3 Continuation-Passing Style

By rewriting λ-terms, a term's continuation 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 [9].

Definition 2.5.1 (Translation of standard λ-terms into CPS) [3]

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

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.

Example 2.5.2 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 $\lambda m.N$:

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

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 the CPS-term:

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

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 `>>=` (read: 'bind') that binds the first suspended computation to a continuation which returns another suspended computation:

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

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 defined the $\lambda\mu$-calculus as a system with an isomorphism to classical natural deduction. 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, μ introduces μ-abstractions. The body of a μ-abstraction must be a named term. A named term consists of a name of the form $[\alpha]$ followed by an unnamed term.

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

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

(Unnamed term)	M, N	$::=$	$x \mid \lambda x.M \mid M\ N \mid \mu\alpha.C$
(Named term)	C	$::=$	$[\alpha]M$

2.6.2 Reduction Rules

Definition 2.6.2 (Reduction rules for $\lambda\mu$-calculus)
i) $\rightarrow_{\beta\mu}$ is defined as the compatible closure of rules:

$$\begin{array}{rrcll}
\text{(logical)} & (\lambda x.M)N & \rightarrow_\beta & M[N/x] & \\
\text{(structural)} & (\mu\alpha.[\beta]M)N & \rightarrow\mu & \mu\alpha.[\beta](M\{[\gamma]M'N/[\alpha]M'\}) & (\beta \neq \alpha) \\
 & (\mu\alpha.[\beta]M)N & \rightarrow\mu & \mu\alpha.[\gamma](M\{[\gamma]M'N/[\alpha]M'\})N & (\beta = \alpha) \\
(\mu\eta) & \mu\alpha.[\alpha]M & \rightarrow\mu & M & (\alpha \notin fn(M)) \\
\text{(renaming)} & \mu\delta.[\beta](\mu\gamma.[\alpha]M) & \rightarrow\mu & \mu\delta.([\alpha]M)[\beta/\gamma] &
\end{array}$$

ii) $\rightarrow^*_{\beta\mu}$ is defined as the transitive closure of $\rightarrow_{\beta\mu}$.

The structural reduction rule simply states that the application of a μ-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 [8]. If we knew how many variables the term is applied to, we could replace

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

with

$$\lambda x_1\ldots\lambda x_n\ldots Mx_1\ldots x_n$$

where n is the number of variables the term is applied to.

2.6.3 Computational Significance

A $\lambda\mu$-abstraction applied to some term N points the μ-variable to the context $\Box N$. In this sense, the $\lambda\mu$-abstraction captures contexts where the contexts are supplied by application. For this reason, $\lambda\mu$-calculus is described as using *applicative* 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.

By translating μ-terms into CPS, we can see the mapping between μ-variables and contexts more clearly:

Definition 2.6.3 CPS translation of μ-terms [3]

$$\begin{array}{rcl}
[\![\mu\alpha.M]\!] & \triangleq & \lambda\alpha.[\![M]\!] \\
[\![[\alpha]M]\!] & \triangleq & \lambda k.[\![M]\!]\ \alpha\ k
\end{array}$$

A μ-abstraction binds a continuation to a variable for use throughout the abstraction's body. A named term applies the term to the name, effectively running the term in the continuation.

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 $\Box N$. The dominant term is now a μ-abstraction so we can think about the variable at the head of the μ-abstraction now mapping to the context $\{\alpha \Rightarrow \Box N\}$:

Example 2.6.4

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 $\Box N$ after which the context will be destroyed. For example, let us replace M with $\mu\circ.[\alpha](\lambda s.fs)$: [3]

Example 2.6.5

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 μ-abstraction now points to a new context. This means that μ-abstractions will pass all of the applicative contexts to the named subterms:

Example 2.6.6

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 Curry-Howard Isomorphism

The typed variant of the $\lambda\mu$-calculus is isomorphic to classical natural deduction. The type assignment rules for the typed $\lambda\mu$-calculus are as follows:

Definition 2.6.7 Typing rules for the typed $\lambda\mu$-calculus
The three type-assignment rules from the λ-calculus are still valid. In addition:

$$\frac{\Gamma \vdash M : B \mid \alpha : A, \beta : B, \Delta}{\Gamma \vdash \mu\alpha.[\beta]M : A \mid \beta : B, \Delta} \qquad \frac{\Gamma \vdash M : A \mid \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.[\alpha]M : A \mid \Delta}$$

These rules correspond to the structural rules of classical natural deduction. The original presentation of Parigot's type assignment makes the isomorphism clearer. Our presentation here makes it easier to understand the role of the operators.

2.6.5 Lazy Reduction

We define lazy-reduction for the $\lambda\mu$-calculus similarly to λ-calculus. The outermost, leftmost reduction takes place first. If no reductions can take place on the outermost level, then the term is in lazy normal form. For instance, consider the μ-term:

$$\mu\alpha.[\beta]((\lambda x.x)z)$$

This term is in lazy normal form: the only redex, $(\lambda x.x)z$, is beneath a μ-abstraction. The term cannot be reduced any further.

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

2.7 Calculus of Delimited Continuations

Simon Peyton-Jones *et al.* extended the λ-calculus with additional operators in order create a framework for implementing delimited continuations [4]. This calculus will be referred to as the calculus of delimited-continuations or *CDC*. Many calculi have been devised with control mechanisms for manipulating continuations, like the $\lambda\mu$-calculus. These control mechanisms manipulate either delimited or undelimited continuations. CDC provides a set of operators that are capable of expressing many of the common control mechanisms found in the literature.

2.7.1 Syntax

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

Definition 2.7.1 (Grammar for CDC)

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

2.7.2 Reduction Rules

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

- e - the current dominant term
- D - the current context/continuation
- E - the stack of remaining continuations
- q - a global counter for producing fresh prompt values

By representing terms in this way, the reduction rules are able to make the control of terms and their continuations more explicit.

The abstract machine also introduces some Haskell-style notation for dealing with sequences. An empty sequence is represented by $[]$. A value added to the head of a list is represented by the symbol $:$, for instance $D : []$. $+\!\!+$ represents two lists appended together. $E^p_\uparrow$ and $E^p_\downarrow$ denote the subsequence of E *until* prompt p and *from* prompt p, respectively; neither of these subsequences contain p.

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.

Definition 2.7.2 (Operational semantics for CDC) [4]

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

- *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

λ^{try} is a system defined by van Bakel in [10] for modelling exceptions. It is an syntactic and semantic extension of the λ-calculus. Unlike previous systems, the λ^{try}-calculus introduces the idea of 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 for which exception. 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 a series of one or more catch statements. For convenience, we will use the notation

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

to describe a catch block with one or more catch statements.

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)[10]

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

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 — CDC Interpreter

This chapter explores the implementation of an interpreter for CDC. It compares portions of source code with reduction cases from the abstract machine. Finally, it provides an example of the interpreter's output and the transliteration to a true CDC derivation.

3.1 Implementation

Although Peyton-Jones *et al.* implement a language-level module for CDC [4], 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 $\lambda\mu$ in CDC. Examining transformation steps in full also allows us to derive proofs of soundness and completeness for these translations. For this reason, the interpreter was implemented as a term-rewriting program.

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

3.1.1 Data structures

There are two data types for representing CDC terms, `Value` and `Expr` (Figure 3.1.1). Values are not evaluated: when a term has been reduced to a value, it has terminated on that value. An expression (`Expr`) is a term that can be evaluated to another term. The only exception is a `Hole` which can take any position an expression can. For this reason, it must be a data constructor for expression types.

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

3.1.2 Utility Functions

We define some utility functions to simplify the implementation. Informally, these functions behave as follows (see Figure 3.1.2 for implementations details):

- `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.

```haskell
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)
```

Figure 3.1: Data structures for the CDC interpreter

- `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 $M[v/x]$.

3.1.3 Reduction Rules

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

Application

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

The `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,

```haskell
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 3.2: Utility functions for CDC interpreter

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.

```
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
```

PushPrompt

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

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

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

WithSubCont

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

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

```
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
```

PushSubCont

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

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

```
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
```

Substitution

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

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

NewPrompt

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

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

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

3.1.4 Example

The terms *pushPrompt*, *newPrompt*, *withSubCont*, and *pushSubCont* are too long and clutter lengthy derivations. We define the following aliases for convenience:

Definition 3.1.1 CDC ALIASES FOR OPERATORS

$$\begin{array}{rcl} \text{NP} & = & newPrompt \\ \text{PP} & = & pushPrompt \\ \text{WSC} & = & withSubCont \\ \text{PSC} & = & pushSubCont \end{array}$$

Additionally, we have defined utility function aliases for our data constructors

```
np = NewPrompt
pp = PushPrompt
wsc = WithSubCont
psc = PushSubCont
var = Val . Var
abst c = Val . Abs c
```

We define an evaluation engine which recursively evaluates the term until the evaluation returns the same term. At this point it terminates. The evaluation engine takes a function of type `State -> String` to specify the format of the printed term.

```
evalFull :: (State -> String) -> State -> IO ()
evalFull formatFn st = do
  putStrLn $ formatFn st
  let st' = eval st in
    if st == st' then
      putStrLn "QED"
    else
      evalFull formatFn st'
```

We can translate the CDC term

$$(\lambda p.\textsc{pp}\ p\ (\textsc{wsc}\ p\ (\lambda\alpha.\textsc{psc}\ \alpha\ s))\ t)\ \textsc{np}$$

into the data structures defined by our interpreter:

```
term = App
  (abst 'p'
    (pp (var 'p') (App
      (wsc (var 'p') (abst 'a'
        (psc (var 'a') (var 's'))))
      (var 't'))))
  np
```

Passing this state into our evaluator with `prettifyState`, we get the following:

```
> evalFull prettifyState term
(((\p.(pp p ((wsc p (\a.(psc a s)))t))) np), _, [], 0)
( np, ((\p.(pp p ((wsc p (\a.(psc a s)))t)))_), [], 0)
(0, ((\p.(pp p ((wsc p (\a.(psc a s)))t)))_), [], 1)
(((\p.(pp p ((wsc p (\a.(psc a s)))t)))0), _, [], 1)
((pp p ((wsc p (\a.(psc a s)))t))[0/p], _, [], 1)
((pp 0 ((wsc 0 (\a.(psc a s)))t)), _, [], 1)
(((wsc 0 (\a.(psc a s)))t), _, [0:_], 1)
((wsc 0 (\a.(psc a s))), (_t), [0:_], 1)
(((\a.(psc a s))(_t)), _, [_], 1)
((psc a s)[(_t)/a], _, [_], 1)
((psc (_t) s), _, [_], 1)
(s, _, [(_t):_:_], 1)
(s, (_t), [_:_], 1)
((st), _, [_:_], 1)
QED
```

From this, which we can translate to the following derivation:

$$\begin{array}{lcrrr}
 & (\lambda p.\textsc{pp}\ p\ (\textsc{wsc}\ p\ (\lambda\alpha.\textsc{psc}\ \alpha\ s))\ t)\ \textsc{np} & \Box & [] & 0 \\
\rightarrow_{CDC} & \textsc{np} & (\lambda p\ldots t)\ \Box & [] & 0 \\
\rightarrow_{CDC} & 0 & (\lambda p\ldots t)\ \Box & [] & 1 \\
\rightarrow_{CDC} & (\lambda p.\textsc{pp}\ p\ (\textsc{wsc}\ p\ (\lambda\alpha.\textsc{psc}\ \alpha\ s))\ t)\ 0 & \Box & [] & 1 \\
\rightarrow_{\beta} & (\textsc{pp}\ p\ (\textsc{wsc}\ p\ (\lambda\alpha.\textsc{psc}\ \alpha\ s))\ t))[p/0] & \Box & [] & 1 \\
\rightarrow_{\beta} & \textsc{pp}\ 0\ (\textsc{wsc}\ 0\ (\lambda\alpha.\textsc{psc}\ \alpha\ s))\ t) & \Box & [] & 1 \\
\rightarrow_{CDC} & (\textsc{wsc}\ 0\ (\lambda\alpha.\textsc{psc}\ \alpha\ s))\ t & \Box & 0:\Box & 1 \\
\rightarrow_{CDC} & \textsc{wsc}\ 0\ (\lambda\alpha.\textsc{psc}\ \alpha\ s)) & \Box\ t & 0:\Box & 1 \\
\rightarrow_{CDC} & (\lambda\alpha.\textsc{psc}\ \alpha\ s))(\Box t) & \Box & \Box & 1 \\
\rightarrow_{\beta} & (\textsc{psc}\ \alpha\ s)[\Box t/\alpha] & \Box & \Box & 1 \\
\rightarrow_{\beta} & \textsc{psc}\ \Box t\ s & \Box & \Box & 1 \\
\rightarrow_{CDC} & s & \Box & \Box t:\Box:\Box & 1 \\
\rightarrow_{CDC} & s & \Box\ t & \Box:\Box & 1 \\
\rightarrow_{CDC} & st & \Box & \Box:\Box & 1
\end{array}$$

Using the output of this interpreter, then, we can generate derivations to help us verify the translation from $\lambda\mu$to CDC.

4 — Translations

In this chapter, we develop a interpretation of $\lambda\mu$ in CDC. 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 CDC. This will then be used as a basis for the implementation of λ^{try} in Haskell.*

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

van Bakel describes the interpretation of λ^{try} to $\lambda\mu$ [10]:

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

throw n(M) terms are modelled using μ-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 μ-abstraction calls c_n. This λ-variable is bound by the translation of **try** terms. This binding means that the exception handlers, represented by $\lambda x.[\![L]\!]$, are in scope for the reduction of the body of the try M.

The translation defined by van Bakel in [10] are from λ^{try} into $\lambda\mu$ restricted to *lazy* reduction. Without lazy reduction, any occurrence of a throw in the body of a try-block – even within an abstraction – would begin executing. Lazy reduction prevents this undesirable internal reduction.

4.2 Interpreting $\lambda\mu$ in CDC

The translation of $\lambda\mu$-terms into CDC 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 CDC is:

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

$$(\lambda\text{P}_0\ .\text{PP}\ \ \text{P}_0\ \ [\![M]\!])\ \text{NP}$$

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

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

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

To implement μ-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 will reduce M and passes the result to β.

Once translated, the CDC terms reduce only the outermost redex. This translation is of a lazy $\lambda\mu$-calculus to CDC.

4.2.1 Notation

To carry out proofs, the full state of the abstract machine is displayed:

$$\langle\ M,\quad D,\quad E\ \rangle$$

where each column corresponds to one component of the original abstract machine. The angle-brackets and commas will be omitted from CDC where it will not effect clarity. Our translations only use a single prompt so we omit the final column of the abstract machine (used for representing the global prompt counter).

When an empty context is in a sequence, it has no effect on the machine: a sequence $D : \Box : D'$ is extensionally equivalent to $D : D'$. For this reason, we omit empty contexts from sequences. For example in the case of the following reduction

$$\langle\ \textsc{psc}\ \beta\ M,\quad \Box,\quad \textsc{p}_0\ \ \rangle\ \ \rightarrow_{CDC}\ \ \langle\ M,\quad \Box,\quad \beta : \Box : \textsc{p}_0\ \ \rangle$$

we will instead write

$$\langle\ \textsc{psc}\ \beta\ M,\quad \Box,\quad \textsc{p}_0\ \ \rangle\ \ \rightarrow_{CDC}\ \ \langle\ M,\quad \Box,\quad \beta : \textsc{p}_0\ \ \rangle$$

4.2.2 Additional Translations

We alter μ-reduction to consume multiple variables. We use $\overline{N}$ to denote a series of one or more terms. The application of a μ-abstraction to multiple variables will consume them all at once:

Definition 4.2.3 μ-reduction to consume multiple variables

$$(\mu\alpha.[\beta]M)\overline{N}\quad \rightarrow_\mu\quad \mu\alpha.([\beta]M\{[\alpha]M'\overline{N}/[\alpha]M'\})$$

This does not change the behaviour of μ-reduction but condenses the reduction steps. The entire applicative context is consumed. Therefore the remaining μ-abstraction will point α to $\Box$. This means that all labelled subterms $[\alpha]M'$ will be translated to $\langle$ $\textsc{psc}$ $\Box$ $[\![M']\!]$ $\rangle$. CDC reduces $\langle$ $\textsc{psc}$ $\Box$ $[\![M']\!]$ $\rangle$ to $[\![M']\!]$. This reduction means we can discard the α labels after consuming the entire context. Given this, we can define the following translation for multiple-variable consumption:

Definition 4.2.4 Translation of multiple variable consumption to CDC

$$[\![M[[\alpha]M'\overline{N}/[\alpha]M']]\!] \;\triangleq\; [\![M]\!][\Box[\![\overline{N}]\!]/\alpha]$$

Example 4.2.5 Example of multiple variable consumption in CDC and $\lambda\mu$

$$\begin{array}{ll} & (\mu\alpha.[\alpha](\lambda x.\lambda y.xy))st \\ \to_\mu & \mu\alpha.([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\}) \end{array}$$

From here we can continue with the $\beta\mu$-reduction:

$$\begin{array}{lcr} & \mu\alpha.([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\}) & \\ \to_\mu & \mu\alpha.[\alpha](\lambda x.\lambda y.xy)st & \\ \to_\beta & \mu\alpha.[\alpha](\lambda y.sy)t & \\ \to_\beta & \mu\alpha.[\alpha]st & \\ \to_\mu & st & (\alpha \notin fn(st)) \end{array}$$

We can also continue by translating to and reducing in CDC:

$$\begin{array}{lccr} & [\![\mu\alpha.([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst)\}]\!] & & \\ \triangleq & \textsc{wsc}\ \textsc{p}_0\ (\lambda\alpha.\textsc{pp}\ \textsc{p}_0\ [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!]) & \Box & \textsc{p}_0 \\ \to & (\lambda\alpha.\textsc{pp}\ \textsc{p}_0\ [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!])(\Box) & \Box & [] \\ \to_\beta & (\textsc{pp}\ \textsc{p}_0\ [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!])[\Box/\alpha] & \Box & [] \\ \to & \textsc{pp}\ \textsc{p}_0\ [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!][\Box/\alpha] & \Box & [] \\ \to & [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!][\Box/\alpha] & \Box & \textsc{p}_0 \\ \to & ((\textsc{psc}\ \alpha(\lambda x.\lambda y.xy))[\Box st/\alpha])[\Box/\alpha] & \Box & \textsc{p}_0 \\ \to & (\textsc{psc}\ \Box st\ (\lambda x.\lambda y.xy))[\Box/\alpha] & \Box & \textsc{p}_0 \\ \to & \textsc{psc}\ \Box st\ (\lambda x.\lambda y.xy) & \Box & \textsc{p}_0 \\ \to & (\lambda x.\lambda y.xy) & \Box & \Box st : \textsc{p}_0 \\ \to & (\lambda x.\lambda y.xy) & \Box st & \textsc{p}_0 \\ \to & (\lambda x.\lambda y.xy)st & \Box & \textsc{p}_0 \\ \to_\beta & (\lambda y.sy)t & \Box & \textsc{p}_0 \\ \to_\beta & st & \Box & \textsc{p}_0 \end{array}$$

Using the translation in Definition 4.2.4, the proof that $[\![\cdot]\!]$ respects $\to_\mu$ follows easily.

4.2.3 Properties

Theorem 4.2.6 (Soundness of $[\![\cdot]\!]$)

$$M \to_\mu^L N \Rightarrow \exists P.\ [\![M]\!] \to^* P \wedge [\![N]\!] \to^* P$$

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

$$\mu\alpha.[\alpha]M \to M : \qquad\qquad (\alpha \notin fn(M))$$

$$\begin{array}{lccr} & [\![\mu\alpha.[\alpha]M]\!] & & \\ \triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \alpha[\![M]\!]) & D & \textsc{p}_0 \\ \to_{CDC} & (\lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \alpha[\![M]\!]))(D) & \Box & [] \\ \to_{CDC} & (\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \alpha\ [\![M]\!]))[D/\alpha] & \Box & [] \\ \to_{CDC} & \textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \alpha\ [\![M]\!])[D/\alpha] & \Box & [] \\ \to_{CDC} & (\textsc{psc}\ \alpha\ [\![M]\!])[D/\alpha] & \Box & \textsc{p}_0 \\ \to_{CDC} & \textsc{psc}\ D\ [\![M]\!][D/\alpha] & \Box & \textsc{p}_0 \\ \to_{CDC} & [\![M]\!][\Box/\alpha] & \Box & D : \textsc{p}_0 \\ \to_{CDC} & [\![M]\!] & \Box & D : \textsc{p}_0 \\ \to_{CDC} & [\![M]\!] & D & \textsc{p}_0 \\ \triangleq & M & D & \textsc{p}_0 \end{array}$$

$$(\mu\alpha.[\alpha]M)\overline{N} \to \mu\alpha.[\alpha]M\{[\alpha]M'\overline{N}/[\alpha]M'\}\overline{N} : \qquad (\alpha \notin fn(\overline{N}))$$

$$\begin{array}{llcr}
 & \llbracket(\mu\alpha.[\alpha]M)\overline{N}\rrbracket & & \\
\triangleq & \llbracket(\mu\alpha.[\alpha]M)\rrbracket\llbracket\overline{N}\rrbracket & & \\
\triangleq & (\textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M\rrbracket))\llbracket\overline{N}\rrbracket & \Box & \mathrm{P}_0 \\
\to_{CDC} & \textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M\rrbracket) & \Box\llbracket\overline{N}\rrbracket & \mathrm{P}_0 \\
\to_{CDC} & \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M\rrbracket)(\Box\llbracket\overline{N}\rrbracket) & \Box & [] \\
\to_{CDC} & (\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M\rrbracket))[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & [] \\
\to_{CDC} & \textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M\rrbracket)[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & [] \\
\to_{CDC} & \textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \Box\llbracket\overline{N}\rrbracket\ \llbracket M\rrbracket[\Box\llbracket\overline{N}\rrbracket/\alpha]) & \Box & [] \\
\to_{CDC} & \textsc{psc}\ \Box\llbracket\overline{N}\rrbracket\ \llbracket M\rrbracket[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & \mathrm{P}_0 \\
\to_{CDC} & \llbracket M\rrbracket[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & \Box\llbracket\overline{N}\rrbracket : \mathrm{P}_0 \\
\to_{CDC} & \llbracket M\rrbracket[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box\llbracket\overline{N}\rrbracket & \mathrm{P}_0
\end{array}$$

This final state is P. Now we must prove $\llbracket N\rrbracket \to^* P$:

$$\begin{array}{llcr}
 & \llbracket\mu\alpha.[\alpha]M\{[\alpha]M'\overline{N}/[\alpha]M'\}\overline{N}\rrbracket & & \\
\triangleq & \textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\overline{N}\rrbracket) & \Box & \mathrm{P}_0 \\
\to_{CDC} & \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\overline{N}\rrbracket)(\Box) & \Box & [] \\
\to_{CDC} & (\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\overline{N}\rrbracket))[\Box/\alpha] & \Box & [] \\
\to_{CDC} & \textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \alpha\ \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\overline{N}\rrbracket)[\Box/\alpha] & \Box & [] \\
\to_{CDC} & (\textsc{psc}\ \alpha\ \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\overline{N}\rrbracket)[\Box/\alpha] & \Box & \mathrm{P}_0 \\
\to_{CDC} & \textsc{psc}\ \Box\ \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\overline{N}\rrbracket[\Box/\alpha] & \Box & \mathrm{P}_0 \\
\to_{CDC} & \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\overline{N}\rrbracket[\Box/\alpha] & \Box & \mathrm{P}_0 \\
\to_{CDC} & \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\rrbracket[\Box/\alpha]\ \llbracket\overline{N}\rrbracket & \Box & \mathrm{P}_0 \\
\to_{CDC} & \llbracket M[[\alpha]M'\overline{N}/[\alpha]M']\rrbracket[\Box/\alpha] & \Box\llbracket\overline{N}\rrbracket & \mathrm{P}_0 \\
\to_{CDC} & \llbracket M\rrbracket[\Box\overline{N}/\alpha][\Box/\alpha] & \Box\llbracket\overline{N}\rrbracket & \mathrm{P}_0 \\
\to_{CDC} & \llbracket M\rrbracket[\Box\overline{N}/\alpha] & \Box\llbracket\overline{N}\rrbracket & \mathrm{P}_0
\end{array}$$

In this final reduction step, α can not occur in $\llbracket M\rrbracket[\Box\overline{N}/\alpha]$ because all occurrences have been substituted for $\Box\llbracket\overline{N}\rrbracket$.

$$(\mu\alpha.[\beta]M)\overline{N} \to \mu\alpha.[\beta](M\{[\alpha]M'N/[\alpha]M'\}) : \qquad (\alpha \neq \beta)$$

$$\begin{array}{llcr}
 & \llbracket(\mu\alpha.[\beta]M)\overline{N}\rrbracket & & \\
\triangleq & \llbracket(\mu\alpha.[\beta]M)\rrbracket\llbracket\overline{N}\rrbracket & & \\
\triangleq & (\textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket))\llbracket\overline{N}\rrbracket & \Box & \mathrm{P}_0 \\
\to_{CDC} & (\textsc{wsc}\ \mathrm{P}_0\ \lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket)) & \Box\llbracket\overline{N}\rrbracket & \mathrm{P}_0 \\
\to_{CDC} & (\lambda\alpha.\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket))(\Box\llbracket\overline{N}\rrbracket) & \Box & [] \\
\to_{CDC} & (\textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket))[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & [] \\
\to_{CDC} & \textsc{pp}\ \mathrm{P}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket)[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & [] \\
\to_{CDC} & (\textsc{psc}\ \beta\ \llbracket M\rrbracket)[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & \mathrm{P}_0 \\
\to_{CDC} & \textsc{psc}\ \beta\ \llbracket M\rrbracket[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & \mathrm{P}_0 \\
\to_{CDC} & \llbracket M\rrbracket[\Box\llbracket\overline{N}\rrbracket/\alpha] & \Box & \beta : \mathrm{P}_0
\end{array}$$

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

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

$$
\begin{array}{lccr}
 & [\![\mu\alpha.[\beta]\mu\gamma.[\delta]M]\!] & & \\
\triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ [\![\mu\gamma.[\delta]M]\!]) & \Box & \textsc{p}_0 \\
\to_{CDC} & (\lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ [\![\mu\gamma.[\delta]M]\!]))(\Box) & \Box & [] \\
\to_{CDC} & (\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ [\![\mu\gamma.[\delta]M]\!]))[\Box/\alpha] & \Box & [] \\
\to_{CDC} & \textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ [\![\mu\gamma.[\delta]M]\!])[\Box/\alpha] & \Box & [] \\
\to_{CDC} & (\textsc{psc}\ \beta\ [\![\mu\gamma.[\delta]M]\!])[\Box/\alpha] & \Box & \textsc{p}_0 \\
\to_{CDC} & \textsc{psc}\ \beta\ [\![\mu\gamma.[\delta]M]\!][\Box/\alpha] & \Box & \textsc{p}_0 \\
\to_{CDC} & [\![\mu\gamma.[\delta]M]\!][\Box/\alpha] & \Box & \beta : \textsc{p}_0 \\
\triangleq & (\textsc{wsc}\ \textsc{p}_0\ \lambda\gamma.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta[\![M]\!]))[\Box/\alpha] & \Box & \beta : \textsc{p}_0 \\
\to_{CDC} & \textsc{wsc}\ \textsc{p}_0\ \lambda\gamma.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta[\![M]\!])[\Box/\alpha] & \Box & \beta : \textsc{p}_0 \\
\to_{CDC} & (\lambda\gamma.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta[\![M]\!])[\Box/\alpha])(\beta) & \Box & [] \\
\to_{CDC} & (\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta[\![M]\!])[\Box/\alpha])[\beta/\gamma] & \Box & [] \\
\to_{CDC} & \textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta[\![M]\!])[\Box/\alpha][\beta/\gamma] & \Box & [] \\
\to_{CDC} & (\textsc{psc}\ \delta[\![M]\!])[\Box/\alpha][\beta/\gamma] & \Box & \textsc{p}_0 \\
\to_{CDC} & \textsc{psc}\ \delta[\![M]\!][\Box/\alpha][\beta/\gamma] & \Box & \textsc{p}_0 \\
\to_{CDC} & [\![M]\!][\Box/\alpha][\beta/\gamma] & \Box & \delta : \textsc{p}_0
\end{array}
$$

$$
\begin{array}{lccr}
 & [\![\mu\alpha[\delta]M[\beta/\gamma]]\!] & & \\
\triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta\ [\![M]\!][\beta/\gamma]) & \Box & \textsc{p}_0 \\
\to_{CDC} & (\lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta\ [\![M]\!][\beta/\gamma]))(\Box) & \Box & [] \\
\to_{CDC} & (\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta\ [\![M]\!][\beta/\gamma]))[\Box/\alpha] & \Box & [] \\
\to_{CDC} & \textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \delta\ [\![M]\!][\beta/\gamma])[\Box/\alpha] & \Box & [] \\
\to_{CDC} & (\textsc{psc}\ \delta\ [\![M]\!][\beta/\gamma])[\Box/\alpha] & \Box & \textsc{p}_0 \\
\to_{CDC} & \textsc{psc}\ \delta\ [\![M]\!][\beta/\gamma][\Box/\alpha] & \Box & \textsc{p}_0 \\
\to_{CDC} & [\![M]\!][\beta/\gamma][\Box/\alpha] & \Box & \delta : \textsc{p}_0
\end{array}
$$

Remark 4.2.7 The terms $[\![M]\!][\beta/\gamma][\Box/\alpha]$ and $[\![M]\!][\Box/\alpha][\beta/\gamma]$ are identical if $\alpha \neq \beta \neq \gamma$.

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

$$\begin{array}{lccr}
 & \llbracket\mu\alpha.[\beta]\mu\gamma.[\gamma]M\rrbracket & & \\
\triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ \llbracket\mu\gamma.[\gamma]M\rrbracket) & \square & \textsc{p}_0 \\
\to_{CDC} & (\lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ \llbracket\mu\gamma.[\gamma]M\rrbracket))(\square) & \square & [] \\
\to_{CDC} & (\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ \llbracket\mu\gamma.[\gamma]M\rrbracket))[\square/\alpha] & \square & [] \\
\to_{CDC} & \textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ \llbracket\mu\gamma.[\gamma]M\rrbracket)[\square/\alpha] & \square & [] \\
\to_{CDC} & (\textsc{psc}\ \beta\ \llbracket\mu\gamma.[\gamma]M\rrbracket)[\square/\alpha] & \square & \textsc{p}_0 \\
\to_{CDC} & \textsc{psc}\ \beta\ \llbracket\mu\gamma.[\gamma]M\rrbracket[\square/\alpha] & \square & \textsc{p}_0 \\
\to_{CDC} & \llbracket\mu\gamma.[\gamma]M\rrbracket[\square/\alpha] & \square & \beta : \textsc{p}_0 \\
\triangleq & (\textsc{wsc}\ \textsc{p}_0\ \lambda\gamma.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \gamma\llbracket M\rrbracket))[\square/\alpha] & \square & \beta : \textsc{p}_0 \\
\to_{CDC} & \textsc{wsc}\ \textsc{p}_0\ \lambda\gamma.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \gamma\llbracket M\rrbracket)[\square/\alpha] & \square & \beta : \textsc{p}_0 \\
\to_{CDC} & (\lambda\gamma.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \gamma\llbracket M\rrbracket)[\square/\alpha])(\beta) & \square & [] \\
\to_{CDC} & (\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \gamma\llbracket M\rrbracket)[\square/\alpha])[\beta/\gamma] & \square & [] \\
\to_{CDC} & \textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \gamma\llbracket M\rrbracket)[\square/\alpha][\beta/\gamma] & \square & [] \\
\to_{CDC} & (\textsc{psc}\ \gamma\llbracket M\rrbracket)[\square/\alpha][\beta/\gamma] & \square & \textsc{p}_0 \\
\to_{CDC} & \textsc{psc}\ \beta\llbracket M\rrbracket[\square/\alpha][\beta/\gamma] & \square & \textsc{p}_0 \\
\to_{CDC} & \llbracket M\rrbracket[\square/\alpha][\beta/\gamma] & \square & \beta : \textsc{p}_0
\end{array}$$

$$\begin{array}{lccr}
 & \llbracket\mu\alpha[\beta]M[\beta/\gamma]\rrbracket & & \\
\triangleq & \textsc{wsc}\ \textsc{p}_0\ \lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket[\beta/\gamma]) & \square & \textsc{p}_0 \\
\to_{CDC} & (\lambda\alpha.\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket[\beta/\gamma]))(\square) & \square & [] \\
\to_{CDC} & (\textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket[\beta/\gamma]))[\square/\alpha] & \square & [] \\
\to_{CDC} & \textsc{pp}\ \textsc{p}_0\ (\textsc{psc}\ \beta\ \llbracket M\rrbracket[\beta/\gamma])[\square/\alpha] & \square & [] \\
\to_{CDC} & (\textsc{psc}\ \beta\ \llbracket M\rrbracket[\beta/\gamma])[\square/\alpha] & \square & \textsc{p}_0 \\
\to_{CDC} & \textsc{psc}\ \beta\ \llbracket M\rrbracket[\beta/\gamma][\square/\alpha] & \square & \textsc{p}_0 \\
\to_{CDC} & \llbracket M\rrbracket[\beta/\gamma][\square/\alpha] & \square & \beta : \textsc{p}_0
\end{array}$$

$\square$

Theorem 4.2.8 (Completeness of $\llbracket\cdot\rrbracket$)

$$\llbracket M\rrbracket \to^* Q \Rightarrow \exists N.M \to_\mu^{L*} N \wedge \llbracket N\rrbracket \to^* Q$$

Proof. By induction on the definition of $\llbracket M\rrbracket$.

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

$$\begin{array}{rcll}
\llbracket\mu\alpha.[\alpha]M\rrbracket & \to^* & \langle\ \llbracket M\rrbracket,\ \square,\ \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)} \\
\mu\alpha.[\alpha]M & \to_\mu & M & \text{(Def 4.2.2)} \\
\llbracket M\rrbracket & \to^* & \langle\ \llbracket M\rrbracket,\ \square,\ \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)}
\end{array}$$

$(\mu\alpha.[\alpha]M)\overline{N}$:

$$\begin{array}{rcll}
\llbracket(\mu\alpha.[\alpha]M)\overline{N}\rrbracket & \to^* & \langle\ \llbracket M\rrbracket[\square\llbracket\overline{N}\rrbracket/\alpha],\ \square\llbracket\overline{N}\rrbracket,\ \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)} \\
(\mu\alpha.[\alpha]M)\overline{N} & \to_\mu & \mu\alpha.[\alpha]M\{[\alpha]M'\overline{N}/[\alpha]M'\}\overline{N} & \text{(Def 4.2.2)} \\
\llbracket\mu\alpha.[\alpha]M\{[\alpha]M'\overline{N}/[\alpha]M'\}\overline{N}\rrbracket & \to^* & \langle\ \llbracket M\rrbracket[\square\llbracket\overline{N}\rrbracket/\alpha],\ \square\llbracket\overline{N}\rrbracket,\ \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)}
\end{array}$$

$(\mu\alpha.[\beta]M)\overline{N}$: $(\alpha \neq \beta)$

$$\begin{array}{rcll}
\llbracket(\mu\alpha.[\beta]M)\overline{N}\rrbracket & \to^* & \langle\llbracket M\rrbracket[\square\llbracket\overline{N}\rrbracket/\alpha],\ \square,\ \beta : \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)} \\
(\mu\alpha.[\beta]M)\overline{N} & \to_\mu & \mu\alpha.[\beta](M\{[\alpha]M'N/[\alpha]M'\} & \text{(Def 4.2.2)} \\
\llbracket\mu\alpha.[\beta](M\{[\alpha]M'N/[\alpha]M'\})\rrbracket & \to^* & \langle\ M[\square\llbracket N\rrbracket/\alpha],\ \square,\ \beta : \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)}
\end{array}$$

$(\mu\alpha.[\beta]\mu\gamma.[\delta]M)$: $\qquad (\gamma \neq \delta)$

$$\begin{array}{rcll}
[\![\mu\alpha.[\beta]\mu\gamma.[\delta]M]\!] & \to^* & \langle [\![M]\!][\Box/\alpha][\beta/\gamma],\ \Box,\ \delta : \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)} \\
\mu\alpha.[\beta]\mu\gamma.[\delta]M & \to_\mu & \mu\alpha[\delta]M[\beta/\gamma] & \text{(Def 4.2.2)} \\
[\![\mu\alpha[\delta]M[\beta/\gamma]]\!] & \to^* & \langle [\![M]\!][\beta/\gamma][\Box/\alpha],\ \Box,\ \delta : \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)}
\end{array}$$

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

$$\begin{array}{rcll}
\mu\alpha.[\beta]\mu\gamma.[\gamma]M & \to^* & \langle [\![M]\!][\Box/\alpha][\beta/\gamma],\ \Box,\ \beta : \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)} \\
\mu\alpha.[\beta]\mu\gamma.[\gamma]M & \to_\mu & \mu\alpha[\beta]M[\beta/\gamma] & \text{(Def 4.2.2)} \\
[\![\mu\alpha[\beta]M[\beta/\gamma]]\!] & \to^* & \langle [\![M]\!][\beta/\gamma][\Box/\alpha],\ \Box, \beta : \textsc{p}_0\ \rangle & \text{(Proof 4.2.6)}
\end{array}$$

□

4.3 Interpreting λ^{try} in CDC

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

Definition 4.3.1 Translation of λ^{try} into CDC

$$\begin{array}{rcl}
[\![x]\!] & \triangleq & x \\
[\![\lambda x.M]\!] & \triangleq & \lambda x.[\![M]\!] \\
[\![MN]\!] & \triangleq & [\![M]\!][\![N]\!] \\
[\![\text{throw n}(M)]\!] & \triangleq & \textsc{wsc}\ \ \textsc{p}_0\ \ \lambda\circ.\textsc{pp}\ \ \textsc{p}_0\ \ (\textsc{psc}\ \ n\ (c_n\ [\![M]\!])) \\
[\![\text{try } M;\ \underline{\text{catch n}(x) = L}]\!] & \triangleq & (\lambda c_n.\textsc{wsc}\ \ \textsc{p}_0\ \ \lambda n.\textsc{pp}\ \ \textsc{p}_0\ \ (\textsc{psc}\ \ n\ [\![M]\!]))(\lambda x.[\![L]\!]) \\
[\![\text{try } M;\ \overline{\text{catch n}_i(x) = M_i};\ \text{catch m}(x) = L]\!] & & \\
& \triangleq & \\
\multicolumn{3}{r}{(\lambda c_m.\textsc{wsc}\ \ \textsc{p}_0\ \ \lambda m.\textsc{pp}\ \ \textsc{p}_0\ \ (\textsc{psc}\ \ m\ [\![\text{try } M;\ \overline{\text{catch n}_i(x) = M_i}]\!]))(\lambda x.[\![L]\!])}
\end{array}$$

5 — Implementation

This chapter explains the implementation of λ^{try} in Haskell. It examines how the translations from the previous chapter and the CDC library published in [4] are transformed into Haskell code. Finally, it explores the possibility of a language extension to Haskell.

5.1 CDC Library

Jones *et al.* defined a Haskell library for CDC in [4]. We informally present here the interface of their library. For full implementation details, see [4].

```
data CC ans a
data Prompt ans a
data SubCont ans a

instance Monad (CC ans)
```

The type `CC` represents a term in continuation-passing style. `SubCont` represents a series of continuations: functions from a value to a CPS term. `Prompt` represents a prompt for delimiting the control operators.

The type interface for the control operators are:

```
runCC :: (forall ans. CC ans a) -> a
newPrompt :: CC Ans (Prompt ans a)
pushPrompt :: Prompt ans a -> CC ans a -> CC ans a
withSubCont :: Prompt ans b -> (SubCont ans a b -> CC ans b)
               -> CC ans a
pushSubCont :: SubCont ans a b -> CC ans a -> CC ans b
```

The control operators behave as described in Section 2.7.3. The only additional operator is `runCC`. `runCC` takes a `CC` term and runs it to return the value it contains. This is like passed the identity function $\lambda x.x$ to a CPS term to extract its final value.

5.2 Naive Implementation

Blindly transcribing the λ^{try}-to-CDC 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:

```
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)))
```

There is no catch construct because the exception handlers are passed directly to `try`. This is because the body of the `try` block needs access to the handlers. In order to resolve the scoping issues that `throw` also needs access to the exception handlers, calls to `throw` are wrapped in a λ-abstraction and passed to the body. This means that throw is not directly invoked by the program but via these λ-abstractions.

In order to run a `try`-block, it is necessary to generate a prompt, push it, and pass it as an argument to both `try` and `throw`:

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

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

However, we only need a single prompt. The same prompt is reused for setting up all the catch handlers. This prompt should either be created and pushed at the beginning of a `try` statement or it should be an unexported global constant.

This implementation also means we cannot have a variadic number of handlers. Instead, we need to define `try`, `try2`, ... for each case:

```
try2 :: ((a -> CC ans a1) -> (a3 -> CC ans a4) -> CC ans a2)
     -> (a3 -> a2) -> (a -> a2) -> CC ans a2
try2 m h1 h2 =
  try (\t1 ->
    try (\t2 -> m t1 t2) h1)
  h2
```

Naming the handlers takes place in the names of the parameters of the abstraction in the body of `try`:

```
try2 p (\name1 -> \name2 -> return 1)
  (\x -> return $ x+2)  -- first handler bound to name1
  (\x -> return $ x+4)  -- second handler bound to name2
```

This is not like the semantics of λ^{try} which binds the names dynamically:

$$\text{throw } \mathrm{n}(x)$$

will behave differently in contexts where the name n is bound to different exception handlers. To mirror this more closely, we would need a mechanism for dynamic variable binding, for example macros.

Additionally, the type signatures expose `CC`, `Prompt`, and `SubCont`. If we want to present a λ^{try} model of exceptions, we want to abstract over these so user does not rely on knowing them. Unless we do this, we have *leaky abstractions.*

5.3 Improved Implementation

By creating a global prompt `p0`, we can implement all the operators without passing a prompt as an additional argument. This requires exporting the `Prompt` data constructor from the `Prompt` module which was previously hidden. We can then define a `runTry` operation that takes a `CC ans a` term constructed by `try` and returns its value:

```
p0 :: Prompt ans a
p0 = Prompt 0

runTry :: (forall ans. CC ans a) -> a
runTry trycc =
  runCC $ pushPrompt p0 trycc
```

With a global prompt defined, the implementations of `try` and `throw` are much closer to the syntax of the λ^{try}-calculus and to the CDC translation defined in the previous chapter.

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

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

The `catch` block is still implicit in the `try` function. In order bind the correct throw to the correct handler, throw is still wrapped in a λ-abstraction in the definition of try. This also means that we still require separate definitions for multiple handlers:

```
try2 m h1 h2 =
  try (\t ->
    try (m t) h2)
  h1

try3 :: ((t -> CC ans a)
     -> (t' -> CC ans a) -> (t'' -> CC ans a) -> CC ans a)
     -> (t -> CC ans a)
     -> (t' -> CC ans a)
     -> (t'' -> CC ans a)
     -> CC ans a
try3 m h1 h2 h3 =
  try (\t1 ->
    try (\t2 ->
      try (\t3 -> m t1 t2 t3) h3)
    h2)
  h1
```

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.

As a simple example:

```
runTry $
  try2 (\t1 -> \t2 -> return randomNumber)
    (\x -> return $ x + 100)
    (\x -> return $ x + 1000)
```

From this example, we can see we still have the same problem with injecting λ-abstractions to correctly bind names. In order to avoid this, we need to extend the syntax of the language.

5.4 Language Extension

The Haskell2010 standard represents $\perp$ as exceptions [5]. Exceptions can be generated by either `error` or `undefined`:

```
error :: String -> a
undefined :: a
```

Haskell is a lazy language which means that an expression of any type could produce an error when it is actually computed. In a strict language, the error would be produced on assignment. For this reason, $\perp$ is implicitly a member of every type in Haskell. This is why the types of both `error` and `undefined` are universally quantified.

The simplest extension to the language is to introduce named exceptions. Named exceptions are produced using `throw name value`. With this extension, we can introduce two other new language-level operators: `try` and `catch`. `try` takes a term that could produce an exception and returns a `try`-block. `catch` is an infix function from a `try`-block, a name, and an exception handler function to a `try`-block. If `try` and `catch` are both left-associative and have the same precedence, we can construct terms with variable numbers of exception handlers:

```
  try body
    catch name1 handler1
    ...
    catch nameN handlerN
```

`throw` then takes a name and a value and returns a named exception (which can inhabit any type).

For example, we can handle errors produced by the system:

```
  try (parseFile pathName)
    catch fileNotFound (\x -> L)
    catch parseError (\x -> L')
```

and somewhere in the body of M, the corresponding errors can be thrown:

```
  throw fileNotFound pathName
```

It would be trivial to implement static analysis to ensure all possible exceptions had corresponding handlers in the catch block.

6 — Conclusion

This chapter evaluates parts of the methodology and findings of the project. With specific reference to individual components, it examines what should have been done differently. It closes with a presentation of suitable directions for future work, especially regarding research questions thrown up and still unanswered by the present work.

6.0.1 Evaluation

Interpreter

We often treat terms generically: we are not specifically interested in the form of the term. For example the M and N in the term $M[N/x]$ are generic. Generic terms like this were used extensively in the soundness and completeness proofs for the $\lambda\mu$ translation. A minor extension to the interpreter would be to extend it with the constants. Like a value, the evaluation of a constant would leave the constant the same. This extension would allow the expression of these generic terms. In this way, the output of the interpreter could have been used to produce proofs without requiring amendments. Additionally, the interpreter could have output LaTeX by writing an appropriate function of type `State -> String` to pass into the evaluation engine.

Choice of Calculus

CDC was not the correct choice of calculus to facilitate a translation from λ^{try} to Haskell. It was chosen because it easily facilitated a translation to Haskell: the syntax of CDC was intentionally close to Haskell and a CDC library had been implemented in Haskell by the authors of the calculus. However there is an impedance mismatch between CDC and $\lambda\mu$. The translation does not use most of the expressive power of CDC. For example, CDC is capable of generating multiple prompts of different names. Only a single prompt is used by the translation of the $\lambda\mu$-calculus. We do not need a continuation stack or multiple named prompts, we just need a single delimiter. We could get away with a map from prompts to contexts in place of a stack. This would have greatly simplified the translation although it would also require reimplementing the CDC operators. We believe this would be useful to develop in follow up work.

Direct Translation

There is possible a more direct translation of λ^{try} to CDC that is not mediated by a translation to $\lambda\mu$. More importantly, there is likely to be a more direct implementation of λ^{try} to Haskell. By translating through two other calculi, we have left this possibility unexamined. Although we still would want a direct implementation of λ^{try} in Haskell that does not use CDC, we have demonstrated that an implementation is possible and laid down the ground work for this to be done.

Translation and Soundness

The embedding of the $\lambda\mu$-calculus in CDC is not as close to the $\lambda\mu$-calculus as it could be. There are some reduction steps in the destination language that are not reflected by the source language. Specifically, the soundness property we proved was

$$M \to_\mu N \Rightarrow \exists P. [\![M]\!] \to^* P \wedge [\![N]\!] \to^* P$$

This says that the translation of M and the translation of the μ-reduction of M reduce to the same term. By defining the substitition of terms explicitly, as in [11], we may have proved

$$M \to_\mu^* N \Rightarrow [\![M]\!] \to^* [\![N]\!]$$

This says that the translation of M reduces to the translation of N. With this property, the correspondence between the source language and the target language would have be closer.

6.0.2 Conclusion

We defined an interpreter for CDC which we used to experiment with translations from the $\lambda\mu$-calculus. The output of the interpreter was transliterated into derivations that we used to prove the soundness and completeness of our translation. Using this translation, along with van Bakel's λ^{try}-to-$\lambda\mu$ translation, we defined a translation from λ^{try} to CDC. This was used as the basis for some proof-of-concept λ^{try} implementations in Haskell. This project presents a clear demonstration that named exceptions, as introduced by the λ^{try}-calculus, are implementable in Haskell. Apart from a minor syntactic extension, the implementation would not require any machinery currently unavailable to Haskell.

Future Work

Finally, we present a list of suitable directions for future work.

- *Type preservation* – Both λ^{try} and $\lambda\mu$ have type assignment systems. We did not explore whether the types are preserved under the translations we defined. Doing this is will give additional insight into how the λ^{try} calculus relates to delimited continuations and how it can be implemented in Haskell without using CDC.
- *Haskell extension* – Proof-of-concept Haskell libraries are presented and a language extension is proposed but the language extension is not implemented. Given the state of the present work, this will be trivial. By exposing the functionality in a language pragma, we will extend the parser with additional grammar rules.
- *Reimplement CDC without stack* – The implementation of $\lambda\mu$ in CDC only ever uses a single prompt: CDC is overengineered for the purposes of a $\lambda\mu$-translation. We will rewrite CDC to replace the context and prompt stack with a mapping from prompts to contexts. Using this, a simpler translation of $\lambda\mu$ to CDC should be found.
- *Translate λ^{try} directly into Haskell* – The current methodology helped expand our understanding of the relation of λ^{try} to Haskell. Using this, we can write a direct implementation of λ^{try} into Haskell that bypasses CDC entirely.

Bibliography

[1] Alonzo Church. A note on the entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

[2] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

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

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

[5] Simon Marlow (editor). Haskell 2010 language report, 2010.

[6] Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 180–190, 1988.

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

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

[9] Gerald J. Sussman and Guy L. Steele Jr. Scheme: A interpreter for extended lambda calculus (reprint). *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

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

[11] Steffen van Bakel and Maria Grazia Vigliotti. A fully-abstract semantics of lambda-mu in the pi-calculus. In *Proceedings Fifth International Workshop on Classical Logic and Computation, CL&C 2014, Vienna, Austria, July 13, 2014.*, pages 33–47, 2014.

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