Imperial College London

Department of Computing

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

by
William S. Fisher

supervised by
Steffen van Bakel

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

September 2016

Abstract

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 for the Calculus of Delimed Continuations in Haskell. 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 for Haskell that are based on the λ^{try}-calculus.

Acknowledgements

Thanks me

Contents

1 Introduction **1**
- 1.1 Solution . . . 1
- 1.2 Contribution . . . 1

2 Background **2**
- 2.1 Formal Systems . . . 2
  - 2.1.1 Syntax and Grammars . . . 2
  - 2.1.2 Derivation Rules . . . 3
  - 2.1.3 Domain Modelling . . . 3
  - 2.1.4 Derivation Strategies . . . 4
- 2.2 λ-Calculus . . . 5
  - 2.2.1 Syntax . . . 5
  - 2.2.2 Reduction Rules . . . 6
  - 2.2.3 Normal Forms . . . 7
- 2.3 Logic and Types . . . 8
  - 2.3.1 Natural Deduction . . . 8
  - 2.3.2 Sequent Calculus . . . 9
  - 2.3.3 Classical Logic . . . 9
  - 2.3.4 Type Assignment . . . 10
  - 2.3.5 Typed λ-Calculus . . . 11
  - 2.3.6 Curry-Howard Isomophism . . . 11
- 2.4 Haskell . . . 12
  - 2.4.1 Data Types . . . 12
  - 2.4.2 Type Level/Value Level . . . 13
  - 2.4.3 Type Classes . . . 13
- 2.5 Continuations . . . 14
  - 2.5.1 Undelimited Continuations . . . 14
  - 2.5.2 Delimited-Continuations . . . 15
  - 2.5.3 Continuation-Passing Style . . . 15
  - 2.5.4 Monads . . . 16
- 2.6 $\lambda\mu$-Calculus . . . 17
  - 2.6.1 Syntax . . . 17
  - 2.6.2 Reduction Rules . . . 18

2.6.3 Computational Significance . . . 18
2.6.4 Curry-Howard Isomorphism . . . 19
2.7 Calculus of Delimited-Continuations . . . 20
2.7.1 Syntax . . . 20
2.7.2 Reduction Rules . . . 20
2.7.3 Significance . . . 21
2.8 λ^{try}-Calculus . . . 21
2.8.1 Exceptions . . . 22
2.8.2 Syntax . . . 22
2.8.3 Reduction Rules . . . 23
2.8.4 Significance . . . 23

3 CDC Interpreter **24**
3.1 Implementation . . . 24
3.1.1 Data structures . . . 24
3.1.2 Utility Functions . . . 24
3.1.3 Reduction Rules . . . 26
3.1.4 Example . . . 29

4 Translations **31**
4.1 Interpreting λ^{try} in $\lambda\mu$ . . . 31
4.2 Interpreting $\lambda\mu$ in CDC . . . 31
4.2.1 Notation . . . 32
4.2.2 Additional Translations . . . 33
4.2.3 Properties . . . 34
4.3 Interpreting λ^{try} in CDC . . . 38

5 Implementation **39**
5.1 CDC Library . . . 39
5.2 Naive Implementation . . . 39
5.3 Improved Implementation . . . 40
5.4 Language Extension . . . 42

6 Conclusion **43**
6.1 Evaluation . . . 43
6.2 Conclusion . . . 43
6.3 Future Work . . . 43

1 — Introduction

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

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

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

Exceptions have been done but unnamed or dispatch on type. λ^{try} introduces exceptions with names.

Features of computer programs are discovered twice: by logicians and by computer scientists.[6] Exceptions have been mapped to continuations which have been mapped to classical logic. What about a calculus that models exceptions directly? HOw does it behave?

1.1 Solution

Use van Bakel's translation of λ^{try} to $\lambda\mu$. Define a translation from $\lambda\mu$ to CDC, which closely models Haskell's syntax. Write a CDC interpreter for generating derivations that can be transcribed into proofs. Investigate properties of $\lambda\mu$ translation. Use this translation to find a translation from λ^{try} to $\lambda\mu$. Implement λ^{try} directly in Haskell by following this.

1.2 Contributions

This paper makes the following contributions:

- Haskell interpreter for a calculus of delimited continuations, CDC, written by SPJ
- Translation of $\lambda\mu$ to CDC along with proof of soundness and completeness with respect to mu reduction.
- A translation of λ^{try} to CDC.
- A proof of concept implementation of λ^{try} in Haskell, based on this translation.

- The specification for a language extension for named exceptions in Haskell, based on λ^{try}.

2 — Background

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

2.1 Formal Systems

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

2.1.1 Syntax and Grammars

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

We defined grammars using Backus-Naur Form or BNF:

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

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

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

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

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

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

2.1.2 Derivation Rules

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

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

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

2.1.3 Domain Modelling

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

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

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

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

2.1.4 Derivation Strategies

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

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

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:

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

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

1. $t\ o\ f \quad \square\ a\ t$
2. $f\ a\ t \quad t\ o\ \square$

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

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

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

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

2.2 λ-Calculus

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

2.2.1 Syntax

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

Definition 2.2.1 (Grammar for untyped λ-calculus)

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

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

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

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

Figure 2.2: Examples of valid λ-abstractions

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

[1] General recursive functions and Turing machines, respectively

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

Figure 2.3: Examples of valid applications

2.2.2 Reduction Rules

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

Definition 2.2.2 (Substitution notation for λ-terms)

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

β-reduction

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

Definition 2.2.3 (β-reduction for λ-calculus)

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

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

α-reduction

The 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 [5]

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

Definition 2.2.5 FREE VARIABLES fv OF λ-TERMS
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)

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

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

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

2.2.3 Normal Forms

A term is a 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 can define β-normal form on λ-terms with the following BNF grammar: We use $\to_\beta^{nf}$ to denote a term in β-normal form. Again, we can omit the β subscript if it is clear from context.

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

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, we can take for granted 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 $\rightarrow$. 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

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

The $\rightarrow \mathcal{E}$ rule says that given the statement $A \rightarrow B$ and the statement A, we can conclude B. 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 $A \rightarrow B$. 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

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

By restricting ourselves to these rules, we are working within *Implicative Intuitionistic Logic* (IIL). [2]

2.3.2 Sequent Calculus

Gentzen explored natural deduction through the sequent calculus as well. We have followed Girard *et al.*'s observation that the syntax of the sequent calculus is overcomplicated for the purpose of natural deduction.[3]

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

$$\Gamma \vdash T$$

On the left-hand side, the Γ represents a sequence of statements called the *antecedent*. On the right-hand side of the $\vdash$, T represents a different sequence of statements called the *succedent*. The whole sequent denotes that the conjunction of the statements in the antecedent imply the disjunction of the statements in the succedent. That is to say, if all the statements on the left are true then at least one of the statements on the right is true:

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

2.3.3 Classical Logic

Gentzen's classical natural deduction used the sequent calculus and introduced a set structural rules for manipulating sequents:

[2] Importantly, 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

Definition 2.3.2 Structural rules of the sequent calculus

$$\frac{A, C, D, A' \vdash B}{A, D, C, A' \vdash B}\ \mathcal{L}X \qquad \frac{A \vdash B, C, D, B'}{A \vdash B, D, C, B'}\ \mathcal{R}X$$

$$\frac{A \vdash B}{A, C \vdash B}\ \mathcal{L}W \qquad \frac{A \vdash B}{A \vdash B, C}\ \mathcal{R}W$$

$$\frac{A, C, C \vdash B}{A, C \vdash B}\ \mathcal{L}C \qquad \frac{A \vdash C, C, B}{A \vdash C, B}\ \mathcal{R}C$$

$$\frac{A \vdash B \quad A', B \vdash C}{A, A' \vdash C}\ \textbf{Cut}$$

The first set of rules are left and right exchange rules. They express the commutativity of statements in the left and right sequences. The second set of rules, weakening rules, allows the introduction of new formula either the left or right sequences. The third set of rules, contraction rules, allow the contraction of multiple occurrences of a formula into a single occurrence. The cut rule allows us to replace assumptions of formulae with their concrete proofs, if we have proved them somewhere else.

In addition to these, the system maintains the standard introduction/elimination rules from natural deduction. The introduction/elimination rules have variants for manipulating the right sequence or the left sequence of statements.

2.3.4 Type Assignment

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

$$M : \alpha$$

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

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

Definition 2.3.3 (Grammar for constructing types)

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

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

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 be applied to a value of type A and returns one of type B. Type assignment rules for the *typed* λ-calculus are:

$$\frac{\Gamma \vdash M : A \to B \qquad N : A}{\Gamma \vdash MN : B}$$

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

The first rule states that the application of a term of type $A \to B$ to a term of type A has type B. The 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 sometimes undesirable properties.

Example 2.3.4 (Type-assignment restricts set of valid terms)

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

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

2.3.6 Curry-Howard Isomophism

The Curry-Howard isomorphism states that their 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.[6]

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

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

The correspondence between logic and programs is not limited to IIL and the typed lambda-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:

```
1        data Animal a = Dog a
2          | 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.

```
1        > let hector = Cat
2        > :t hector
3        hector :: Animal a
4
5        > let topaz = Dog "foo"
6        > :t topaz
7        topaz :: Animal String
```

The type parameter is constrained by the type of the value the data constructor was initialized with. In the example above, calling the `Dog` data constructor with a string makes the type `Animal String` rather than the more general `Animal a`.

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

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

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

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

```
type Name = String
```

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

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 One Two = ThreeOrMore
  add Two One = 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 **??**, compound λ-terms can be decomposed into a dominant term and a context:

Assume that $M \rightarrow_\beta M'$

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

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

The reduction strategy will define how the term will be decomposed. When M has β-reduced to a value, M', then the hole of the context $\Box N$ is filled to form $M'N$. When the hole of a context is filled, the reduction of the compound term continues. This means the context contains the remaining terms to be reduced: it represents how reduction will continue. Thus a context is also called a *continuation*.

2.5.1 Undelimited Continuations

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

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

Figure 2.5: Decomposing a term into multiple contexts

By amalgamating continuations into one big continuation we only have two components at 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 a stack of continuations when decomposing complex terms, we can keep continuations separated:

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

Figure 2.6: Decomposing a term into multiple contexts

Here, when a dominant term has been reduced, the reduct is returned to 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. These are commonly called *prompts*, after Felleisen first introduced them as such. 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.

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.

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

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

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

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

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

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

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 $\lambda m.N$:

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

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 a 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 `>>=` that binds the first suspended computation to a continuation which returns another suspended computation[3]:

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

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

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:

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

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

2.6 $\lambda\mu$-Calculus

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

2.6.1 Syntax

Just as λ introduces λ-abstractions, 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 $\alpha, \beta, \ldots$

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

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

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

2.6.2 Reduction Rules

The terse reduction rule of *iii*) simply states that the application of a $\lambda\mu$-abstraction $\mu\alpha.M$ to a term N applies all the sub-terms of M labelled $[\alpha]$ to N and relabels them with a fresh μ variable. It can be thought of as a λ-abstraction that can be applied to any number of variables. If we knew how many variables the term is applied to, we could replace

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

with

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

[4]

2.6.3 Computational Significance

The $\lambda\mu$-calculus makes use of *applicative contexts*: An $\lambda\mu$-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. 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 [1]

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

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 $\Box N$. Informally, we can imagine that the variable at the head of the μ-abstraction now maps to this 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 $\Box N$ and the context will destroyed. For example, let us replace M with $\mu \circ .[\alpha](\lambda s.fs)$: [4]

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

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

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.

[4] 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 [2]. 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 and undelimited continuations. CDC provides a set of operations 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 tuple 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 : 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.

Definition 2.7.2 (Operational semantics for CDC) [2]

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

2.7.3 Significance

The additional terms behave as follows:

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

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

2.8 λ^{try}-Calculus

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

2.8.1 Exceptions

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

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

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

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

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

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

2.8.2 Syntax

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

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

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

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

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

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

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

2.8.3 Reduction Rules

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

Definition 2.8.2 (λ^{try} REDUCTION RULES)

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

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 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. Portions of source code are examined in detail although the full source can be found in the appendix.

3.1 Implementation

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

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

3.1.1 Data structures

There are two data types for representing CDC terms, `Value` and `Expr` (Figure **??**). 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

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

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

Figure 3.1: Data structures for the CDC interpreter

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

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

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

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

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

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

PushPrompt

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

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

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

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

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

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

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

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

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

Figure 3.2: Utility functions for CDC interpreter

WithSubCont

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

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

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

PushSubCont

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

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

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

Substitution

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

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

NewPrompt

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

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

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

3.1.4 Example

The terms *pushPrompt*, *newPrompt*, *withSubCont*, and *pushSubCont* are too long to work with easily. We define the following aliases for convenience:

Definition 3.1.1 CDC ALIASES FOR OPERATORS

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

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

```
1    np = NewPrompt
2    pp = PushPrompt
3    wsc = WithSubCont
4    psc = PushSubCont
5    var = Val . Var
6    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.

```
1    evalFull :: (State -> String) -> State -> IO ()
2    evalFull formatFn st = do
3      putStrLn $ formatFn st
4      let st' = eval st in
5        if st == st' then
6          putStrLn "QED"
7        else
8          evalFull formatFn st'
```

We can translate the CDC term

$$(\lambda p.\text{PP}\ p\ (\text{WSC}\ p\ (\lambda\alpha.\text{PSC}\ \alpha\ s))\ t)\ \text{NP}$$

into the data structures defined by our interpreter:

```
1    term = App
2      (abst 'p'
3        (pp (var 'p') (App
4          (wsc (var 'p') (abst 'a'
5            (psc (var 'a') (var 's'))))
6          (var 't'))))
7      np
```

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

```
(((\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:

$$
\begin{array}{lcccc}
 & (\lambda p.\text{PP}\ p\ (\text{WSC}\ p\ (\lambda\alpha.\text{PSC}\ \alpha\ s))\ t)\ \text{NP} & \Box & [] & 0 \\
\to_{CDC} & \text{NP} & (\lambda p \ldots t)\ \Box & [] & 0 \\
\to_{CDC} & 0 & (\lambda p \ldots t)\ \Box & [] & 1 \\
\to_{CDC} & (\lambda p.\text{PP}\ p\ (\text{WSC}\ p\ (\lambda\alpha.\text{PSC}\ \alpha\ s))\ t)\ 0 & \Box & [] & 1 \\
\to_{\beta} & (\text{PP}\ p\ (\text{WSC}\ p\ (\lambda\alpha.\text{PSC}\ \alpha\ s))\ t))[p/0] & \Box & [] & 1 \\
\to_{\beta} & \text{PP}\ 0\ (\text{WSC}\ 0\ (\lambda\alpha.\text{PSC}\ \alpha\ s))\ t) & \Box & [] & 1 \\
\to_{CDC} & (\text{WSC}\ 0\ (\lambda\alpha.\text{PSC}\ \alpha\ s))\ t & \Box & 0:\Box & 1 \\
\to_{CDC} & \text{WSC}\ 0\ (\lambda\alpha.\text{PSC}\ \alpha\ s)) & \Box\ t & 0:\Box & 1 \\
\to_{CDC} & (\lambda\alpha.\text{PSC}\ \alpha\ s))(\Box t) & \Box & \Box & 1 \\
\to_{\beta} & (\text{PSC}\ \alpha\ s)[\Box t/\alpha] & \Box & \Box & 1 \\
\to_{\beta} & \text{PSC}\ \Box t\ s & \Box & \Box & 1 \\
\to_{CDC} & s & \Box & \Box t:\Box:\Box & 1 \\
\to_{CDC} & s & \Box\ t & \Box:\Box & 1 \\
\to_{CDC} & st & \Box & \Box:\Box & 1
\end{array}
$$

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 $\lambda\mu$:

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

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

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

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

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)

$$(\lambda \mathrm{p}_0\ .\mathrm{PP}\ \ \mathrm{p}_0\ \ [\![M]\!])\ \mathrm{NP}$$

This creates a new prompt p_0 which is in scope for all subterms of M. It also prepares the stack by pushing 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)

$$\begin{array}{lcl}
[\![x]\!] & \triangleq & x \\
[\![\lambda x.M]\!] & \triangleq & \lambda x.[\![M]\!] \\
[\![MN]\!] & \triangleq & [\![M]\!][\![N]\!] \\
[\![\mu\alpha.M]\!] & \triangleq & \mathrm{WSC}\ \ \mathrm{p}_0\ \ (\lambda\alpha.\mathrm{PP}\ \ \mathrm{p}_0\ \ [\![M]\!]) \\
[\![[\beta]M]\!] & \triangleq & \mathrm{PSC}\ \ \beta\ [\![M]\!]
\end{array}$$

To implement μ-abstractions, we capture the subcontinuation until the last occurrence of p_0 on the stack. This subcontinuation is bound to α which ensures the subcontinuation is distributed to all occurrences of α in M. 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.

4.2.1 Notation

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

$$M \quad D \quad E$$

where each column corresponds to one component of the original abstract machine. 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

$$\mathrm{PSC}\ \beta\ M \quad \Box \quad \mathrm{p}_0 \quad \rightarrow_{CDC} \quad M \quad \Box \quad \beta : \Box : \mathrm{p}_0$$

we will instead write

$$\mathrm{PSC}\ \beta\ M \quad \Box \quad \mathrm{p}_0 \quad \rightarrow_{CDC} \quad M \quad \Box \quad \beta : \mathrm{p}_0$$

4.2.2 Additional Translations

We alter μ-reduction to consume multiple variables. 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 PSC $\Box$ $[\![M']\!]$. CDC reduces PSC $\Box$ $[\![M']\!]$ 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

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

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

$$\begin{array}{c} (\mu\alpha.[\alpha](\lambda x.\lambda y.xy))st \\ \rightarrow_\mu \quad \mu\alpha.([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\}) \end{array}$$

From here we either can continue with the $\lambda\mu$ reduction:

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

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

$$\begin{array}{lcll} & [\![\mu\alpha.([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst)\}]\!] & & \\ \triangleq & \text{WSC P}_0\ (\lambda\alpha.\text{PP P}_0\ [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!]) & \Box & \text{P}_0 \\ \rightarrow & (\lambda\alpha.\text{PP P}_0\ [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!])(\Box) & \Box & [] \\ \rightarrow_\beta & (\text{PP P}_0\ [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!])[\Box/\alpha] & \Box & [] \\ \rightarrow & \text{PP P}_0\ [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!][\Box/\alpha] & \Box & [] \\ \rightarrow & [\![([\alpha](\lambda x.\lambda y.xy))\{[\alpha]Mst/[\alpha]Mst\})]\!][\Box/\alpha] & \Box & \text{P}_0 \\ \rightarrow & ((\text{PSC } \alpha(\lambda x.\lambda y.xy))[\Box st/\alpha])[\Box/\alpha] & \Box & \text{P}_0 \\ \rightarrow & (\text{PSC } \Box st\ (\lambda x.\lambda y.xy))[\Box/\alpha] & \Box & \text{P}_0 \\ \rightarrow & \text{PSC } \Box st\ (\lambda x.\lambda y.xy) & \Box & \text{P}_0 \\ \rightarrow & (\lambda x.\lambda y.xy) & \Box & \Box st : \text{P}_0 \\ \rightarrow & (\lambda x.\lambda y.xy) & \Box st & \text{P}_0 \\ \rightarrow & (\lambda x.\lambda y.xy)st & \Box & \text{P}_0 \\ \rightarrow_\beta & (\lambda y.sy)t & \Box & \text{P}_0 \\ \rightarrow_\beta & st & \Box & \text{P}_0 \end{array}$$

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

4.2.3 Properties

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

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

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

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

	$[\![\mu\alpha.[\alpha]M]\!]$		
$\triangleq$	$\text{WSC P}_0\ \lambda\alpha.\text{PP P}_0\ (\text{PSC } \alpha [\![M]\!])$	D	P_0
$\to_{CDC}$	$(\lambda\alpha.\text{PP P}_0\ (\text{PSC } \alpha [\![M]\!]))(D)$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PP P}_0\ (\text{PSC } \alpha\ [\![M]\!]))[D/\alpha]$	$\Box$	$[]$
$\to_{CDC}$	$\text{PP P}_0\ (\text{PSC } \alpha\ [\![M]\!])[D/\alpha]$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PSC } \alpha\ [\![M]\!])[D/\alpha]$	$\Box$	P_0
$\to_{CDC}$	$\text{PSC } D\ [\![M]\!][D/\alpha]$	$\Box$	P_0
$\to_{CDC}$	$[\![M]\!][\Box/\alpha]$	$\Box$	$D : \text{P}_0$
$\to_{CDC}$	$[\![M]\!]$	$\Box$	$D : \text{P}_0$
$\to_{CDC}$	$[\![M]\!]$	D	P_0
$\triangleq$	M	D	P_0

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

	$[\![(\mu\alpha.[\alpha]M)\overline{N}]\!]$		
$\triangleq$	$[\![(\mu\alpha.[\alpha]M)]\!][\![\overline{N}]\!]$		
$\triangleq$	$(\text{WSC P}_0\ \lambda\alpha.\text{PP P}_0\ (\text{PSC } \alpha\ [\![M]\!]))[\![\overline{N}]\!]$	$\Box$	P_0
$\to_{CDC}$	$\text{WSC P}_0\ \lambda\alpha.\text{PP P}_0\ (\text{PSC } \alpha\ [\![M]\!])$	$\Box[\![\overline{N}]\!]$	P_0
$\to_{CDC}$	$\lambda\alpha.\text{PP P}_0\ (\text{PSC } \alpha\ [\![M]\!])(\Box[\![\overline{N}]\!])$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PP P}_0\ (\text{PSC } \alpha\ [\![M]\!]))[\Box[\![\overline{N}]\!]/\alpha]$	$\Box$	$[]$
$\to_{CDC}$	$\text{PP P}_0\ (\text{PSC } \alpha\ [\![M]\!])[\Box[\![\overline{N}]\!]/\alpha]$	$\Box$	$[]$
$\to_{CDC}$	$\text{PP P}_0\ (\text{PSC } \Box[\![\overline{N}]\!]\ [\![M]\!][\Box[\![\overline{N}]\!]/\alpha])$	$\Box$	$[]$
$\to_{CDC}$	$\text{PSC } \Box[\![\overline{N}]\!]\ [\![M]\!][\Box[\![\overline{N}]\!]/\alpha]$	$\Box$	P_0
$\to_{CDC}$	$[\![M]\!][\Box[\![\overline{N}]\!]/\alpha]$	$\Box$	$\Box[\![\overline{N}]\!] : \text{P}_0$
$\to_{CDC}$	$[\![M]\!][\Box[\![\overline{N}]\!]/\alpha]$	$\Box[\![\overline{N}]\!]$	P_0

This final state is P. Now we must prove $[\![N]\!] \to^* P$:

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

In this final reduction step, α can it occur in $[\![M]\!][\Box\overline{N}/\alpha]$ because all occurrences have been substituted for $\Box[\![\overline{N}]\!]$.

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

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

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

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

	$[\![\mu\alpha.[\beta]\mu\gamma.[\delta]M]\!]$		
$\triangleq$	$\text{WSC P}_0\ \lambda\alpha.\text{PP P}_0\ (\text{PSC}\ \beta\ [\![\mu\gamma.[\delta]M]\!])$	$\Box$	P_0
$\to_{CDC}$	$(\lambda\alpha.\text{PP P}_0\ (\text{PSC}\ \beta\ [\![\mu\gamma.[\delta]M]\!]))(\Box)$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PP P}_0\ (\text{PSC}\ \beta\ [\![\mu\gamma.[\delta]M]\!]))[\Box/\alpha]$	$\Box$	$[]$
$\to_{CDC}$	$\text{PP P}_0\ (\text{PSC}\ \beta\ [\![\mu\gamma.[\delta]M]\!])[\Box/\alpha]$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PSC}\ \beta\ [\![\mu\gamma.[\delta]M]\!])[\Box/\alpha]$	$\Box$	P_0
$\to_{CDC}$	$\text{PSC}\ \beta\ [\![\mu\gamma.[\delta]M]\!][\Box/\alpha]$	$\Box$	P_0
$\to_{CDC}$	$[\![\mu\gamma.[\delta]M]\!][\Box/\alpha]$	$\Box$	$\beta : \text{P}_0$
$\triangleq$	$(\text{WSC P}_0\ \lambda\gamma.\text{PP P}_0\ (\text{PSC}\ \delta[\![M]\!]))[\Box/\alpha]$	$\Box$	$\beta : \text{P}_0$
$\to_{CDC}$	$\text{WSC P}_0\ \lambda\gamma.\text{PP P}_0\ (\text{PSC}\ \delta[\![M]\!])[\Box/\alpha]$	$\Box$	$\beta : \text{P}_0$
$\to_{CDC}$	$(\lambda\gamma.\text{PP P}_0\ (\text{PSC}\ \delta[\![M]\!])[\Box/\alpha])(\beta)$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PP P}_0\ (\text{PSC}\ \delta[\![M]\!])[\Box/\alpha])[\beta/\gamma]$	$\Box$	$[]$
$\to_{CDC}$	$\text{PP P}_0\ (\text{PSC}\ \delta[\![M]\!])[\Box/\alpha][\beta/\gamma]$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PSC}\ \delta[\![M]\!])[\Box/\alpha][\beta/\gamma]$	$\Box$	P_0
$\to_{CDC}$	$\text{PSC}\ \delta[\![M]\!][\Box/\alpha][\beta/\gamma]$	$\Box$	P_0
$\to_{CDC}$	$[\![M]\!][\Box/\alpha][\beta/\gamma]$	$\Box$	$\delta : \text{P}_0$

	$[\![\mu\alpha[\delta]M[\beta/\gamma]]\!]$		
$\triangleq$	$\text{WSC P}_0\ \lambda\alpha.\text{PP P}_0\ (\text{PSC}\ \delta\ [\![M]\!][\beta/\gamma])$	$\Box$	P_0
$\to_{CDC}$	$(\lambda\alpha.\text{PP P}_0\ (\text{PSC}\ \delta\ [\![M]\!][\beta/\gamma]))(\Box)$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PP P}_0\ (\text{PSC}\ \delta\ [\![M]\!][\beta/\gamma]))[\Box/\alpha]$	$\Box$	$[]$
$\to_{CDC}$	$\text{PP P}_0\ (\text{PSC}\ \delta\ [\![M]\!][\beta/\gamma])[\Box/\alpha]$	$\Box$	$[]$
$\to_{CDC}$	$(\text{PSC}\ \delta\ [\![M]\!][\beta/\gamma])[\Box/\alpha]$	$\Box$	P_0
$\to_{CDC}$	$\text{PSC}\ \delta\ [\![M]\!][\beta/\gamma][\Box/\alpha]$	$\Box$	P_0
$\to_{CDC}$	$[\![M]\!][\beta/\gamma][\Box/\alpha]$	$\Box$	$\delta : \text{P}_0$

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

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

$$
\begin{array}{lcll}
 & [\![\mu\alpha.[\beta]\mu\gamma.[\gamma]M]\!] & & \\
\triangleq & \text{WSC P}_0\ \lambda\alpha.\text{PP P}_0\ (\text{PSC}\ \beta\ [\![\mu\gamma.[\gamma]M]\!]) & \Box & \text{P}_0 \\
\to_{CDC} & (\lambda\alpha.\text{PP P}_0\ (\text{PSC}\ \beta\ [\![\mu\gamma.[\gamma]M]\!]))(\Box) & \Box & [] \\
\to_{CDC} & (\text{PP P}_0\ (\text{PSC}\ \beta\ [\![\mu\gamma.[\gamma]M]\!]))[\Box/\alpha] & \Box & [] \\
\to_{CDC} & \text{PP P}_0\ (\text{PSC}\ \beta\ [\![\mu\gamma.[\gamma]M]\!])[\Box/\alpha] & \Box & [] \\
\to_{CDC} & (\text{PSC}\ \beta\ [\![\mu\gamma.[\gamma]M]\!])[\Box/\alpha] & \Box & \text{P}_0 \\
\to_{CDC} & \text{PSC}\ \beta\ [\![\mu\gamma.[\gamma]M]\!][\Box/\alpha] & \Box & \text{P}_0 \\
\to_{CDC} & [\![\mu\gamma.[\gamma]M]\!][\Box/\alpha] & \Box & \beta : \text{P}_0 \\
\triangleq & (\text{WSC P}_0\ \lambda\gamma.\text{PP P}_0\ (\text{PSC}\ \gamma[\![M]\!]))[\Box/\alpha] & \Box & \beta : \text{P}_0 \\
\to_{CDC} & \text{WSC P}_0\ \lambda\gamma.\text{PP P}_0\ (\text{PSC}\ \gamma[\![M]\!])[\Box/\alpha] & \Box & \beta : \text{P}_0 \\
\to_{CDC} & (\lambda\gamma.\text{PP P}_0\ (\text{PSC}\ \gamma[\![M]\!])[\Box/\alpha])(\beta) & \Box & [] \\
\to_{CDC} & (\text{PP P}_0\ (\text{PSC}\ \gamma[\![M]\!])[\Box/\alpha])[\beta/\gamma] & \Box & [] \\
\to_{CDC} & \text{PP P}_0\ (\text{PSC}\ \gamma[\![M]\!])[\Box/\alpha][\beta/\gamma] & \Box & [] \\
\to_{CDC} & (\text{PSC}\ \gamma[\![M]\!])[\Box/\alpha][\beta/\gamma] & \Box & \text{P}_0 \\
\to_{CDC} & \text{PSC}\ \beta[\![M]\!][\Box/\alpha][\beta/\gamma] & \Box & \text{P}_0 \\
\to_{CDC} & [\![M]\!][\Box/\alpha][\beta/\gamma] & \Box & \beta : \text{P}_0
\end{array}
$$

$$
\begin{array}{lcll}
 & [\![\mu\alpha[\beta]M[\beta/\gamma]]\!] & & \\
\triangleq & \text{WSC P}_0\ \lambda\alpha.\text{PP P}_0\ (\text{PSC}\ \beta\ [\![M]\!][\beta/\gamma]) & \Box & \text{P}_0 \\
\to_{CDC} & (\lambda\alpha.\text{PP P}_0\ (\text{PSC}\ \beta\ [\![M]\!][\beta/\gamma]))(\Box) & \Box & [] \\
\to_{CDC} & (\text{PP P}_0\ (\text{PSC}\ \beta\ [\![M]\!][\beta/\gamma]))[\Box/\alpha] & \Box & [] \\
\to_{CDC} & \text{PP P}_0\ (\text{PSC}\ \beta\ [\![M]\!][\beta/\gamma])[\Box/\alpha] & \Box & [] \\
\to_{CDC} & (\text{PSC}\ \beta\ [\![M]\!][\beta/\gamma])[\Box/\alpha] & \Box & \text{P}_0 \\
\to_{CDC} & \text{PSC}\ \beta\ [\![M]\!][\beta/\gamma][\Box/\alpha] & \Box & \text{P}_0 \\
\to_{CDC} & [\![M]\!][\beta/\gamma][\Box/\alpha] & \Box & \beta : \text{P}_0
\end{array}
$$

□

Theorem 4.2.8 (Completeness of $[\![\cdot]\!]$)

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

Proof. By induction on the definition of $[\![M]\!]$.

$$\boxed{\mu\alpha.[\alpha]M}$$

From the proof for Theorem **??**, we know that:

$$[\![\mu\alpha.[\alpha]M]\!] \ \to^{nf} \ [\![M]\!] \ \Box \ \text{P}_0$$

From the definition of $\to_\mu$:

$$\mu\alpha.[\alpha]M \to^{nf} M$$

And from the translation in **??**:

$$[\![M]\!] \triangleq [\![M]\!] \quad \square \quad \mathrm{P}_0$$

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

...

$$\boxed{(\mu\alpha.[\beta]M)N \qquad (\alpha \neq \beta)}$$

...

$$\boxed{(\mu\alpha.[\beta]\mu\gamma.[\delta]M) \qquad (\gamma \neq \delta)}$$

...

$$\boxed{(\mu\alpha.[\beta]\mu\gamma.[\gamma]M)}$$

... $\square$

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

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

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 [2] are transformed into Haskell code. Finally, it explores the possibility of a language extension to Haskell.

5.1 CDC Library

5.2 Naive Implementation

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

```
1   try :: Prompt ans b
2          -> ((t -> CC ans a) -> CC ans a1) -> (t -> CC ans a1)
3          -> CC ans a1
4   try p0 m handler = withSubCont p0 (\n ->
5       pushPrompt p0 (pushSubCont n (m $ \x -> throw p0 n handler x)))
6
7   throw :: Prompt ans b
8            -> SubCont ans a1 b -> (t -> CC ans a1) -> t
9            -> CC ans a
10  throw p0 n c m = withSubCont p0 (\_ ->
11    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 lambda-abstraction and passed to the body. This means that throw is not directly invoked by the program but via these lambda-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 means we cannot have variadic numbers of handlers. Instead, we need to define `try`, `try2`, ... for different numbers of handlers:

```
try2 p0 m h1 h2 = withSubCont p0 (\n ->
    pushPrompt p0 (pushSubCont n (m (\x -> throw p0 n h1 x)
                                    (\x -> throw p0 n h2 x))))
```

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:

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

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. 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 :: ((a -> CC ans b) -> CC ans r) -> (a -> r) -> CC ans r
try m handler =
  withSubCont p0 (\n -> pushPrompt p0
    (pushSubCont n (m $ \x -> throw n $ return $ 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 lambda abstraction in the definition of try. This also means that we still require separate definitions for multiple handlers:

```
try2 m h1 h2 =
  withSubCont p0 (\n -> pushPrompt p0
    (pushSubCont n (m (\x -> throw n $ return $ h1 x)
                      (\x -> throw n $ return $ h2 x))))
```

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 (\thr1 -> \thr2 -> thr1 randomNumber)
              (\x -> x + 100) -- first handler
              (\x -> x + 1000) -- second handler
```

From this example, we can see we still have the same problem with injecting lambdas 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. [**?**] 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

6.1 Evaluation

6.2 Conclusion

6.3 Future Work

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

Bibliography

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

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

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

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

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

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