1 — 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.

1.1 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.

1.1.1 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.

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

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

Figure 1.1: Grammar for producing the letters t or f connected by the letters a or o

1.1.2 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 1.1.1, we add derivation rules:

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

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

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

1.1.3 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.

1.1.4 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.

1.2 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.

1.2.1 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 1.2.1 (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 1.2: 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 1.3: Examples of valid applications

1.2.2 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 1.2.2 (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 1.2.3 (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 1.2.4 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 1.2.5 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 1.2.6 (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

1.2.3 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.

1.3 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.

1.3.1 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 1.3.1 $\rightarrow$ 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]

1.3.2 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:

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

1.3.3 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 1.3.2 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.

1.3.4 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 1.3.3 (Grammar for constructing types)

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

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

1.3.5 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 1.3.4 (Type-assignment restricts set of valid terms)

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

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

1.3.6 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
$\to \mathcal{E}$	$\dfrac{\Gamma \vdash M : A \to B \qquad N : A}{\Gamma \vdash MN : B}$	$\dfrac{A \to B \quad A}{B}$
$\to \mathcal{I}$	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$	$\dfrac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \to B}$

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

1.4 Haskell

1.4.1 Data Types

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

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

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

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

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

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

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

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

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

1.4.2 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.

1.4.3 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.

1.5 Continuations

As in the example in Figure 1.1.4, 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 1.4: 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*.

1.5.1 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 1.5: 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.

1.5.2 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 1.6: 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.

1.5.3 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 1.5.1 (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 1.7: 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:

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

1.5.4 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:

```
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 `>>=`.

1.6 $\lambda\mu$-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.

1.6.1 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 1.6.1 (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 1.6.2 (Reduction rules for $\lambda\mu$-calculus)

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

1.6.2 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]

1.6.3 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 1.6.3 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 1.6.4

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 1.6.5

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 1.6.6

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)

1.6.4 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 1.6.7 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.

1.7 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.

1.7.1 Syntax

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

Definition 1.7.1 (Grammar for CDC)

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

1.7.2 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 1.7.2 (Operational semantics for CDC) [2]

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

1.7.3 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 1.7.2 also encodes the reduction strategy. The first block of rules define in what order redexes are reducted.

1.8 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.

1.8.1 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.

1.8.2 Syntax

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

Definition 1.8.1 (Grammar of the λ^{try}-calculus)

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

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

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

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.

1.8.3 Reduction Rules

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

Definition 1.8.2 (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.

1.8.4 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.

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.