Imperial College London

FINAL PROJECT REPORT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

An Investigation into Adding Exception Handling to Haskell

Author:
Călin Fărcaș

Supervisor:
Steffen van Bakel

June 9, 2020

Submitted in partial fulfillment of the requirements for the MEng Computing Degree of Imperial College London

Abstract

We explore the possibility of adding the notion of exception handling by name to Haskell. We begin by presenting some relevant formal systems, such as the λ-Calculus, $\lambda\mu$ and, notably, λ^{try}, which is a calculus that models exception handling. We then review the previous work done on the subject, which involves translating λ^{try} into $\lambda\mu$ and $\lambda\mu$ into a calculus called CDC in order to obtain a Haskell implementation of λ^{try} based on an existing CDC library. Unlike the previous work, we do not follow the $\lambda\mu$ and CDC path, but seek out a more direct implementation, possibly involving extensions to the language itself. We present a λ^{try} Haskell evaluator based on the `Either` data type and prove that it preserves reduction. We review GHC, the Haskell compiler, and explore the calculi that it is based on: Hindley-Milner and System FC. We outline that we do not believe a translation from λ^{try} to Hindley-Milner or System FC is possible and develop our evaluator into a fully fledged extension to Hindley-Milner which allows the mapping of λ^{try} to it. We prove that our mapping preserves reduction. We conclude with an evaluation of our results and an outlining of some possibilities for further development.

Acknowledgements

I would like to thank my supervisors, Steffen van Bakel and Nicolas Wu, for our very fruitful discussions and the continuous support that they have provided during these trying times. I have had a wonderful experience working on this project and it would not have been possible without them.

I also want to thank Bianca for her patience with my countless hours of being lost in theoretical proofs in the last months - and, not least, I would like to thank my parents for supporting me in the last 4 years.

Contents

Chapter 1

Introduction

What does it mean to *compute* something? This is a question of utmost importance for computer scientists, and there have been many ways that people have attempted to deal with this matter. Two of the more well-known approaches were Alan Turing's *Turing Machines* and Alonzo Church's λ-*Calculus*, both of which were introduced to model computation.

In particular, the λ-Calculus has become the basis for the functional programming paradigm, perhaps also due to its simple nature: everything is either a variable, an abstraction, or an application. This is powerful enough to model a very large subset of the computation that might occur in a computer program, but also simple enough to be reasoned about. In its basic form, however, it is still not expressive enough to meaningfully model certain concepts, one of these being exception handling.

In [1], van Bakel presented λ^{try} - an extension of the λ-Calculus which formally models exceptions and exception handling in a functional style. Part of the novelty that this brings is that exceptions are being discriminated by name, instead of the usual approaches - which are discrimination by type (Java, Haskell and others), or not discriminating them at all (Javascript).

This calculus (in an earlier version) has been studied along the years: various properties were proven by Baciu in [2], and Griffiths further studied and developed certain extensions of it in [3]. In particular, in [4], Fisher explored an implementation of the λ^{try}-Calculus in Haskell. This implementation did work as a proof of concept - however, as mentioned in his paper, the end result suffered from a number of shortcomings, many of which were due to the fact that, in order to come up with an implementation in Haskell, the original λ^{try}-Calculus was first translated through two other systems. This proved that a translation to Haskell was possible, while leaving open ended the issue of the existence of a direct, more practical implementation.

This project aims to explore the possibility of such an implementation and what level of language modification it would imply, as well as to further study any relevant properties of it.

Chapter 2

Background

This chapter sets the context for the whole project and outlines previous work related to the implementation of λ^{try} in Haskell.

2.1 Formal Systems

A *formal system* [5] is a mathematical tool used to model some domain in order to predict, extract, prove and illustrate various properties of the domain.

Formal systems come in many forms and flavours. For the purpose of this report, we are going to assume the following (reasonable) definition: a formal system is given by an *alphabet*, a *grammar* and a set of *derivation rules*. We explain each of them below.

2.1.1 Alphabet

The alphabet of a formal system is a set of symbols that defines the vocabulary of the system: everything that the system can "talk about", it has to do so using only symbols that exist in the alphabet.

For example, an alphabet could be the set of digits $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ or the set of upper case letters $\{A, B, ..., Z\}$.

Then, a concatenation of symbols from the alphabet is called a *formula* - for the example alphabets above, some possible formulas are 007 and XYZ, respectively.

2.1.2 Grammar

A *grammar* is a set of rules that specify how to construct more complex formulas from simpler formulas. It is common that the rules have the following form:

$$\mathcal{F} ::= f_1 \mid \ldots \mid f_n$$

$\mathcal{F}$ is a symbol that stands for the formulas that are being described by the rule, $f_1, f_2, \ldots, f_n$ are sub-rules, and the interpretation of the whole rule is that a formula $\mathcal{F}$ can be obtained via any of the sub-rules f_1 to f_n. The sub-rules themselves take the form of basic formulas (using only the symbols in the alphabet), with the caveat that they are also allowed to contain the symbol $\mathcal{F}$ - in that case, they are *recursive*, meaning that the $\mathcal{F}$ in them stands for any other formula that is described by the same rule.

In a formal system, the grammar is a way of specifying which formulas are *well-formed* and which are not. A formula is said to be *well-formed* if it can be constructed using the grammar of the system. Consider, for example, the alphabet given by the set of digits mentioned above. Then, a possible grammar is:

$$M ::= 1 \mid 12 \mid 60$$

In this case, M is the general notation for a well-formed formula. The rule is read as: 1 is a well-formed formula, 12 is a well-formed formula, 60 is a well-formed formula and nothing else is.

The rule above does not look incredibly useful. However, as mentioned, sub-rules can also be recursive, for example:

$$M ::= 1 \mid M0$$

This is to be read like this: 1 is a well-formed formula and any formula obtained by appending a 0 to a well-formed formula is also well-formed. Thus, we could say that the well-formed formulas generated by this rule represent all of the natural powers of 10: 1, 10, 100 and so on. In contrast, 123 is *not* well-formed here, because we have no way of constructing it with our grammar.

Given a formula, the situation might not be as clear as in the example above: there has to be some precise decision procedure in order to establish whether the formula can be constructed with the existing rules or not. When talking about *well-formed* formulas from now on, we will assume the existence of such a procedure.

Grammars give us a way to talk about only a subset of all of the possible formulas given by the alphabet. Typically, we do not care about the others, as a formal system is mostly concerned with formulas that are well-formed: anything else is invalid and can not be reasoned about, just as if we were to ask what the result of the mathematical expression $01)(?$ was.

2.1.3 Derivation Rules

For the rest of this report, we will refer to formulas as *terms*.

We have seen how to construct well-formed terms in a formal system. *Derivation rules* (also called *reduction rules*) provide us with a way of manipulating these terms, effectively by replacing one term with another. This is useful if we want our formal system to be able to express something about the workings of the domain that we are trying to model.

Reduction rules specify how term rewriting can take place. They have the form

$$\textit{LHS} \rightarrow \textit{RHS}$$

where *LHS* and *RHS* take the form of sub-rules (similar in appearance to the ones found in the grammar rules). A well-formed term is said to *match* the *LHS* of a rule if, were we to have the *LHS* as a sub-rule in our grammar, we could use it to construct the term. The *RHS* represents what we can replace a term that matches the *LHS* with - importantly, if the *LHS* is recursive (which means that it contains a symbol that stands for another well-formed term), then that symbol is ”captured” and can also appear on the *RHS*. Wherever the *LHS* is matched, such symbols on the *LHS* and *RHS* stand for the same term (whose exact form will be determined by actually constructing the term to be reduced using the *LHS*). Consider the following example:

Let our alphabet be $\{e, o, t\}$ with the following grammar:

$$M, N ::= e \mid o \mid M\ t\ N$$

M and N both stand for well-formed terms in the rules above: we use two letters in order to not give the impression that M and N have to be the same in the rule $M\ t\ N$.

Then, take the reduction rules:

$$\begin{array}{lll} e\ t\ M & \rightarrow e & \textit{(even)} \\ o\ t\ M & \rightarrow M & \textit{(odd)} \end{array}$$

If we have a (well-formed) term that matches the left hand side of any of the reduction rules, we are allowed to replace it with the right hand side of the rule. This process is called *reduction*. Below are a few examples of valid reductions:

$$e\ t\ (o\ t\ o) \xrightarrow{\text{(even)}} e$$

$$e\ t\ (o\ t\ o) \xrightarrow{\text{(odd)}} e\ t\ o \xrightarrow{\text{(even)}} e$$

$$o\ t\ (o\ t\ o) \xrightarrow{\text{(odd)}} o\ t\ o \xrightarrow{\text{(odd)}} o$$

The rule that we have used is written above the arrow for each step. Note the use of brackets to remove the ambiguity - without them, the term $e\ t\ (o\ t\ o)$ could be parsed in two different ways. Also note that, starting with that term, we were able to apply two different derivation rules (the first two reductions above). This means that, in this case, we also allow reductions for *subterms* that appear as part of the term on the left hand side - we can reduce the whole term using the (*even*) rule, or first choose to reduce the $(o\ t\ o)$ subterm using the (*odd*) rule. In general, a formal system often specifies some *reduction strategies*, which make clear how reduction is allowed to happen in any situation.

Note that, if we try to reduce the terms as much as we can, we reach a point where we can no longer reduce them. This final form that we reach is called the *normal form*. In general, a term is said to be in *normal form* if we can not reduce it any further using the given reduction rules.

2.1.4 Significance

We have already mentioned that formal systems are tools used to model certain domains. As such, they should have a connection with the domain they represent and they should provide an abstraction that facilitates reasoning about and proving properties of the domain.

The last example we have shown models the parity of integers and what happens when multiplying numbers of various parities. The reduction rules express that multiplying an even number with anything yields an even number, and multiplying an odd number with anything does not change its parity.

We are now going to present a series of formal systems that are central to this project - however, these systems model something slightly more interesting: *computation*.

2.2 λ-Calculus

The λ-Calculus [6] is a formal system developed in the 1930's by Alonzo Church with the intention to model the mathematical notion of *computable* functions. It is remarkable because of its elegance and simplicity and it has become the basis for many functional programming languages such as Haskell and ML. The mechanisms of the λ-Calculus, as presented in [6], are described below.

2.2.1 λ-terms

In the λ-Calculus, terms are obtained from a set of *term variables* - $\{x, y, z, ...\}$ - and two operations: *abstraction* and *application*. Thus:

$$M, N ::= \underset{\textit{variable}}{x} \mid \underset{\textit{abstraction}}{(\lambda x.M)} \mid \underset{\textit{application}}{(M \cdot N)}$$

From a mathematical/computational point of view, these rules can be interpreted as a λ-term being one of the following:

- a variable;
- a function that takes one input parameter and acts on it, producing another term;
- an application of a term to another term, where M can be viewed as a function and N as its argument.

Below are some examples of valid λ-terms:

$$\lambda x.\ x$$
$$\lambda xyz.\ xy(xz)$$
$$(\lambda xy.\ xy)(\lambda z.\ z)$$

There are some notational conveniencies visible above that we are going to follow through this report. Firstly, we omit the dot between the terms in an application, as the dot is the only possible operation between two terms. Secondly, we omit the leftmost, outermost brackets and only use them to remove ambiguity in certain cases (application is *left associative*): for example, the $xy(xz)$ above stands for $((xy)(xz))$, and yzx would stand for $(yz)x$, not $y(zx)$. Lastly, we contract consecutive abstractions under a single lambda: $\lambda xyz.\ ...$ stands for $\lambda x.\ (\lambda y.\ (\lambda z.\ ...))$.

Note that abstractions in the λ-Calculus are anonymous, in that they do not possess a name - they are identified only by their input parameters and their effects. For example (note that numbers and mathematical symbols are not part of the λ-Calculus by definition), if we were to represent the mathematical function *double*, defined as *double*$(x) = \ 2 * x$, we could do so using the λ-term $\lambda x.\ 2 * x$. Moreover, the call *double(3)* would be represented as $(\lambda x.\ 2 * x)3$, which we could argue should "evaluate" to 6 - we are going to make this notion more precise in the coming paragraphs.

First, we define the notions of *free* and *bound* variables. For a term M, we denote its bound variables by $bv(M)$. Thus:

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

The free variables are denoted by $fv(M)$. Then:

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

Intuitively, a variable is *bound* if it appears anywhere under the influence of some lambda which abstracts over it, and a variable is *free* if it appears anywhere and it is *not* under the influence of any lambda that abstracts over it. A variable can appear in multiple places, so it can be both bound and free in the same term.

2.2.2 Reduction Rules

We are now going to give the reduction rules of the system. The most important rule is that $(\lambda x.\ M)N$ should reduce to the body of M in which all occurences of x are replaced by N, which essentially works the same way as a mathematical function. In the λ-Calculus, this is expressed through the notion of *substitution*. Denote by $M[N/x]$ the term obtained from M by replacing all occurrences of x with N. This is defined inductively, thus:

$$\begin{aligned} x[N/x] &= N & \\ y[N/x] &= y & \text{if } (x \neq y) \\ (MP)[N/x] &= M[N/x]P[N/x] & \\ (\lambda x.M)[N/x] &= \lambda x.M & \\ (\lambda y.M)[N/x] &= \lambda y.(M[N/x]) & \text{if } (x \neq y) \end{aligned}$$

However, the last rule might pose a problem: for example, using it, we have that $(\lambda y.yx)[y/x] = \lambda y.yy$. The y that was originally free in N has become bound after the substitution - this is called a *variable capture* and is considered bad, as it changes the *meaning* of N. In order to avoid this problem, we assume *Barendregt's Convention*: free variable names and bound variable names are always different. This implies that, in the last rule above, since $y \in bv(\lambda y.M)$, y can not also appear free in N, which makes the substitution safe. However (as we will see), Barendregt's Convention is not necessarily preserved by reduction and we may still end up in a situation similar to the example we just gave (for example, when reducing $(\lambda xy.xy)(\lambda xy.xy)$, the convention holds at the start, but after two reduction steps it does not hold anymore). In these situations, to enforce Barendregt's Convention, the notion of α-conversion is employed: informally, it means that we are free, at any point, to rename the binding occurrence of variable in a term (and all of the occurrences that it binds) to a *fresh* variable, without changing the meaning of the term. This is necessary in order to proceed with reduction - in our example above, we would first rename $\lambda y.yx$ to $\lambda z.zx$, after which the substitution will happen correctly: $(\lambda z.zx)[y/x] = \lambda z.zy$. We assume that α-conversion, whenever necessary, happens silently.

We now define the notion of reduction for the λ-Calculus - it is called β-reduction and it is denoted by $\rightarrow_\beta$. There is one main rule:

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

Additionally, there are 3 other inductive rules, which specify that we allow reduction for subterms and under abstractions:

$$M \rightarrow_\beta N \Rightarrow \begin{cases} PM \rightarrow_\beta PN \\ MP \rightarrow_\beta NP \\ \lambda x.M \rightarrow_\beta \lambda x.N \end{cases}$$

The reflexive, transitive closure of $\to_\beta$ is denoted by $\to_\beta^*$.

These rules model how computation works in the λ-Calculus - essentially, progress happens by applying a function to an argument. Below are some examples of valid reductions:

$$(\lambda x.\ x)y \to_\beta y$$
$$(\lambda xy.\ x)y \to_\beta \lambda z.\ y$$
$$(\lambda x.\ xx)(\lambda x.\ xx) \to_\beta (\lambda x.\ xx)(\lambda x.\ xx) \to_\beta ...$$
$$(\lambda xy.\ xy)((\lambda x.\ x)(\lambda z.\ z)) \to_\beta (\lambda xy.\ xy)(\lambda z.\ z) \to_\beta \lambda y.\ (\lambda z.\ z)y \to_\beta \lambda y.\ y$$

As mentioned before, a term which we can no longer reduce is said to be in *normal form*. We can already notice in the third reduction above that, in this calculus, not all terms can be brought to a normal form - the interpretation of this could be that there exist computer programs that never terminate.

2.2.3 Types

A very useful extension to the λ-Calculus comes in the form of typing information: that is, we may try to assign a type to each term and deduce something about those terms that can be typed, and those that can not.

Such information is useful because it provides an *abstraction* of a program by distilling all of the terms to the type level, which is less detailed and easier to reason about: the focus is on the kind of input and output of the terms. From a programming perspective, type information is also important for a compiler - if it has information about the types of functions and variables, it can give more detailed error messages at compile-time, for example to warn the programmer about a function designed for integers that is mistakenly applied to a string. It therefore also provides a way to sanity-check a program before it is actually run.

We present here the type assignment system for the λ-Calculus, as given in [6].

The set of types, ranged over by $A, B, C, ...$, is defined inductively over a set of *type variables* $\{\varphi, \tau, ...\}$ via the following rule:

$$A, B ::= \varphi \ | \ (A \to B)$$

The main feature of this system is that λ-abstractions will be assigned a type of the form $A \to B$, adhering to the fact that we think about them as being functions (that take a parameter of type A and return something of type B).

A *statement* is an expression of the form $M : A$, which is read as ”the λ-term M has type A”.

A *context*, Γ, is a set of such statements that concerns types only for distinct, individual variables, such as x or y. We write $x \in \Gamma$ if there exists some A such that

$x : A \in \Gamma$, and $x \notin \Gamma$ otherwise. We also write $\Gamma, x : A$ for $\Gamma \cup \{x : A\}$. Contexts will be used for tracking the types of free variables while typing a term.

As a notational conveniency (arrow type chains are *right associative*), when writing types, we omit the rightmost, outermost brackets, such that the type $\varphi_1 \to \varphi_2 \to \varphi_3$ stands for $(\varphi_1 \to (\varphi_2 \to \varphi_3))$.

We now give Curry's type assignment system, which is defined using three derivation rules:

$$\frac{}{\Gamma, x : A \vdash_c x : A}\,(Ax) \qquad \frac{\Gamma, x : A \vdash_c M : B \quad x \notin \Gamma}{\Gamma \vdash_c \lambda x.M : A \to B}\,(\to I) \qquad \frac{\Gamma \vdash_c M : A \to B \quad \Gamma \vdash_c N : A}{\Gamma \vdash_c MN : B}\,(\to E)$$

The rules take the form of a natural deduction system, their meaning being: if the statements above the line (the *premisses*) hold, then we can deduce the statement below the line (the *conclusion*). When we write $\Gamma \vdash_c M : A$, we mean that there exists some derivation constructed using the rules above *that has that statement as the conclusion* - this derivation proves that, in the context Γ, we can assign the type A to M.

To illustrate the rules, we give below an example derivation for the term $\lambda xy.xy$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \varphi_1 \to \varphi_2, y : \varphi_1 \vdash_c x : \varphi_1 \to \varphi_2}\,(Ax) \quad \dfrac{}{x : \varphi_1 \to \varphi_2, y : \varphi_1 \vdash_c y : \varphi_1}\,(Ax)}{x : \varphi_1 \to \varphi_2, y : \varphi_1 \vdash_c xy : \varphi_2}\,(\to E)}{x : \varphi_1 \to \varphi_2 \vdash_c \lambda y.xy : \varphi_1 \to \varphi_2}\,(\to I)}{\varnothing \vdash_c \lambda xy.xy : (\varphi_1 \to \varphi_2) \to \varphi_1 \to \varphi_2}\,(\to I)$$

Note that this is not the only type that we could have assigned to this term; in fact, replacing all of the occurrences of any of the type variables in the derivation by something else also yields a correct derivation. The derivation is by no means an *algorithm* for finding all of the types we could assign to a term: it is merely an illustrative justification that a particular type works.

There are also terms that *can not* be typed in the λ-Calculus. A very simple example is provided in [6]: the self-application xx. We would like a derivation for $\Gamma \vdash_c xx : B$, for some B. Looking at our derivation rules, this could only have come from an application of $(\to E)$, which means that our context would have to assign the type $A \to B$ (for some A) to the left hand side x, and the type A to the right hand side x. However, this would mean that we would either need to have two distinct statements for x in our context, which is not allowed, or we would need to find a solution for $A \to B = A$, which is impossible. Hence the term can not be typed.

This type assignment system enjoys many interesting properties, such as *subject reduction*: if $M \to_\beta^* N$ and there exist a context Γ and a type A such that $\Gamma \vdash_c M : A$, then $\Gamma \vdash_c N : A$.

There have been many extensions brought to the λ-Calculus. We present two of them that are relevant to this project in what follows.

2.3 $\lambda\mu$-Calculus

In this section we present the $\lambda\mu$-Calculus, which is an extension to the λ-Calculus introduced by Parigot in [7]. It is strongly related to a subset of logic called *minimal classical logic*, in the sense that there is a correspondence between natural deductive proofs in minimal classical logic and type derivations for terms in $\lambda\mu$. This is called a *Curry-Howard isomorphism* and, as mentioned by Griffiths in [3], others exist between different calculi and logics as well. In *minimal classical logic*, proofs have one active conclusion and a number of alternative conclusions: this is reflected in $\lambda\mu$ in the form of the typing judgements. A judgement will have the form $\Gamma : M : A \mid \Delta$, where $M : A$ corresponds to the active conclusion and Δ (consisting of pairs of *names* and types) holds the alternative ones. To model changing between conclusions (*activation* and *deactivation*), Parigot introduces new constructs: *named terms* (or *commands*), $[\alpha]M$ (where α is the name), and μ-abstractions, $\mu\alpha.[\beta]M$. Through their reduction rules, which, as we will see, are significantly different from normal λ-Calculus ones, they capture the "algorithmic meaning" of proofs in minimal classical logic and, in a sense, they model *control flow manipulation*. We present the version of $\lambda\mu$ given by Griffiths in [3]:

2.3.1 Terms

Terms in $\lambda\mu$ are defined as for λ, with an additional construct:

$$M, N ::= ... \mid (\mu\alpha.[\beta]M)$$

The novelty is called the μ-abstraction. Constructions of the form $[\beta]M$ are usually called *commands* and are treated as terms when convenient. α and β are called *names*, and the notion of *bound* variables and names is defined similarly to λ:

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

$$\begin{aligned} bn(x) &= \varnothing \\ bn(\lambda x.\ M) &= bn(M) \\ bn(MN) &= bn(M)\ \cup\ bn(N) \\ bn(\mu\alpha.[\beta]M) &= bn(M)\ \cup\ \{\alpha\} \end{aligned}$$

Names and variables are called *free* if they appear in a place where they are not bound by any abstraction.

$$\begin{array}{lcr}
\textit{logical } (\beta): & (\lambda x.M)N \to_{\beta\mu} M[N/x] & \\
\textit{structural } (\mu): & (\mu\delta.C)N \to_{\beta\mu} \mu\gamma.C\{N \cdot \gamma/\delta\} & (\gamma \textit{ fresh}) \\
(\textit{erasing}): & \mu\delta.[\delta]M \to_{\beta\mu} M & (\delta \notin M) \\
(\textit{renaming}): & [\gamma]\mu\delta.C \to_{\beta\mu} C\{\gamma/\delta\} &
\end{array}$$

$$M \to_{\beta\mu} N \Rightarrow \begin{cases} MP \to_{\beta\mu} NP \\ \mu\alpha.[\beta]M \to_{\beta\mu} \mu\alpha.[\beta]N \end{cases}$$

Figure 2.1: $\lambda\mu$ call-by-name reduction rules

2.3.2 Reduction Rules

The $\lambda\mu$-Calculus introduces new reduction rules to deal with the new construct. First, we informally define a new kind of substitution: the *naming substitution*. It can be denoted by $M\{\gamma/\alpha\}$, which stands for the term obtained from M by replacing all commands of the form $[\alpha]N$ with $[\gamma]N$. Moreover, we denote by $M\{P\cdot\gamma/\alpha\}$ the term obtained from M by replacing each command of the form $[\alpha]N$ with $[\gamma]NP$. This last operation can be seen as a way of passing new information, P, to all the commands associated with the name α and then giving them a new name, γ. Formally, this is defined inductively:

$$\begin{aligned}
x\{P \cdot \gamma/\alpha\} &= x \\
(\lambda x.M)\{P \cdot \gamma/\alpha\} &= \lambda x.(M\{P \cdot \gamma/\alpha\}) \\
(MN)\{P \cdot \gamma/\alpha\} &= M\{P \cdot \gamma/\alpha\}N\{P \cdot \gamma/\alpha\} \\
[\alpha]M\{P \cdot \gamma/\alpha\} &= [\gamma](M\{P \cdot \gamma/\alpha\}P) \\
[\beta]M\{P \cdot \gamma/\alpha\} &= [\beta](M\{P \cdot \gamma/\alpha\}) \qquad (\beta \neq \alpha) \\
(\mu\delta.C)\{P \cdot \gamma/\alpha\} &= \mu\delta.(C\{P \cdot \gamma/\alpha\})
\end{aligned}$$

Term substitution is defined as for the λ-Calculus. With these notations, we can now define the reduction relation $\to_{\beta\mu}$ for $\lambda\mu$, which is shown in figure 2.1.

The reduction strategy we take is called *call-by-name* - note that we only allow reduction for the first subterm in an application, and we do not allow reduction under λ-abstractions.

As before, we define the multi-step reduction relation $\to^*_{\beta\mu}$ as the reflexive, transitive closure of $\to_{\beta\mu}$.

These rules make clear the difference between the two types of abstractions. Unlike λ-abstractions, which express computation via applying functions, μ-abstractions specify *direction*: they direct the incoming information to the relevant channels. As such, they have proven very useful in modelling *control flow manipulation*.

Below is a worked example of a $\lambda\mu$ reduction, also presented in [3]. Note that the notation $\mu\_.M$ is used to denote a name that does not appear free in M, and the $\_$

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ (Ax)$$

$$\frac{\Gamma, x : A \vdash M : B \mid \Delta \quad x \notin \Gamma}{\Gamma \vdash \lambda x.M : A \to B \mid \Delta}\ (\to I) \qquad \frac{\Gamma \vdash M : A \to B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta}\ (\to E)$$

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

Figure 2.2: Type assignment rules for $\lambda\mu$

does not count for binding. Assume that $\alpha \notin M$ and $\alpha \neq \beta$:

$(\mu\alpha.[\beta](\lambda x.\mu\_.[\alpha]x)M)N$
$\to_{\beta\mu}$ $\mu\gamma.(([\beta](\lambda x.\mu\_.[\alpha]x)M)\{N \cdot \gamma/\alpha\})$ (by (μ))
$= \mu\gamma.[\beta](\lambda x.\mu\_.[\gamma]xN)M$ (by def. subst., note $\alpha \notin M$ and $\alpha \neq \beta$)
$\to_{\beta\mu}$ $\mu\gamma.[\beta]\mu\_.[\gamma]MN$ (by (β), reducing under μ-abstraction)
$\to_{\beta\mu}$ $\mu\gamma.([\gamma]MN)\{\beta/\_\}$ (by (*renaming*))
$= \mu\gamma.[\gamma]MN$ (by def. subst. and _)
$\to_{\beta\mu}$ MN (by (*erasing*))

2.3.3 Types

We now present the type assignment system for $\lambda\mu$, as laid out in [1] and [3]. In order to deal with the new constructs, a new notion is introduced: the *naming context*, usually denoted by Δ. It is defined exactly the same as the usual Γ context form the λ-Calculus, but it specifies the types for names instead of variables. In the new system, conclusions will have the form $\Gamma \vdash M : A \mid \Delta$ - the naming context will be used to keep track of the types of the names, enforcing the principle that names should have the same type as the term they are naming. The rules are given in figure 2.2.

It has been proven that there is a direct correspondence (also called a *Curry-Howard isomorphism*) between type derivations in this system and natural deductive proofs in a subset of logic called *minimal classical logic*, however we do not go into details here. We show below an example given in [3] of a type derivation for the term $\lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x)$ (which, in fact, would correspond to a proof of Peirce's Law from logic):

$$\cfrac{\cfrac{\cfrac{\cfrac{}{y : (A \to B) \to A \vdash y : (A \to B) \to A \mid \alpha : A}\ (Ax) \quad \cfrac{\cfrac{\cfrac{}{y : (A \to B) \to A, x : A \vdash x : A \mid \beta : B, \alpha : A}\ (Ax)}{y : (A \to B) \to A, x : A \vdash \mu\beta.[\alpha]x : B \mid \alpha : A}\ (\mu_{\alpha\beta})}{y : (A \to B) \to A \vdash \lambda x.\mu\beta.[\alpha]x : A \to B \mid \alpha : A}\ (\to I)}{y : (A \to B) \to A \vdash y(\lambda x.\mu\beta.[\alpha]x) : A \mid \alpha : A}\ (\to E)}{y : (A \to B) \to A \vdash \mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : A \mid \varnothing}\ (\mu_\alpha)}{\varnothing \vdash \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : ((A \to B) \to A) \to A \mid \varnothing}\ (\to I)$$

2.3.4 Significance

Due to its reduction rules, the $\lambda\mu$-Calculus proves to be useful for modelling control flow manipulation in a program, including exception handling. The next section details this link.

2.4 λ^{try}-Calculus

In this section we present the λ^{try}-Calculus for *recoverable exceptions* as introduced by van Bakel in [1].

λ^{try} is an extension to the λ-Calculus that aims to model traditional exception handling in a functional style, introducing a new way of discriminating exceptions: *by name*. "Throwing" an exception is represented by the term *throw* $n(M)$, which stands for throwing M to a *named handler* called n. "Catching" is modeled via the term *try* $M;\ \overline{\textit{catch}\ n_i(x) = N_i}$, which defines a series of named handlers to be taken into account during the "execution" of M: if a throw to one of the handlers occurs during said execution, then any remaining execution in the *try* block is disregarded and replaced by the execution of the corresponding handler, which uses the information passed over by the throw. The new terms and their operation are reminiscent of imperative languages such as Java, with the mention that exceptions are discriminated by name, rather than by type. We give the main features of λ^{try} below, as introduced by van Bakel in [1].

2.4.1 Terms

The set of *pre-terms* in λ^{try} is defined as follows:

$$\begin{aligned} \textit{Catch\_Block} &::= \textit{catch}\ m(x) = M \mid \textit{Catch\_Block}; \textit{catch}\ n(x) = N \\ M, N &::= V \mid MN \mid \textit{try}\ M;\ \textit{Catch\_Block} \mid \textit{throw}\ n(M) \\ V &::= x \mid \lambda x.M \end{aligned}$$

We call the n in *catch* $n(x) = M$ a *declared name* and we will write $\overline{\textit{catch}\ n_i(x) = N_i}$ for the catch block

$$\textit{catch}\ n_1(x) = N_1;\ \textit{catch}\ n_2(x) = N_2;\ ...;\ \textit{catch}\ n_n(x) = N_n.$$

Then, *terms* are defined to be pre-terms that satisfy two additional conditions:

1. In $\overline{\textit{catch}\ n_i(x) = M_i}$, the names n_i do not occur inside the exception handler M_j (for any i and j from 1 to n) and all of the declared names n_1, ..., n_n are distinct.

2. For each *throw* $n_l(N)$ that occurs in M in the term *try* $M;\ \overline{\textit{catch}\ n_i(x) = N_i}$, none of the names n_i occur in N.

$$\begin{array}{cc}
(\beta): & (\lambda x.M)N \to M[N/x] \\
(\textit{throw}): & (\textit{throw}\ n(N))M \to \textit{throw}\ n(N) \\
(\textit{try-throw}): & \textit{try throw}\ n_l(N);\ \overline{\textit{Catch_Block}};\ \textit{catch}\ n_l(x) = M_l \to M_l[N/x] \\
(\textit{try-normal}): & \textit{try}\ N;\ \overline{\textit{catch}\ n_i(x) = M_i} \to N \quad (\overline{n_i \notin N})
\end{array}$$

$$M \to N \Rightarrow \begin{cases} MP \to NP \\ \textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = M_i} \to \textit{try}\ N;\ \overline{\textit{catch}\ n_i(x) = M_i} \end{cases}$$

Figure 2.3: λ^{try} call-by-name reduction rules

Note that the new constructs resemble exception-handling syntax that one might expect to find in a programming language such as C++ or Java. However, in this case, exceptions are discriminated by name: a throw happens to a named handler, which in turn expects an argument from the throw and does something with it.

2.4.2 Reduction Rules

The original paper [1] presents various reduction strategies. We give here the rules for *call-by-name* reduction on λ^{try}: $\to^N_{TRY}$. They can be found in figure 2.3.

The first main rule is the basic one from the λ-Calculus. The other 3 main rules express the exception handling mechanisms that one would expect: the code after a throw does not execute (hence the throw consumes the terms after it), if a throw to a handler named n_l happens in a try block, then the corresponding exception handler (M_l) is invoked, and if no relevant throw happens inside a try block, then the try block can be ignored. Note that reduction under λ-abstractions is not allowed: this is because, from a computational point of view, it would mean allowing an exception to be raised just because it appears in a function definition, disregarding whether program execution has actually led to the exception (and additionally, as shown by van Bakel in [1], subject reduction would fail instantly).

2.4.3 Types

We give here the type assignment system introduced by van Bakel in [1] (see figure 2.4). The notions of *context* Γ and *naming context* Δ are defined in the same way as for $\lambda\mu$. A characteristic of this system is that, for successfully typing a *try* term, it demands that all of the exception handlers return the same type as the one of the main term: this makes exceptions *recoverable*, in the sense that, if the *try* term has a type, a computer program will be able to rely on it regardless of whether any exceptions were raised or not. If they have, they will have been handled correctly.

The first 3 rules are the same as in the normal λ-Calculus. The rule ($\textit{throw}$) allows us to assign any type to a *throw* term, *provided that there exists a suitable exception handler of the correct type*. The rule ($\textit{try}$) states that a *try* term can be typed only if

$$\frac{}{\Gamma, x : A \vdash x : A \mid \Delta}\ (Ax) \qquad \frac{\Gamma, x : A \vdash M : B \mid \Delta \quad x \notin \Gamma}{\Gamma \vdash \lambda x.M : A \to B \mid \Delta}\ (\to I)$$

$$\frac{\Gamma \vdash M : A \to B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta}\ (\to E) \qquad \frac{\Gamma \vdash N : A \mid n : A \to B, \Delta}{\Gamma \vdash \textit{throw}\ n(N) : C \mid n : A \to B, \Delta}\ (throw)$$

$$\frac{\Gamma \vdash M : B \mid \overline{n_i : A_i \to B}, \Delta \quad \Gamma, x : A_i \vdash N_i : B \mid \Delta\ (\forall i \in \{1, ..., n\}) \quad (\overline{n_i \notin \Delta})}{\Gamma \vdash \textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i} : B \mid \Delta}\ (try)$$

Figure 2.4: λ^{try} type assignment rules

all of the exception handlers return the same type as that of the main term.

This type assignment system has the property of subject reduction.

2.4.4 Representation in $\lambda\mu$

We now give the translation of the λ^{try}-Calculus into $\lambda\mu$ as presented in [1]. This will prove useful in the coming sections. It is based on two observations (which were already made by Crolard in [8]):

1. Throwing M to handler n can be represented by the $\lambda\mu$-term $\mu\_.[n]M$ $(n \notin M)$, as it correctly models the reduction rule (*throw*) (the consuming of contexts): $(\mu\_.[n]M)N \to \mu\_.[n]M$, as $\_$ does not occur in M.
2. Catching on (and giving scope to) the name n can be represented by $\mu n.[n]$, which, in combination with the representation of the throw, reduces to M: $\mu n.[n]\mu\_.[n]M \xrightarrow{(renaming)} \mu n.[n]M \xrightarrow{(erasing)} M$ (as $n \notin M$).

These observations outline a possible interpretation into $\lambda\mu$. However, due to the nature of $\lambda\mu$ and the observations above, when interpreting the term *try* $M;\ \overline{\textit{catch}\ n_i(x) = N_i}$, we actually have to bring all of the exception handlers *inside* the representation of M. This is achieved by introducing a new variable c_n for each handler *catch* $n(x) = M$, which is dealt with by substitution. The interpretation $\llbracket\cdot\rrbracket_{\lambda\mu}$ is then given below:

$$\begin{aligned}
\llbracket x \rrbracket_{\lambda\mu} &\triangleq x \\
\llbracket \lambda x.M \rrbracket_{\lambda\mu} &\triangleq \lambda x.\llbracket M \rrbracket_{\lambda\mu} \\
\llbracket MN \rrbracket_{\lambda\mu} &\triangleq \llbracket M \rrbracket_{\lambda\mu}\llbracket N \rrbracket_{\lambda\mu} \\
\llbracket \textit{throw}\ n(M) \rrbracket_{\lambda\mu} &\triangleq \mu_.[n]c_n\llbracket M \rrbracket_{\lambda\mu} \\
\llbracket \textit{try}\ M;\ \textit{catch}\ n(x) = N \rrbracket_{\lambda\mu} &\triangleq (\mu n.[n]\llbracket M \rrbracket_{\lambda\mu})[\lambda x.\llbracket N \rrbracket_{\lambda\mu}/c_n] \\
\llbracket \textit{try}\ M;\ \textit{Catch_Block};\ \textit{catch}\ n(x) = N \rrbracket_{\lambda\mu} &\triangleq (\mu n.[n]\llbracket M;\ \textit{Catch_Block} \rrbracket_{\lambda\mu})[\lambda x.\llbracket N \rrbracket_{\lambda\mu}/c_n]
\end{aligned}$$

Thus, wherever there was a *throw* in the original term, there will be a special variable (like c_n) in the translation, which, after substitution, will become the translation of the corresponding handler. The two previous observations then kick in, with terms reducing as desired. This is illustrated through the following example:

$[\![$*try throw* $n(z)$; *catch* $n(x) = x]\!]_{\lambda\mu}$
$= (\mu n.[n]\mu\_.[n]c_n z)[\lambda x.x/c_n]$ (by def. trans.)
$= \mu n.[n]\mu\_.[n](\lambda x.x)z$ (by def. subst.)

Note how, through the use of c_n and substitution, the interpretation of the handler n has been brought *inside* the interpretation of *throw*; $n(z)$. Then, exactly as in observation 2 above, the above reduces to $(\lambda x.x)z$ via the rules (*renaming*) and (*erasing*), and then to z by rule (β).

This shows that λ^{try} is expressible in $\lambda\mu$, and the interpretation above also enjoys two important properties (as shown by van Bakel in [1]): it preserves reduction and it preserves assignable types (under the type assignment system shown in the previous section).

Before we explore the previous work that was done by Fisher in [4], it is necessary to present some useful Haskell notions.

2.5 Haskell Monads

Monads are structures that originate from the domain of *category theory* - however, they turned out to have a wide range of applications in functional programming languages such as Haskell. We give first a formal definition of the Haskell Monad class, followed by a discussion of their benefits via some examples.

2.5.1 Haskell Types

Haskell natively supports types such as `Int`, `Char` or `List[String]` (representing a list of strings). In addition to these, we also have the possibility to define our own data types, like this:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Here, `Day` is a type and `Mon`, ..., `Sun` are values of that type, just like `6` is a value of type `Int`. This is an example of *enumerated type*. Data types can also be parameterized with type variables, like in the following example:

```
data Maybe a = Nothing | Just a
```

This is Haskell's built-in `Maybe` type, which is very useful in computations where we might want to account for the fact that something "went wrong" (such as a table lookup that might return `Nothing` if no corresponding entry was found, or `Just a` if an entry - of type `a` - was found). An equivalent type in Java would be `Optional<T>`. In the above definition:

1. `Maybe` is called a *type constructor*: it takes another type, `a`, and it generates a new type: `Maybe a` - for example, `Maybe Int`;

2. `Nothing` and `Just` are called *data constructors*: they construct *values* of type `Maybe a`, possibly taking some arguments. `Nothing` does not take any arguments - it is a value of type `Maybe a`, for any `a`. `Just`, however, takes one argument (of type `a`, for some `a`): thus, `Just 3` would be a value of type `Maybe Int`.

A powerful feature of Haskell are *type classes*. A *type class* is the collection of types over which certain functions are defined - an equivalent concept from OOP would be interfaces. A class declaration might look like this:

```
class Eq a where
  (==) :: a -> a -> Bool
```

This could be the class of types that have an equality method, ==, defined over them. The method takes two values of the same type and returns a boolean value representing the result of the comparison. We might want to add a notion of "comparability" to values of the type `Day` that we have defined above. Then, we could simply add our type to this class by implementing an "equals" method:

```
instance Eq Day where
  Mon == Mon = True
  Tue == Tue = True
  ...
  Sun == Sun = True
  _ == _     = False
```

This specifies the equality of days in the way we would expect it to; now, we can write `x == y` if `x` and `y` have type `Day`, and, additionally, the type `Day` could be used in any place where Haskell expects something that is a member of `Eq` (meaning that values of that type can be tested for equality).

2.5.2 Definition

Let `a` and `b` be two types. A *monad*, then, is defined as a triple consisting of:

1. A *type constructor*, `M`, which constructs the *monadic type* `M a`;

2. A function, called `return`, that takes a value of type `a` and embeds it in the monad, yielding a *monadic value*;

3. A *binding operator*, usually denoted by >>=, which describes how to obtain a new monadic value from an existing one and a function that acts on an unwrapped value.

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

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail s = error s
```

Figure 2.5: Monad type class in Haskell

In Haskell, monads are represented by the type class given in figure 2.5.

Note that, in the definition, `m` is not a type, but a type constructor. Thus, `m a` should be viewed as a type, just like `Maybe Int`.

The definition says that a type constructor `m` is a *monad* if there are functions with the corresponding type signatures implemented for it. Essentially, a monad can be viewed as a piece of computation wrapped around a value (of type `a`, say) - the binding operator, then, specifies how two such pieces of computation are to be sequenced (the value wrapped in the first computation is *bound* to the second one, which might use it). The meaning of these functions is as follows:

1. `return`: the purpose of this function is to construct monadic values from ”normal” values; in the case of `Maybe`, this is achieved by mapping `x` to `Just x`;

2. `>>=`: this is a function that takes two arguments: the first one is a monadic value, and the second one is a function that takes a normal, unwrapped value and generates a new monadic value from it. The binding operator then specifies how the two of them are combined: it might choose to ignore the second argument completely, or it might try to unwrap the value contained in the first argument and then pass it to the second one, which would use it in some way. In any case, the operator must return a new monadic value that represents the effect of combining the two pieces of computation;

3. `>>`: this operator does the same as the one above, except that we know for sure that the second argument does not use the value wrapped in the first one, hence it is not included in the type signature anymore (see the default implementation in the class definition);

4. `fail`: the purpose of this function is to deal with pattern matching errors. This is because Haskell provides the following syntactic sugar (do-notation):

```haskell
do x <- m1; m2
```

is equivalent to

```
m1 >>= \x -> m2
```

However, the x on the left hand side is also allowed to be a pattern: if pattern matching fails for the value returned by the first computation, then the result is a call to `fail` with some meaningful error message - by default, `fail` itself calls `error`, but it can be made to do more useful things.

We illustrate the concept further using `Maybe`, which is a built-in monad in Haskell. In its case, the instance statement is:

```
instance Monad Maybe where
  return = Just

  Nothing >>= _ = Nothing
  Just x >>= f  = f x

  fail _ = Nothing
```

The meaning of >>= for `Maybe` is now clear: if the first argument (a monadic value) is `Nothing`, then it returns `Nothing` regardless of the second argument: there is no unwrapped value to bind to it! If, however, the first argument is a `Just`, then we can unwrap the value from it and feed it to the second argument (a function - see type signature in `Monad` class definition), which uses it to return a new piece of computation (monadic value). Furthermore, note how `fail` in this case just returns `Nothing` instead of throwing an error: this might be because we want to signal the fact that something has gone wrong like this, instead of terminating the program on the spot.

2.5.3 Benefits

Consider the following case: we want to implement a function `mult` that multiplies together two `Maybe Int` values, returning `Nothing` if any of them is `Nothing`, and `Just result` otherwise. This is the direct way of doing it:

```
mult ma mb =
    case ma of
        Nothing -> Nothing
        Just a  -> case mb of
            Nothing -> Nothing
            Just b  -> Just (a * b)
```

And here is a way of doing it with monads:

```
mult ma mb = ma >>= (\a -> mb >>= (b -> return (a * b)))
```

Or even better, using the do-notation:

```
mult ma mb = do
    a <- ma
    b <- mb
    return (a * b)
```

Thus, the role of the monad is to make the code clearer and more readable by providing a pipeline: the initial value is passed from one step to another, potentially changing its form, and the bind operator describes what happens to it between two steps: in this case, the handling of `Nothing` happens in the monad, which does not apply any future computation to it, but returns it straight away (if any of `ma` or `mb` were to be `Nothing`).

Therefore, monads provide a way to abstract away computational details regarding computations on a specific type (in the case above, `Maybe`, where `Nothing` entails `Nothing`); they incorporate various information that is passed down along the computation. Furthermore, they can also isolate various side effects of the computations, which is useful for modelling things like input and output in a purely functional language (note that IO in Haskell is done using monads).

2.6 Existing Haskell Implementation

We are now going to show how Fisher [4] produced an implementation of λ^{try} in Haskell.

The implementation is based on the interpretation of λ^{try} in $\lambda\mu$ (which we have already shown) which is then composed with an interpretation of $\lambda\mu$ into *CDC (Calculus of Delimited Continuations)* - a different calculus which we are going to explain next and for which there already existed a Haskell library at the time [9].

2.6.1 Calculus of Delimited Continuations

We first explain the notion of *continuation*, as shown in [3]. Essentially, a continuation represents the remaining reduction steps to be applied to a term after it has been reduced. Consider the following example, where $M \to_\beta^* M'$:

(Compound term)	MN
(Decomposition)	$M\,\Box N$
(Reduction)	$M'\,\Box N$
(Refill hole)	$M'N$

Here, M is called a *dominant term* and $\Box N$ is called a *context* or a *continuation*, because it represents the future computation of M: what will happen to it after it has been reduced. Now, if we consider a more complex term, we may have multiple continuations:

$$\begin{array}{lll} (MN)P & & \\ MN & \Box P & \\ M & \Box N & \Box P \end{array}$$

Here, continuations are maintained separately, in a stack. When M has reduced, the result is returned to the first continuation on the stack, then the result of that is returned to the second continuation, and so on.

We could have also combined all of the continuations in a single continuation: $(\Box N)P$.

If a programming language has control operators that allow the manipulation of individual portions of the continuation stack, then they are called *delimited continuations*. If, however, only manipulation of the entire remaining continuation is allowed (as in the "combined" continuation above), the continuations are called *undelimited*.

In order to properly express these concepts, Peyton Jones *et al.* have introduced what they call *a monadic framework for delimited continuations* in [9]. This framework is presented in the context of the λ-calculus, and Fisher [4] refers to it as the *Calculus of Delimited Continuations*, or CDC for short, which we will do as well. We give below its syntax as presented in [9]. The terms are called *expressions* and $x, y, ...$ range over a set of variables.

$$\begin{array}{rl} e ::= x \mid \lambda x.e \mid e\, e' \mid \textit{newPrompt} \mid \textit{pushPrompt}\ e\ e \\ \mid \textit{withSubCont}\ e\ e \mid \textit{pushSubCont}\ e\ e \end{array}$$

In short, the new constructs allow for manipulation of the continuation stack: they model pushing certain *prompts* onto it, and retrieving portions of it until a certain prompt is found. We do not give a complete operational semantics here - for our purpose, it suffices to keep in mind that the reduction rules borrow some notation from Haskell, in which CDC was implemented in the form of a library [9].

2.6.2 $\lambda\mu$ to CDC

The final piece that we need is an interpretation of $\lambda\mu$ into CDC. We give below the one introduced by Fisher in [4]. We will abbreviate the new constructs in CDC by *NP*, *PP*, *WSC* and *PSC*, respectively.

As shown by Fisher, in order to run a full $\lambda\mu$ program, say M, in CDC, it is assumed that there exists a global prompt, say P_0, that has already been pushed onto the stack. Therefore, the "initialization" is given by

$$(\lambda P_0.\ \textit{PP}\ P_0\ \ [\![M]\!])\ \textit{NP}$$

where:

$$[\![x]\!] \triangleq x$$

$$
\begin{aligned}
[\![\lambda x.M]\!] &\triangleq \lambda x.[\![M]\!] \\
[\![MN]\!] &\triangleq [\![M]\!][\![N]\!] \\
[\![\mu\alpha.M]\!] &\triangleq \mathit{WSC}\ P_0\ (\lambda\alpha.\ \mathit{PP}\ P_0\ \ [\![M]\!]) \\
[\![[\beta]M]\!] &\triangleq \mathit{PSC}\ \beta\ [\![M]\!]
\end{aligned}
$$

This interpretation corresponds to $\lambda\mu$ with the notion of *lazy reduction*. Fisher proceeds to prove various properties of the interpretation - we present the Haskell implementation of λ^{try} yielded by this translation, as obtained in [4].

2.6.3 Final Result

We have mentioned the existence of a CDC library for Haskell [9]. We give below the interface of this library, as presented by Fisher in [4]:

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

instance Monad (CC ans)
```

The control operators have the types:

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

We have seen a translation from λ^{try} to $\lambda\mu$ in subsection 2.4.4 and a translation from $\lambda\mu$ to CDC in subsection 2.6.2. By straightforwardly composing the two translations and using the library illustrated above, Fisher obtains the following implementation of the *try* and *throw* constructs of λ^{try} in Haskell:

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

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

As explained in [4], this suffers from a number of shortcomings. For example, note the absence of the *catch* construct. This is because, with this implementation, the exception handlers are being passed as an argument directly to the `try` function. Furthermore, it also means that the number of exception handlers is not allowed to vary - instead, a new function (such as `try2`, `try3` and so on) has to be defined for each number of handlers. The `try2` function defined in [4] is:

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

It would then be used like:

```
try2 p (\name1 -> \name2 -> return 1)
  (\x -> return $ x+2)
  (\x -> return $ x+4)
```

Notice how the two handlers, `name1` and `name2`, are passed as arguments to `try2` in the form of anonymous functions - the first handler is bound to `name1` and the second one to `name2`. This is also different from the λ^{try} syntax, where names are bound dynamically.

The paper then gives an improved, more aesthetically pleasing implementation, but which still suffers from the problems illustrated above. Nevertheless, as a proof of concept, it shows that a translation from λ^{try} to Haskell is possible.

It concludes with the remarks that "CDC was not the correct choice of calculus to facilitate a translation from λ^{try} to Haskell" and that, in order to model λ^{try} more accurately, some extensions to the language itself would be needed, rather than just making use of the CDC library. Essentially, the syntax we would like to end up with is this (given in [4]):

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

Then, `throw` would specify a name and a value to throw to it, and the corresponding handler would act accordingly. For example (also given in [4]):

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

Presumably, then, in the body of the `try`, some exceptions could be raised - in this case, `parseFile` could contain a throw (that would be caught by the first handler, which in turn might attempt some recovery action):

```
throw fileNotFound pathName
```

The aim of this project is to explore what such a language extension would imply.

Chapter 3

A Haskell Evaluator

To explore the possibility of expressing λ^{try} in Haskell, we first introduce an evaluator for it.

3.1 Definitions

The following data structure declarations are used to represent λ^{try} terms:

```haskell
type Var = String
type Name = String

data TryExpr = Ident Var
             | Abs Var TryExpr
             | App TryExpr TryExpr
             | Throw Name TryExpr
             | Try TryExpr Name Var TryExpr
  deriving (Eq, Show)
```

The first two type synonyms mean that variables and names are both represented as strings. The recursive data type definition closely mimics the syntax of λ^{try}. The full mapping, denoted by $[-]$, maps λ^{try} terms to values of type `TryExpr` in Haskell. It is defined as follows:

$$
\begin{aligned}
[x] &\triangleq \texttt{Ident "x"} \\
[\lambda x.M] &\triangleq \texttt{Abs "x"}\ [M] \\
[MN] &\triangleq \texttt{App}\ [M]\ [N] \\
[\mathit{throw}\ n(N)] &\triangleq \texttt{Throw "n"}\ [N] \\
[\mathit{try}\ M;\ \mathit{catch}\ n(x) = N] &\triangleq \texttt{Try}\ [M]\ \texttt{"n" "x"}\ [N] \\
[\mathit{try}\ M;\ \mathit{Catch\_Block};\ \mathit{catch}\ n(x) = N] &\triangleq \texttt{Try}\ [\mathit{try}\ M;\ \mathit{Catch\_Block}]\ \texttt{"n" "x"}\ [N]
\end{aligned}
$$

3.2 Implementation

Traditionally in Haskell, exception handling is done using data types, such as `Either`:

```haskell
data Either a b = Left a | Right b
```

It is parameterized over two types: `a` and `b`. A value of type `Either a b` is either a `Left` with an associated value of type `a`, or a `Right` with an associated value of type `b`.

This is because, in Haskell, there is no traditional notion of ”throwing” something, like, for example, in Java. Instead, whenever the programmer wants to signal that something has gone wrong (the place where we might usually expect an exception to be thrown), it is usual to incorporate this information in some data type, taking advantage of the powerful typing features of Haskell (as shown in [10]).

For example, we might have a function that looks up a key in a map from strings to integers. While, in Java, this function could throw an exception if the key does not exist, a common approach in Haskell would be to return a `Nothing` in this case. The return type of the function would be `Maybe Int`: either the key was not found and `Nothing` was returned, or the key was found and a `Just Int` was returned (with the corresponding integer wrapped in the `Just`). Thus, the possibility that something might have gone wrong is included in the type itself, and whoever uses the function will be forced to deal with that possibility by the type system. The `Either` data type can be used for similar purposes: one option would be to include an exception message or some other useful information as the argument to the `Left`.

This leads us to explore the possibility of encoding λ^{try} terms using the `Either` data type: an intuitive approach is to represent throws as `Left`'s (as they are the terms that require special treatment in terms of reduction rules) and everything else as `Right`'s.

The evaluator takes a term of type `TryExpr` and runs it, returning one of the following: a `Right TryExpr`, if there was no uncaught throw while evaluating the term - in this case, the result is stored in the argument to `Right` - or a `Left (TryExpr, Name)`, if an uncaught throw was encountered while evaluating the term - in this case, the first argument in the pair represents the computation to do after the throw is caught (the execution of the handler), and the second argument is simply the name to which the throw occurred.

Thus, `Left` and `Right` are used as markers - one representing a throw that needs to be handled, and the other a successful computation. The full code of the evaluator can be found in figure 3.1.

There is one main entry point - the `eval` function. It takes a `TryExpr` and then delegates to an auxiliary function, `evalAux`, which also carries around a context (a map from names to expressions), which is used when evaluating `Throw`'s. The cases are explained below:

1. `Ident x` - this corresponds to a single variable, which is a value in λ^{try}, so there is no reduction to be done; the context is irrelevant and `Right (Ident x)` is returned immediately.

```
1  -- main evaluation function
2  eval :: TryExpr -> Either (TryExpr, Name) TryExpr
3
4  eval e = evalAux e empty
5
6  -- auxiliary evaluation function
7  evalAux :: TryExpr -> Map Name TryExpr -> Either (TryExpr, Name) TryExpr
8
9  evalAux (Ident x) _ = Right (Ident x)
10
11 evalAux (Abs x m) _ = Right (Abs x m)
12
13 evalAux (App (Abs x m) q) env = evalAux (subst m q x) env
14 evalAux (App p q) env =
15   case evalAux p env of
16     Left thr -> Left thr
17     Right p' -> if (p == p') then Right (App p q) else evalAux (App p' q) env
18
19 evalAux (Throw n m) env = Left ((App (env ! n) m), n)
20
21 evalAux (Try m n x catchRes) env =
22   case evalAux m (insert n (Abs x catchRes) env) of
23     Left (res, m) -> if (m == n) then  evalAux res env else Left (res, m)
24     Right res ->
25         if (occurs n res) then Right (Try res n x catchRes) else Right res
```

Figure 3.1: A Haskell evaluator

2. `Abs x m` - again, abstractions are values in λ^{try}, so we proceed similarly to the previous case.

3. `App (Abs x m) q` - this corresponds to $(\lambda x.M)Q$; to run this term, we do the substitution (see `subst` function below) and then run the result.

4. `App p q` - in this case, we do not know what `p` is (although we know it is not an abstraction, otherwise we would have followed the previous branch); therefore, we first evaluate `p`, and then: if `p` evaluated to a `Left` (representing a throw), it needs to consume its applicative context (`q`), so we disregard `q` and return the `Left`. Otherwise, we check if the result of running `p` is different from `p` itself - if it is not, then we stop and return `Right (App p q)`, because there is nothing more to be done and we want to avoid an infinite cycle (our aim is to not introduce non-termination); otherwise, we plug the result in the application and run it in the new form.

Note: lines 14-17 in the evaluator can also be expressed in monadic fashion, since `Either a` is a Haskell monad:

```haskell
evalAux (App p q) env = (evalAux p env) >>=
  (\p' -> if (p == p') then Right (App p q) else evalAux (App p' q) env)
```

5. `Throw n m` - this throws `m` to the name `n`, so we look the latter up in the environment, which should give us an abstraction to which we can apply `m`; we put this application (that represents the execution of the handler), without running it (it will be run when the throw is caught), in a `Left`, together with the name `n`, to remember the targeted handler. (Note that the evaluator assumes the required name will always be present in the environment, which is to say that all the terms on which `eval` is called are closed in this respect; it is not hard to modify the evaluator to take the other case into consideration as well, for example, by using a new variable instead when the name is not present, and the assumption makes our lives easier.)

6. `Try m n x catchRes` - this represents a try with a single handler; we first create a new context by inserting the handler into the existing one (by representing *catch*$; n(x) = N$ as $\lambda x.N$ and storing it as the entry for name `n`) and then evaluate `m` in the new context - there are two cases to consider:

   (a) `m` runs to a `Left` - this means that there was a throw inside: we check if the throw was to the handler of our current `Try` block; if it does, then the throw is caught here and we proceed with evaluating the execution of the handler, which was stored in the `Left`; if not, we just pass the `Left` along to the outer `Try` blocks (until the responsible handler is found).

   (b) `m` runs to a `Right` - in this case, there was no throw when running `m`, and to comply with the λ^{try} reduction rules we need only check (see `occurs` function below) whether the current handler name still occurs in `m` (despite it having not run to a `Left`); if yes, then we can not proceed with evaluation and we simply return `Right (Try res n x catchRes)`, to make sure that the evaluator will terminate; if, however, it does not, then the result is free to escape the `Try` and we return it instead.

The `subst` and `occurs` functions are given below:

```haskell
-- substitution function
subst :: TryExpr -> TryExpr -> Var -> TryExpr

subst (Ident x) sub y
  | x == y    = sub
  | otherwise = Ident x

subst (Abs x m) sub y
  | x == y    = Abs x m
  | otherwise = Abs x (subst m sub y)

subst (App p q) sub x = App (subst p sub x) (subst q sub x)
```

```

subst (Throw n m) sub x = Throw n (subst m sub x)

subst (Try m n x catchRes) sub y
  | x == y     = Try (subst m sub y) n x catchRes
  | otherwise = Try (subst m sub y) n x (subst catchRes sub y)

-- check if a name occurs in a TryExpr
occurs :: Name -> TryExpr -> Bool

occurs _ (Ident _) = False

occurs n (Abs x m) = occurs n m

occurs n (App p q) = (occurs n p) || (occurs n q)

occurs n (Throw n' m) = n == n' || occurs n m

occurs n (Try m n' x catchRes) = n == n' || occurs n m || occurs n catchRes
```

Note that `subst` makes no attempt at α-conversion: we operate under the assumption that any term that `eval` or `evalAux` are ever called on are such that α-conversion is not necessary for correct reduction. This assumption is strong and may seem unjustified, however we only aim to demonstrate the feasibility of our evaluator, and for this purpose it makes matters easier.

3.3 Preservation of Reduction

In what follows, we take `f x` $\downarrow$ to mean that the Haskell call `f x` terminates, and `f x` $\downarrow$ `y` to mean that it terminates with the value of `y` (where `y` is a value or is known itself to terminate). We are now going to prove the main property of our evaluator:

Theorem 3.3.1 (Evaluator Preserves Reduction) *For any λ^{try} terms P and Q such that $P \to^* Q$ and Q is in normal form (with respect to call-by-name reduction), it holds that* `eval` $[P] \downarrow$ `eval` $[Q]$.

The result follows from a number of auxiliary lemmas which we give below.

Lemma 3.3.2 *For any λ^{try} term N and name n, $n \in N \iff$* `"n"` $\in [N] \iff$ `occurs "n"` $[N]$.

The lemma above is stated without proof.

Lemma 3.3.3 *For any λ^{try} terms M and N and variable x,* $[M\{N/x\}] =$ `subst` $[M]$ $[N]$ `"x"`.

The proof is by induction on the structure of terms and is given below:

(x): $[x\{N/x\}] = [N] =$ `subst` $[x]$ $[N]$ `"x"` (by definition of substitution and `subst`)

$(y \neq x)$: $[y\{N/x\}] = [y] =$ `subst` $[y]$ $[N]$ `"x"` (by definition of substitution and `subst`)

$(\lambda x.M)$: $[(\lambda x.M)\{N/x\}] = [\lambda x.M] =$ `Abs "x"` $[M] =$ `subst (Abs "x"` $[M]$`)` $[N]$ `"x"`
$=$ `subst` $[\lambda x.M]$ $[N]$ `"x"` (by definitions)

$(\lambda y.M,\ y \neq x)$: $[(\lambda y.M)\{N/x\}] = [\lambda y.M\{N/x\}] =$ `Abs "y"` $[M\{N/x\}]$
$=$ (IH) `Abs "y" (subst` $[M]$ $[N]$ `"x")` $=$ `subst (Abs "y"` $[M]$`)` $[N]$ `"x"`
$=$ `subst` $[\lambda y.M]$ $[N]$ `"x"` (by def. `subst` and mapping)

(PQ): $[(PQ)\{N/x\}] = [P\{N/x\}Q\{N/x\}] =$ `App` $[P\{N/x\}]$ $[Q\{N/x\}]$
$=$ (IH) `App (subst` $[P]$ $[N]$ `"x") (subst` $[Q]$ $[N]$ `"x")`
$=$ `subst (App` $[P]$ $[Q]$`)` $[N]$ `"x"` $=$ `subst` $[PQ]$ $[N]$ `"x"` (by def. `subst` and mapping)

$(\textit{throw}\ n(M))$: $[(\textit{throw}\ n(M))\{N/x\}] = [\textit{throw}\ n(M\{N/x\})] =$ `Throw "n"` $[M\{N/x\}]$
$=$ (IH) `Throw "n" (subst` $[M]$ $[N]$ `"x")` $=$ `subst (Throw "n"` $[M]$`)` $[N]$ `"x"`
$=$ `subst` $[\textit{throw}\ n(M)]$ $[N]$ `"x"` (by def. `subst` and mapping)

$(\textit{try}\ M;\ \textit{catch}\ m(x) = L)$: $[(\textit{try}\ M;\ \textit{catch}\ m(x) = L)\{N/x\}]$
$= [\textit{try}\ M\{N/x\};\ \textit{catch}\ m(x) = L] =$ `Try` $[M\{N/x\}]$ `"m" "x"` $[L]$
$=$ (IH) `Try (subst` $[M]$ $[N]$ `"x") "m" "x"` $[L] =$ `subst (Try` $[M]$ `"m" "x"` $[L]$`)` $[N]$ `"x"`
$=$ `subst` $[\textit{try}\ M;\ \textit{catch}\ m(x) = L]$ $[N]$ `"x"` (by def. `subst` and mapping)

$(\textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i;}\ \textit{catch}\ m(x) = L)$:
$[(\textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i;}\ \textit{catch}\ m(x) = L)\{N/x\}]$
$= [\textit{try}\ M\{N/x\};\ \overline{\textit{catch}\ n_i(x) = N_i;}\ \textit{catch}\ m(x) = L]$
$=$ `Try` $[\textit{try}\ M\{N/x\};\ \overline{\textit{catch}\ n_i(x) = N_i}]$ `"n" "x"` $[L]$
$=$ `Try` $[(\textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i})\{N/x\}]$ `"n" "x"` $[L]$
$=$ (IH on the num. of catches) `Try (subst` $[\textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i}]$ $[N]$ `"x") "m" "x"` $[L]$
$=$ `subst (Try` $[\textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i}]$ `"m" "x"` $[L]$`)` $[N]$ `"x"`
$=$ `subst` $[\textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i;}\ \textit{catch}\ m(x) = L]$ $[N]$ `"x"` (by def. `subst` and mapping)

Note: for the last case, there is an implicit IH on the number of catches in the catch block (due to the nesting nature of the mapping). Additionally, in the last two cases, if we substitute for y with $y \neq x$, the proofs are almost identical (except for additional applications of the IH on the outer handlers), so we do not show them here. QED.

Lemma 3.3.4 *For any terms P and Q such that $P \to Q$ and $\forall \epsilon.$* `evalAux` $[Q]$ $\epsilon \downarrow$*, it holds that $\forall \epsilon.$* `evalAux` $[P]$ $\epsilon \downarrow$ `evalAux` $[Q]$ ϵ.

The proof is by induction on $\to$:

Assume $P \to Q$ and $\forall\epsilon.$ `evalAux` $[Q]\ \epsilon \downarrow$. Take an arbitrary context ϵ. We must show that `evalAux` $[P]\ \epsilon \downarrow$ `evalAux` $[Q]\ \epsilon$.

Base Cases:

(β): then $P = (\lambda x.M)N$, $Q = M\{N/x\}$ and we have (abbreviating `evalAux` by `eA` for brevity):

`eA` $[P]\ \epsilon$
$=$ `eA (App (Abs "x"` $[M]$`)` $[N]$`)` ϵ (by def. of mapping)
$=$ `eA (subst` $[M]\ [N]$ `"x")` ϵ (by def. of algorithm)
$=$ `eA` $[M\{N/x\}]\ \epsilon$ (by Lemma 3.3.3)
$=$ `eA` $[Q]\ \epsilon \downarrow$, as required.

(*throw*): then $P = (\textit{throw}\ n(N))M$, $Q = \textit{throw}\ n(N)$ and we have:

```
eA [P] ϵ
= eA (App (Throw "n" [N]) [M]) ϵ    (by def. mapping)
= case eA (Throw "n" [N]) ϵ of
      ...      (by def. algorithm)
= case Left (App (ϵ "n") [N], "n") of
      Left thr -> Left thr
      ...      (by def. algorithm)
= Left (App (ϵ "n") [N], "n")    (by Haskell rules)
= eA (Throw "n" [N]) ϵ    (by def. algorithm)
= eA [throw n(N)] ϵ    (by def. mapping)
= eA [Q] ϵ ↓
```

□

(*try-throw*): then $P = \textit{try throw}\ n_l(N);\ \overline{\textit{catch}\ n_i(x) = N_i};\ \textit{catch}\ n_l(x) = M_l$, $Q = M_l\{N/x\}$.

Note that, in what follows, we will write Γ_n for the context
$\{$`"n`$_1$`"` : `Abs "x"` $[M_1]$, `"n`$_2$`"` : `Abs "x"` $[M_2], ...,$ `"n`$_n$`"` : `Abs "x"` $[M_n]\}$. We have:

```
eA [P] ϵ
= eA (Try [try throw n_l(N); catch n_i(x) = N_i] "n_1" "x" [M_l]) ϵ
= case eA [try throw n_l(N); catch n_i(x) = N_i] ϵ ∪ {"n_1" : Abs "x" [M_l]} of
     Left (res, m) ->
          if (m == "n_1") then eA res ϵ else Left (res, m)
     ...      (by def. algorithm)
= case
      case
          ...
             case eA [throw n_l(N)] ϵ ∪ {"n_1" : Abs "x" [M_l]} ∪ Γ_n of
```

```
                  Left (res, m) ->
                      if (m == "n₁") ... else Left (res, m)
                  Right res -> ...
              of Left (res, m) -> ...
                 Right res -> ...
          ...
      of Left (res, m) ->
              if (m == "nₙ") ... else Left (res, m)
          ...
 of Left (res, m) -> if (m == "n₁") then eA res ϵ else ...
    Right res -> ...      (by def. algorithm, unfolding)
= case
      case
          ...
              case Left (App (Abs "x" [M_l]) [N], "n₁") of
                  Left (res, m) ->
                      if (m == "n₁") ... else Left (res, m)
                  Right res -> ...
              of Left (res, m) -> ...
                 Right res -> ...
          ...
      of Left (res, m) ->
              if (m == "nₙ") ... else Left (res, m)
          ...
 of Left (res, m) -> if (m == "n₁") then eA res ϵ else ...
    Right res -> ...      (evaluating the throw)
```

Note that λ^{try} rules say that the names n_1, n_2, ..., n_n and n_l have to be pairwise distinct.

Consider, then, the first if-test that is going to be executed above: it is going to be `if ("n₁" == "n₁")`, which is going to fail because of the reason stated above. The else branch will be taken, and the `Left` is going to go unchanged into the second test, which has `"n₂"` instead of `"n₁"`, and so on. Therefore the `Left` is going to escape the first n case constructs unchanged, and continuing from the last point, we have:

```
...
= case Left (App (Abs "x" [M_l]) [N], "n₁") of
      Left (res, m) -> if (m == "n₁") then eA res ϵ else ...
      Right res -> ...      (by Haskell rules)
= eA (App (Abs "x" [M_l]) [N]) ϵ    (by Haskell rules)
= eA (subst [M_l] [N] "x") ϵ    (by def. algorithm)
= ea [M_l{N/x}] ϵ    (by Lemma 3.3.3)
= ea [Q] ϵ ↓
```

$\square$

(*try-normal*): then $P = \textit{try}\ N;\ \overline{\textit{catch}\ n_i(x) = M_i}$, $Q = N$, $\overline{n_i \notin N}$ and we have:

```
eA [P] ϵ
= case
       ...
          case eA [N] ϵ ∪ Γn of
              Left (res, m) ->
                  if (m == "n1") ... else Left (res, m)
              Right res ->
                  if (occurs "n1" res) ... else Right res
       ...
  of Left (res, m) ->
          if (m == "nn") ... else Left (res, m)
     Right res ->
          if (occurs "nn" res) ... else Right res
```

In the above, we have unfolded the algorithm similarly to what we had in the previous case. Note that, by assumption, we know `eA` $[N]$ $\epsilon \cup \Gamma_n \downarrow$. There are two cases to consider:

1. `eA` $[N]$ $\epsilon \cup \Gamma_n$ evaluates to `Left (res', m')`. In this case, we have that `m'` can not be any of `"n1"`, ..., `"nn"`. We can prove this by contradiction: if it was equal to some `"ni"`, it would have to have come from evaluating some `Throw "ni"` $[M]$ while evaluating $[N]$, which by Lemma 3.3.2 implies that N contains the name n_i, which we know is false.

   Given that all of the if-tests will be of the form `if (m' == "ni")`, from the previous argument, it follows that they will all fail and the `Left` will escape them all unchanged. Continuing from the last point, then:

   ...
   = `Left (res', m')` (by the above and assumption)
   = `eA` $[N]$ $\epsilon \cup \Gamma_n$ (by assumption)

   However, note that $[N]$ does not contain any of the names defined in Γ_n, so there will never be a usage of any `"ni"`. As such, removing Γ_n can not make a difference to the evaluation, because it is never used. Continuing, then, we have:

   ...
   = `eA` $[N]$ ϵ (by the above)
   = `eA` $[Q]$ $\epsilon \downarrow$
   □

2. `eA` $[N]$ $\epsilon \cup \Gamma_n$ evaluates to `Right res'`. By Lemma 3.3.2, we know $[N]$ does not contain any `"ni"` - therefore, `res'`, which is the result of evaluating $[N]$, can not contain any `"ni"` either. We can prove this by contradiction: assume `res'` contained some `"ni"` - since it was not in $[N]$ to begin with, it follows that it must have somehow been introduced during evaluation, and the only possibility is if this happened via some lookup in the context. There are two options:

either it came from some handler in ϵ, implying that a handler in an outer try block references a handler in the inner one, or it came from some handler in Γ_n, implying that a handler in the current try block references another handler in the same try block. None of these options are allowed under λ^{try} rules.

We therefore have that `res'` does not contain any `"n_i"` and by Lemma 3.3.2 all tests of the form `occurs "n_i" res'` are going to fail, so the `Right` is going to escape all the case blocks unchanged. Continuing from the last point:

$\texttt{...}$
$= \texttt{Right res'}$ (by the above and assumption)
$= \texttt{eA}\ [N]\ \epsilon \cup \Gamma_n$ (by assumption)
$= \texttt{eA}\ [N]\ \epsilon$ (similar reasoning to case 1)
$= \texttt{eA}\ [Q]\ \epsilon \downarrow$
$\square$

Inductive Cases:

$(P \to Q \Rightarrow PM \to QM)$: assume (IH) that, if $\forall \epsilon.\ \texttt{eA}\ [Q]\ \epsilon \downarrow$, then $\forall \epsilon.\ \texttt{eA}\ [P]\ \epsilon \downarrow \texttt{eA}\ [Q]\ \epsilon$.

Must show that if $\forall \epsilon.\ \texttt{eA}\ [QM]\ \epsilon \downarrow$, then $\forall \epsilon.\ \texttt{eA}\ [PM]\ \epsilon \downarrow \texttt{eA}\ [QM]\ \epsilon$.

Assume (ass1) that $\forall \epsilon.\ \texttt{eA}\ [QM]\ \epsilon \downarrow$. We must now show the right hand side of the implication above. For any ϵ, we have that:

$\texttt{eA}\ [QM]\ \epsilon$
$= \texttt{eA (App}\ [Q]\ [M]\texttt{)}\ \epsilon$ (by def. mapping)

If Q is an abstraction, then it is clear from the algorithm that $\texttt{eA}\ [Q]\ \epsilon \downarrow$ for any ϵ (line 11 in the evaluator). Otherwise, the above becomes:

$\texttt{...}$
$= \texttt{case eA}\ [Q]\ \epsilon\ \texttt{of ...}$

which we know from (ass1) must terminate for any ϵ. Since the term inside the `case` is first evaluated before proceeding, it follows that it must also terminate for any ϵ, meaning $\forall \epsilon.\ \texttt{eA}\ [Q]\ \epsilon \downarrow$.

Therefore, in both cases we have $\forall \epsilon.\ \texttt{eA}\ [Q]\ \epsilon \downarrow$, and by the IH we have that $\forall \epsilon.\ \texttt{eA}\ [P]\ \epsilon \downarrow \texttt{eA}\ [Q]\ \epsilon$. (*) We now take an arbitrary ϵ and we must show $\texttt{eA}\ [PM]\ \epsilon \downarrow \texttt{eA}\ [QM]\ \epsilon$. We have:

$\texttt{eA}\ [PM]\ \epsilon$
$= \texttt{case eA}\ [P]\ \epsilon\ \texttt{of ...}$ (by def. algorithm)
$= \texttt{case eA}\ [Q]\ \epsilon\ \texttt{of ...}$ (by (*))
$= \texttt{eA}\ [QM]\ \epsilon \downarrow$ (by def. algorithm and (ass1))
$\square$

$(P \to Q \Rightarrow$ *try* $P;$ *CB* $\to$ *try* $Q;$ *CB*$)$: the IH is the same as before. We must show that, if $\forall \epsilon.\ \texttt{eA}$ [*try* $Q;$ *CB*] $\epsilon \downarrow$, then $\forall \epsilon.\ \texttt{eA}$ [*try* $P;$ *CB*] $\epsilon \downarrow \texttt{eA}$ [*try* $Q;$ *CB*] ϵ. Assume (ass1) the left hand side of the implication above. Then we must show the right hand side.

Note that, for any ϵ, we have:

```
eA [try Q; CB] ε
= case
        ...
            case eA [Q] ε ∪ Γn of ...
        ...
  of ...
```

which we know from (ass1) must terminate. Since the term inside the innermost `case` is evaluated first, it follows that it must also terminate for any context $\epsilon \cup \Gamma_n$. However, we can take Γ_n to contain only names that do not appear in Q, and therefore, as argued previously, not making a difference to the evaluation. If we take such a Γ_n, then `eA` $[Q]$ $\epsilon \cup \Gamma_n =$ `eA` $[Q]$ ϵ, which we deduce must terminate for any ϵ. So $\forall \epsilon.$ `eA` $[Q]$ $\epsilon \downarrow$ and we can apply the IH to obtain that $\forall \epsilon.$ `eA` $[P]$ $\epsilon \downarrow$ `eA` $[Q]$ ϵ. (*)

We now take an arbitrary ϵ and must show that `eA` $[$*try* $P;$ *CB*$]$ $\epsilon \downarrow$ `eA` $[$*try* $Q;$ *CB*$]$ ϵ. We have:

```
eA [try P; CB] ε
= case
        ...
            case eA [P] ε ∪ Γn of ...
        ...
  of ...      (by def. algorithm, unfolding)
= case
        ...
            case eA [Q] ε ∪ Γn of ...
        ...
  of ...      (by (*))
= eA [try Q; CB] ε ↓   (by def. algorithm and (ass1))
```

□

This completes the proof of Lemma 3.3.4.

Lemma 3.3.5 *If a term H is in normal form with respect to CBN reduction, then* $\forall \epsilon.$ `eA` $[H]$ $\epsilon \downarrow$.

The proof is direct, using the structure of terms in normal form. Note that these can be described by the following rules:

$$H ::= xM_1...M_n \mid \lambda x.M \mid \textit{throw}\ n(N) \mid (\textit{try}\ H;\ \textit{CB})M_1...M_n$$

where the H in the last case is not allowed to be a *throw* and must contain at least one of the names defined in the catch block, M_i are regular λ^{try} terms and $n \geq 0$.

We also define a related syntax, called H_2:

$$H_2 ::= xM_1...M_n \mid \lambda x.M \mid \textit{throw}\ n(N) \mid (\textit{try}\ H_2;\ \textit{CB})M_1...M_n$$

where the conditions on H_2 in the last case are as above, except that H_2 contains **all** the names in the catch block. We will see that H_2 represents, in a sense, the fixed points of the evaluator. Note that this syntax generates a subset of what the first one generates. We will employ the following result:

Lemma 3.3.6 *The result of running* `eA` $[\textit{try}\ H;\ \textit{CB}]\ \epsilon$ *(where try $H;$ CB is as described in the syntax H) is going to be of the form* `Right` $[\textit{try}\ H_2;\ \textit{CB}_2]$ *for some H_2 and $\textit{CB}_2$ (where try $H_2;$ $\textit{CB}_2$ is as described in the syntax H_2).*

This is because the evaluator models a slightly stronger reduction than stipulated in λ^{try} - specifically, it allows the removal of any handler *catch* $n_i(x) = ...$ from the catch block in the term *try* $M;$ *CB* if it satisfies $n_i \notin M$. This happens because, when mapping terms to Haskell, *try*-terms are translated as nested `Try` expressions, each with a single handler. The evaluator has no global information and evaluates each handler in part, discarding the ones that are unnecessary, as opposed to stopping straight away (with all handlers intact) if *any* of the names exist in the term (as described in λ^{try}).

This is a price that we pay for the simplicity of the data structure, which employs nesting to map *try*-terms. It could be fixed by making the data structure more complex, by adding an explicit list of handlers to the *Try* expression instead of a single handler, however the current solution suffices for our purpose. This lemma can be proved by induction on the structure of H, however we do not give a proof here.

Pick an arbitrary ϵ. We must show `eA` $[H]\ \epsilon \downarrow$. The proof of Lemma 3.3.5 proceeds as follows:

$(xM_1...M_n)$:

If $n = 0$, then `eA` $[x]\ \epsilon =$ `Right (Ident "x")`, so it terminates.

If $n \geq 1$, then we have:

```
eA [xM_1...M_n] ϵ
= case
     ...
        case Right (Ident "x") of
            Left -> ...
            Right p' -> if (Ident "x" == p') then
                Right (App (Ident "x") [M_1]) else ...
     ...
  of Left -> ...
     Right p' -> if (App_{n-1} == p') then Right App_n else ...
```
(by unfolding and evaluating `Ident "x"`)

where by App_i we mean

`App (App (... (App (Ident "x")` $[M_1]$`)` $[M_2]$`) ... )` $[M_i]$

The innermost if-test will be successful, and the innermost case construct will return `Right` App_1. The second if-test will test for equality with App_1 and it will be successful, thereby returning `Right` App_2. The process will go on until, in the end, `Right` App_n is returned, which proves that the call terminates (and in fact $App_n = [xM_1...M_n]$).

$(\lambda x.M)$: then `eA` $[\lambda x.M]\ \epsilon =$ `Right (Abs "x"` $[M]$`)`, so it terminates.

$(\textit{throw}\ n(N))$: then `eA` $[\textit{throw}\ n(N)]\ \epsilon =$ `Left (App (`ϵ` "n")` $[N]$`, "n")`, so it terminates.

$((\textit{try}\ H;\ \textit{CB})M_1...M_n)$, where H can not be a throw and must contain at least one name defined in *CB*:

If $n = 0$: then, by Lemma 3.3.6, we have that `eA` $[\textit{try}\ H;\ \textit{CB}]\ \epsilon =$ `Right` $[\textit{try}\ H_2\ \textit{CB}_2]$ for some H_2 and $\textit{CB}_2$ as described in the lemma, so it terminates.

If $n \geq 1$: we first prove an additional statement by induction on the structure of H_2: for any H_2 (where we disregard the *throw* from the syntax) and ϵ, we have that `eA` $[H_2]\ \epsilon =$ `Right` $[H_2]$ (*) - this is what we mean when saying that H_2 (where we disregard *throw*'s from the syntax) are the fixed points of the evaluator.

We have already seen proofs for $xM_1...M_n$ and $\lambda x.M$ above (it is easy to check that the result of the evaluation is the translation of the term we started with, encased in a `Right`). The remaining (inductive) case is $(\textit{try}\ H_2;\ \textit{CB})M_1...M_n$ where H_2 is not a *throw*. Assume (IH) that $\forall\epsilon.$ `eA` $[H_2]\ \epsilon =$ `Right` $[H_2]$. Let us consider the case when $n = 0$ - we must show `eA` $[\textit{try}\ H_2;\ \textit{CB}]\ \epsilon =$ `Right` $[\textit{try}\ H_2;\ \textit{CB}]$:

```
eA  [try H₂; CB]  ε
= case
      ...
         case eA  [H₂]  ε ∪ Γₙ
         of ...
      ...
  of ...    (unfolding the catch block)
= case
      ...
         case Right [H₂] of
             Left -> ...
             Right res -> if (occurs "n₁" res)
                 then Right (Try res "n₁" "x" [M₁]) else ...
      ...
  of Left -> ...
     Right res -> if (occurs "nₙ" res)
```

```
        then Right (Try res "n_n" "x" [M_n]) else ...
```
(by the IH)

In the above, we have applied the IH and unfolded the catch block - note how each catch block introduces a check: if its name exists in the term, then it is kept, otherwise it is discarded. We know by definition that H_2 contains all of the names in the catch block, ie. all of $\mathtt{n_1}$, $\mathtt{n_2}$, ..., $\mathtt{n_n}$, so by Lemma 3.3.2 all of the if-tests will be successful. It is easy to see that the i^{th} innermost case construct will return `Right` Try_i, where Try_i is defined as:

```
Try (Try (... (Try [H_2] "n_1" "x" [M_1]) ...) "n_i" "x" "[M_i]"
```

The last case construct will therefore return `Right` Try_n, which is the final result of the call; note that $\mathit{Try}_n = [\mathit{try}\ H_2;\ \mathit{CB}]$, as required.

If $n \geq 1$, we have $(\mathit{try}\ H_2;\ \mathit{CB})M_1...M_n$ - the proof will proceed in a similar fashion, except that there are n applications to unwrap first. This is exactly the same reasoning as the proof for $xM_1...M_n$ above, with the derivation for the case $n = 0$ (that we have just finished) appearing as part of it once the applications are unwrapped. Therefore we do not show it here. This completes the proof of the auxiliary statement.

Back to the main proof, we now have to deal with $(\mathit{try}\ H;\ \mathit{CB})M_1...M_n$. We have:

```
eA [(try H; CB)M_1...M_n] ε
= case
      ...
         case eA [try H; CB] ε of ...
      ...
  of ...     (unfolding the applications)
= case
      ...
         case Right [try H_2; CB_2] of
             Left -> ...
             Right p' -> if ([try H; CB] == p')
                 then Right (App [try H; CB] [M_1])
                 else eA (App p' [M_1]) ε
      ...
  of ...     (by Lemma 3.3.6)
```

At this point, there are two cases:

1. There is a chance that the initial *try* H; *CB* already conformed to H_2 syntax (which says that every term that is in a *try* references all of the names in the catch block). Then, by (*), we would have that $[\mathit{try}\ H;\ \mathit{CB}]$ == $[\mathit{try}\ H_2;\ \mathit{CB}_2]$. The if-test in the innermost case construct will be successful and `Right` $[(\mathit{try}\ H;\ \mathit{CB})M_1]$ will be returned. The other case constructs will behave similarly, with the i^{th} innermost one returning `Right` $[(\mathit{try}\ H;\ \mathit{CB})M_1M_2...M_i]$. The final result will be `Right` $[(\mathit{try}\ H;\ \mathit{CB})M_1M_2...M_n]$, therefore the algorithm terminates.

2. Otherwise, we have that $[try\ H;\ CB] \neq [try\ H_2;\ CB_2]$. The innermost if-test will detect the difference and return a re-evaluation: `eA` $[(try\ H_2;\ CB_2)M_1]\ \epsilon$. Following the reasoning from case 1, this will return `Right` $[(try\ H_2;\ CB_2)M_1]$ (since we definitely know that *try* $H_2;\ CB_2$ conforms to H_2 syntax now). This will be returned to the second innermost case construct, which in turn will also re-evaluate, having spotted a difference: it returns `eA` $[(try\ H_2;\ CB_2)M_1M_2]\ \epsilon$, which by the same reasoning from case 1 is going to be `Right` $[(try\ H_2;\ CB_2)M_1M_2]$. This happens all the way until the outermost case construct, which returns a final re-evaluation (and the final result of the call): `Right` $[(try\ H_2;\ CB_2)M_1M_2...M_n]$. So the algorithm terminates.

In both cases, the algorithm terminates and this completes the proof of Lemma 3.3.5.

Lemma 3.3.7 *If $P \to^* Q$ and Q is in normal form, then* $\forall\epsilon.$ `eA` $[P]\ \epsilon \downarrow$ `eA` $[Q]\ \epsilon$.

This follows directly from Lemma 3.3.4 and Lemma 3.3.5. We have the following reduction path:

$$P \to P_1 \to ... \to P_n \to Q$$

where Q is in normal form. Applying Lemma 3.3.5 to Q, we have that $\forall\epsilon.$ `eA` $[Q]\ \epsilon \downarrow$.

Then, applying Lemma 3.3.4 to P_n and Q, we have that $\forall\epsilon.$ `eA` $[P_n]\ \epsilon \downarrow$ `eA` $[Q]\ \epsilon$.

Applying Lemma 3.3.4 to P_{n-1} and P_n, we have that $\forall\epsilon.$ `eA` $[P_{n-1}]\ \epsilon \downarrow$ `eA` $[P_n]\ \epsilon$, but since we know that $\forall\epsilon.$ `eA` $[P_n]\ \epsilon \downarrow$ `eA` $[Q]\ \epsilon$, it follows that $\forall\epsilon.$ `eA` $[P_{n-1}]\ \epsilon \downarrow$ `eA` $[Q]\ \epsilon$.

We can keep this up until we reach P, which yields $\forall\epsilon.$ `eA` $[P]\ \epsilon \downarrow$ `eA` $[Q]\ \epsilon$ as required.

Main result

In particular, by setting ϵ to the empty context in Lemma 3.3.7, we obtain Theorem 3.3.1: For any λ^{try} terms P and Q such that $P \to^* Q$ and Q is in normal form (with respect to call-by-name reduction), it holds that `eval` $[P] \downarrow$ `eval` $[Q]$. This completes the proof of the main result.

Chapter 4

GHC and System F

GHC [11] is one of the best-known Haskell compilers and the one that we will preoccupy ourselves with. The key idea is that GHC contains an intermediate language called Core, to which all compiled Haskell code is eventually translated to. While Core used to be an implementation of a variant of a calculus called System F ([12], [13]), it has been extended in various ways over time. Eventually, System F has evolved into another calculus, called System FC, which, at the time of writing this paper, is implemented in GHC (in the form of Core) [14]. This chapter reviews the main structure of GHC and presents the above-mentioned calculi that it is based on.

4.1 GHC

In order to see what it would mean to make Haskell implement λ^{try}, it is required to understand the workings of one of the best-known Haskell compilers: GHC [11].

Broadly speaking, GHC adheres to the traditional structure of a compiler: the source code is first parsed into an intermediate representation, various transformations are run on said representation, and the process ends with the code generation phase, which yields machine code.

A detailed explanation of the whole compiler pipeline is given in [15]. We present below a summary of it:

1. The Haskell source code is parsed into an intermediate, much simpler language, called *Core*
2. A number of Core-to-Core transformations are run on the result of the previous step
3. The code is further simplified and converted into another intermediate language called *STG*
4. STG is converted into a subset of C called *C--*, after which the code generation phase follows one of three paths:

(a) The C-- code may be pretty-printed as stylised C, for compilation with GCC (the C compiler)

(b) If generating native machine code, the native code generator is invoked

(c) If generating LLVM [16] code, the LLVM code generator is invoked

GHC is where features of the normal λ-Calculus are already implemented, for example term substitution. In principle, one could directly extend GHC with everything that λ^{try} introduces: new syntax, new reduction rules and new type assignment rules. However, this would involve major changes on all of the compiler levels detailed above and the effect of such an approach on the existing Haskell features is unpredictable.

Therefore, it makes sense to wonder if such a major extension is actually needed to accomplish our goal: is it not possible to express λ^{try} using whatever GHC already has to offer? To this extent, we further explore the expressive power of the compiler.

4.2 Core and System F

We have mentioned that, as a first step, Haskell source code is converted into an intermediate language called Core. Therefore, if we were able to translate our new language features (that would be added by λ^{try}) into Core, we would not have to touch any other lower level of the compiler.

4.2.1 Before System FC

Core is a very simple language compared to the Haskell source code. As we have mentioned, it used to be the case ([12], [13]) that Core was an implementation of a variant of a calculus called System F. We present it, together with some examples, below, by adapting the information in [13] and [17].

System F specification

The main feature of System F is that it formalizes parametric polymorphism by introducing the type-level lambda. It is also worth noting that System F is a typed language: when abstracting over a variable, that variable has to be annotated with its type. Thus, types are now part of the syntax of terms: this is unlike any of the calculi we have presented so far.

Types are given by the following grammar:

$$A,\ B ::= \varphi \mid A \to B \mid \forall \alpha . A$$

where φ and α range over the set of type variables.

Terms are given by the following grammar:

$$M,\ N ::= x \mid \lambda x^A.M \mid MN \mid \Lambda\varphi.M \mid MA$$

$\forall\alpha.A$ is a *polymorphic* type and it can be viewed as the collection of all the types that can be obtained from A by replacing the free occurrences of α in it with any other type. For example, we might expect a polymorphic identity function to have type $\forall\alpha.\alpha \to \alpha$, because it takes an object and it just gives it back (and this works for any type α that the object may have, hence the $\forall\alpha$). The terms that have such types are characterised by the Λ operator: this has the same meaning as λ in the normal abstraction $\lambda x.M$, except that it is for types. $\Lambda\varphi.M$ can be viewed as a function that takes a type as an input parameter and that type may be used later inside M - this passing of a type as a parameter is represented by the application MA (M is expected to be a Λ-abstraction, and A a type).

Note that the x in the usual $\lambda x.M$ now has to be explicitly annotated with its type A (which can also include type variables bound by Λ-abstractions). Also note that, given the new constructs (type application, specifically) types are partly treated as ordinary terms in System F.

The same conventions (regarding brackets and associativity) that we employed for the normal λ-Calculus apply here.

System F adds the following reduction rule:

$$(\Lambda\varphi.M)A \to M[A/\varphi]$$

where type variable substitution is defined similarly to term variable substitution (also taking place in type annotations), taking into account Barendregt's Convention and assuming that α-conversion takes place silently whenever necessary.

There are also two new type assignment rules, one for each new construct (we also show the one for the normal $\lambda x.M$, which is slightly different because of the type annotation):

$$\frac{\Gamma, x : A \vdash M : B \quad x \notin \Gamma}{\Gamma \vdash \lambda x^A.M : A \to B}\ (\to I) \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \Lambda\varphi.M : \forall\varphi.A}\ (\forall I) \qquad \frac{\Gamma \vdash M : \forall\varphi.B}{\Gamma \vdash MA : B[A/\varphi]}\ (\forall E)$$

For example, the polymorphic identity function (which takes an x and gives back the same x) can be represented in System F as $\Lambda\alpha.\lambda x^\alpha.x$, which has type $\forall\alpha.\alpha \to \alpha$. The advantage of polymorphism then becomes clear: if we were to denote this function by f (and if *Int* and *Char* were types), then the expression $((f\ \textit{Int})\ 1,\ (f\ \textit{Char})\ 'a')$ would typecheck, whereas if f was just $\lambda x.x$, the expression $(f\ 1,\ f\ 'a')$ would not, because f it would need to have both type *Int* $\to$ *Int* and *Char* $\to$ *Char* (so we would end up having to implement two different identity functions, one for each type).

As mentioned in [13], type checking (the problem of deciding whether a type can be assigned to a particular term or not) for this system is straightforward because of the type annotations contained in the terms; however (also as mentioned in [13]), it has been proved that the same problem for a variant of this calculus *without* the annotations is undecidable, which makes it not very practical for serving as the basis of a functional programming language. Instead, a relaxation of this system, called *Hindley-Milner*, was used.

Hindley-Milner specification

Here we present the main feature of Hindley-Milner, as described in [6] and [18].

The principal difference from System F is that Hindley-Milner only allows $\forall$ quantifiers for types to appear on the top level of the type. This makes type checking (the problem of deciding whether a type - either any type at all or a specific type - can be assigned to a particular term) efficiently decidable. Thus, types are classified into *monotypes*:

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

and *polytypes*:

$$\sigma ::= A \mid \forall\varphi.\sigma$$

The general form of a type is therefore $\forall\varphi_1 \ldots \forall\varphi_n.A$, $n \geq 0$, where A is a monotype.

Terms are defined as for the λ-Calculus, with the addition of:

$$M,\ N ::= \ldots \mid \textit{let } x = M \textit{ in } N$$

The notion of reduction is extended by:

$$\textit{let } x = M \textit{ in } N \to N[M/x]$$

The typing rules deal with the new construct and new kinds of types:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ (Ax) \qquad \frac{\Gamma, x : A \vdash M : B \quad x \notin \Gamma}{\Gamma \vdash \lambda x.M : A \to B}\ (\to I) \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\to E)$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma, x : \sigma \vdash N : B \quad x \notin \Gamma}{\Gamma \vdash \textit{let } x = M \textit{ in } N : B}\ (let) \qquad \frac{\Gamma \vdash M : \sigma \quad \varphi \notin \Gamma}{\Gamma \vdash M : \forall\varphi.\sigma}\ (\forall I) \qquad \frac{\Gamma \vdash M : \forall\varphi.\sigma}{\Gamma \vdash M : \sigma[A/\varphi]}\ (\forall E)$$

The rule *(let)* is where polymorphism is introduced: note that it allows *x* to have a polytype, while the rules $(\to I)$ and $(\to E)$ do not allow polytypes. Additionally, the last two rules deal with the *generalisation* and *specialisation* of types: broadly speaking, one can generalise a type by introducing a $\forall$ quantifier over a type variable in it, and one can specialise a $\forall$ type by using a type substitution over the variable that is bound.

For example, this typing system is powerful enough to express self application for the polymorphic identity function: *let* $i = \lambda x.x$ *in* ii can be shown to have type $A \to A$ for any A.

This system can also be extended with recursion, as shown in [6] and [18], but we do not give the details here.

Relationship to Core

Due to its expressivity and practical solution to the type checking problem, Hindley-Milner was implemented in many functional programming languages, such as ML or Haskell [18].

In particular, Core used to be an implementation of Hindley-Milner used in GHC [12], although it has been extended in various ways over the years [18].

4.2.2 System FC

At the time of writing this report, Core is an implementation of *System FC* ([14], [19]), which is an extension to (and also a superset of) System F that we describe below.

As mentioned in [14], System FC builds upon System F by introducing *type equality coercions*. A coercion is a piece of evidence that can be passed around and symbolizes that two types can and should be considered equal.

There are also many additional features, such as data constructors (with *case* expressions), type functions and value type constructors. We give below the fragment of the syntax that shows *expressions*, taken from [14].

$$
\begin{gathered}
e ::= u \quad \text{(Term atoms)} \\
\mid \Lambda\alpha : \kappa.e \mid e\ \varphi \quad \text{(Type abstraction/application)} \\
\mid \lambda x : \sigma.e \mid e_1\ e_2 \quad \text{(Term abstraction/application)} \\
\mid \textbf{let}\ x : \sigma = e_1\ \textbf{in}\ e_2 \\
\mid \textbf{case}\ e_1\ \textbf{of}\ \overline{p \to e_2} \\
\mid e \blacktriangleright \gamma \quad \text{(Cast)}
\end{gathered}
$$

In the above, u stands for term atoms (which are either variables or *data constructors* - not shown here), κ is a *kind* (not shown here) that accompanies the type α in the Λ-abstraction, p is a *pattern* (not shown here), φ stands for a type or a *coercion* (not shown here), σ is a type and γ is a coercion. The full syntax is quite complex and can be found in [14], on page 4.

Most of the constructs have familiar forms: we have already seen variables, variable/type abstraction/application and *let*-terms. The *case* construct resembles Haskell syntax and allows different actions to be taken based on the form of e_1 (patterns and data constructors are also something new that the syntax introduces), and casts are statements (involving an expression and a coercion) which are used to implement advanced typing features in the compiler.

For reasons of brevity, we do not give the type assignment rules or operational semantics, although they can be found in [14] on pages 5 and 9, respectively.

The authors go on to show how several features can be implemented in System FC, such as generalised abstract data types (GADTs) and associated types.

A Core data type

As mentioned before, Core is an implementation of System FC [14]. In concrete terms, since GHC is itself written in Haskell, this means that Core consists on data types (and functions on them) that represent System FC: Haskell source code is parsed into these data types, and machine code is then generated from them (following the compiler pipeline).

Here is how the Core data type looks like for expressions (taken from [20]), which would correspond to the fragment of the syntax that we have given above:

```
type CoreExpr = Expr Var

data Expr b          -- "b" for the type of binders,
  = Var             Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type

type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)

data AltCon = DataAlt DataCon | LitAlt  Literal | DEFAULT

data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]
```

The correspondence between this data type and the System FC syntax is clear from the names of the constructors.

As mentioned in [20], all of Haskell is eventually compiled down to this relatively small data type.

Ideas for λ^{try} translation

Since all of Haskell is compiled down to Core, which is an implementation of System FC, it follows that what we should be looking for is a suitable translation from λ^{try} to System FC (or a version of it with as few extensions as possible). Such a translation would then pave the way for adding actual language features (based on λ^{try}) which should then straightforwardly translate into Core.

Chapter 5

Extension to Hindley-Milner

In this chapter, following our previous reasoning, we present an extension to Hindley-Milner (as presented previously) that allows λ^{try} to be mapped to it. We have chosen to target Hindley-Milner, rather than System FC directly, as it is much simpler and we think it provides a good starting point to see what a translation from λ^{try} could look like.

5.1 Motivation

Firstly, note that, although we do not prove this, we think that it is unlikely that λ^{try} could be mapped to the normal Hindley-Milner. Recall that terms are given by the following syntax:

$$M, N ::= x \mid \lambda x.M \mid MN \mid \textit{let } x = M \textit{ in } N$$

If we were to translate λ^{try} to this, a most obvious question is how we would deal with the term *throw* $n(M)$ in the presence of the *(throw)* reduction and type assignment rules.

Assume that the translation of the term *throw* $n(M)$ was some term N in Hindley-Milner. If we were to translate application in λ^{try} as application in Hindley-Milner (which seems to be the most natural option), then, in order to preserve the notion of reduction and because of the (*throw*) reduction rule, N would have to satisfy $NP \to N$ for at least some P (that would also be produced by the translation). This would imply, by the Hindley-Milner type assignment rules and subject reduction, that N can be assigned both of the types $A \to B$ and B (for some A, which would be the type of P, and some B).

Furthermore, the type assignment rule (*throw*) allows *throw* $n(M)$ to be assigned any type at all (provided that M is typeable and there is a suitable name in the context). If we aim to preserve type assignment (which is desirable), this implies that we also need to be able to assign any type at all (in the right conditions) to N.

These observations suggest that N would have to be a polymorphic term to which we can assign any type, which is to say that we can assign the type $\forall\varphi.\varphi$ to it. Looking at our terms and type assignment rules in Hindley-Milner, however, there is no obvious candidate for such a term.

At this point, we are stuck. It is worth considering whether a popular extension to Hindley-Milner, which deals with recursion, could help. As presented by van Bakel in [6], it sees the addition of *fix* $g.M$ to the syntax of terms, with the reduction rule

$$\textit{fix}\ g.M \to M[(\textit{fix}\ g.M)/g]$$

and the type assignment rule

$$\frac{\Gamma, g : A \vdash M : A}{\Gamma \vdash \textit{fix}\ g.M : A}\ (fix)$$

This helps express recursive functions. An informal example would be the factorial function, F, which could be defined as *fix fac*.λn. *if* $(n = 0)$ *then* 1 *else* $n \times$ *fac* $(n-1)$. Then, for example, it is easy to show that $F\ 5$ reduces to $5 \times F\ 4$ and $F\ 0$ reduces to 1, conforming to the mathematical definition of the factorial.

More importantly, this provides a polymorphic term that we can assign any type to: *fix* $g.g$. Note:

$$\cfrac{\cfrac{\cfrac{}{\Gamma, g : \varphi \vdash g : \varphi}\ (Ax)}{\Gamma \vdash \textit{fix}\ g.g : \varphi}\ (fix)}{\Gamma \vdash \textit{fix}\ g.g : \forall\varphi.\varphi}\ (\forall I)$$

Could the translation of the *throw* be a term with such a form? Unfortunately, just using *fix* $g.g$ does not work. While it is true that we can assign any type to it, it is not so useful with regards to reduction, as it will just keep reducing to itself forever. If our previous N would be this term, then, instead of the desirable $NP \to N$, we would have $NP \to NP \to \ldots$

Another approach could be representing the context consuming nature of the throw with a term such as $fix\ g.\lambda x.g$. In this case, if this was our N, we would have that $NP \to (\lambda x.N)P \to N$, which satisfies the (*throw*) rule. However, this term is still problematic: we can no longer assign all types to it, since any type we could assign would have to include an arrow type (because of the lambda). Furthermore, even if the (*throw*) rule is satisfied, dealing with the *(try)* reduction rules seems harder: how would one encode the name and payload of a throw, and how would one "catch" a term like $fix\ g.\lambda x.g$?

The answer to these questions is not obvious, and, even with the recursion extension, it seems that it is hard to translate λ^{try} (at least in the presence of the assumption that application in λ^{try} is translated as application in Hindley-Milner - but what else to translate it as is, again, not obvious). To address this, we further introduce some additional extensions and discuss their feasibility.

5.2 Following the Evaluator

We have seen in Chapter 3 that there might be some merit to the idea of encoding λ^{try} using the Either data type from Haskell.

This suggests that a good first attempt in our search could be to formalise our evaluator into a target language for the translation, where this target language should be an extension to Hindley-Milner.

The most important features introduced by the evaluator are the *Left* and *Right* terms, together with the ability to perform a case deconstruction on them. As shown in the evaluator, this provides an answer to the question (form the previous section) of what λ^{try} application should be translated as: with these features, it could be translated as a case construct which first "evaluates" the first term in the application and then, if it is a throw, provides special treatment for it. Furthermore, the *try* construct can be translated in a similar fashion.

This seems to suggest a first set of extensions that our target language should have. Starting from Hindley-Milner, we add to the syntax of terms

$$M, N ::= \ldots \mid \textit{Left}\ (M) \mid \textit{Right}\ (M) \mid \textit{case}\ M\ \textit{of}\ \begin{cases} \textit{Left}\ x \to M_L \\ \textit{Right}\ x \to M_R \end{cases}$$

Left and *Right* are used as syntactic markers. With regards to the *case* statement, note that the arrows have nothing to do with reduction rules: they are simply part of the syntax (this is similar to the Haskell syntax for the *case* statement, and a similar approach can also be found in System FC [14]). The words *Left* and *Right* are fixed, x is a variable, and M_L and M_R are other terms produced by the syntax (the subscripts are used in order to avoid giving the impression that the terms have to be the same, or identical to the M immediately after the *case* keyword - this is similar to the possible use of subscripts when defining application as M_1M_2, rather than MM). We consider the x to be *bound* in M_L and M_R.

The additions to the reduction rules are

$$\begin{gathered} \textit{case Left}\ (M)\ \textit{of}\ \begin{cases} \textit{Left}\ x \to M_L \\ \textit{Right}\ x \to M_R \end{cases} \to M_L[M/x] \\ \textit{case Right}\ (M)\ \textit{of}\ \begin{cases} \textit{Left}\ x \to M_L \\ \textit{Right}\ x \to M_R \end{cases} \to M_R[M/x] \\ M \to N \Rightarrow \textit{case}\ M\ \textit{of}\ \cdots \to \textit{case}\ N\ \textit{of}\ \ldots \end{gathered}$$

These rules say that, when dealing with the *case* construct, say *case* M *of* $\ldots$, we are allowed to reduce the M as much as we wish - if such reduction leads to a term of the form *Left* (P) or *Right* (P), then the *case* statement and the syntactic markers are escaped and reduction follows the corresponding branch. The result of reducing M is not lost: P is available to each branch in the form of the variable x, which it substitutes.

The syntax for types is also extended with the construct

$$A, B ::= ... \mid \textit{Either } A\ B$$

and we add the following type assignment rules:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \textit{Left } (M) : \textit{Either } A\ B}\ (Left) \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \textit{Right } (M) : \textit{Either } A\ B}\ (Right)$$

$$\frac{\Gamma \vdash M : \textit{Either } A\ B \quad \Gamma \vdash \lambda x.M_L : A \to C \quad \Gamma \vdash \lambda x.M_R : B \to C}{\Gamma \vdash \textit{case } M \textit{ of } \begin{cases} \textit{Left } x \to M_L \\ \textit{Right } x \to M_R \end{cases} : C}\ (case)$$

These rules allow the introduction of the *Either* type (for *Left* and *Right* terms, provided the encased term can also be typed), and its elimination (in order to type a *case* construct, M needs to have an *Either* type, and each branch must take its corresponding type from the *Either* and produce the same type, C).

What we have defined here is a variant of the extension for the *disjoint union* (or *sum*) type constructor, which is also presented by van Bakel in [6], on page 48. He gives the following extensions to the syntax of λ-terms:

$$E ::= \ldots \mid \textit{injl } (E) \mid \textit{injr } (E) \mid \textit{case } (E_1, E_2, E_3)$$

The new reduction rules are

$$\begin{array}{c} \textit{case } (\textit{injl } (E_1), E_2, E_3) \to E_2 E_1 \\ \textit{case } (\textit{injr } (E_1), E_2, E_3) \to E_3 E_1 \end{array}$$

The syntax of types is extended with

$$A, B ::= \ldots \mid A + B$$

and the new type assignment rules are:

$$\frac{\Gamma \vdash E : A}{\Gamma \vdash \textit{injl } (E) : A + B}\ (injl) \qquad \frac{\Gamma \vdash E : B}{\Gamma \vdash \textit{injr } (M) : A + B}\ (injr)$$

$$\frac{\Gamma \vdash E_1 : A + B \quad \Gamma \vdash E_2 : A \to C \quad \Gamma \vdash E_3 : B \to C}{\Gamma \vdash \textit{case } (E_1, E_2, E_3) : C}\ (case)$$

Our extension and the extension given by van Bakel [6] are very similar. To see this, all we need to do (informally) is map *Left* (M) to *injl* (M), *Right* (M) to *injr* (M), $\textit{case } M \textit{ of } \begin{cases} \textit{Left } x \to M_L \\ \textit{Right } x \to M_R \end{cases}$ to *case* $(M, \lambda x.M_L, \lambda x.M_R)$ and *Either* $A\ B$ to $A + B$. Then, the reduction rules and type assignment rules match completely, except for the fact that, in [6], van Bakel does not allow reducing under the *case* construct (meaning that there is no rule of the form $M \to M' \Rightarrow$ *case* $(M, N, P) \to$ *case* (M', N, P), while in our case, there is).

These extensions are enough to explore how our translation could look like. We would like it to have two properties:

1. The translation of any λ^{try}-term should reduce to a term of the form *Left* (M) or *Right* (M). This is to be consistent with the evaluator, which always produces a *Left* or a *Right*.
2. The translation should be proper, in the sense that when translating a term, the translation should also be passed down recursively to all of its subterms. This is a key difference from the evaluator: unlike in that case, this time λ^{try} is not part of our target language.

   For example, in the case of our evaluator, when evaluating $\lambda x.M$, it would simply produce *Right* $(\lambda x.M)$, without touching the M. This was acceptable, because λ^{try} was part of the return type and the types would match. For our translation, however, this would not work, as it would not be well-typed: instead, we would have to translate the M as well. The same idea applies to all λ^{try}-terms.

   A consequence of this is that whatever evaluation steps the evaluator took before, we would have to represent via reduction rules in our extended system. Then, a full call to the evaluator would correspond to a reduction sequence that ends in a term as described in property 1.

With these properties in mind, we can start translating. Denote the translation with $[\cdot]$. Then we may reasonably have that

$$[x] = \textit{Right}\ (x)$$

as we must produce a *Left* or a *Right* (and we would like to reserve the *Left*'s for throws, as in the evaluator). Further, it also seems that the only reasonable option for the λ-abstraction is to have

$$[\lambda x.M] = \textit{Right}\ (\lambda x.[M])$$

Going further, it is time to deal with application. Assume for now that we translate *throw* $n(M)$ as *Left* (N) for some N. Then the reasonable approach for application would be:

$$[MN] = \textit{case}\ [M]\ \textit{of}\ \begin{cases} \textit{Left}\ x \to \textit{Left}\ x \\ \textit{Right}\ x \to\ ? \end{cases}$$

The translation above means that we reduce $[M]$ until it becomes a *Left* or a *Right*, and then we act accordingly. If we get a *Left*, then that corresponds to a throw, so we simply leave it as it is and we ignore N, in the spirit of the $(\textit{throw})$ reduction rule. However, what should we do if we obtain a *Right*? Even with only the translations we have so far, we see a problem.

If $[M]$ reduces to something of the form *Right* $(\lambda x.P)$, then we want to apply $\lambda x.P$ to the translation of N, in order to fulfill the (β) reduction rule. This would suggest replacing the question mark with $x[N]$ (note that this is normal application in the extended system, which can be straightforwardly defined starting from Hindley-Milner). Importantly, note that we can not say $[xN]$ (which seems similar to the evaluator approach), because the translation would recurse infinitely, and even if we were to assume that x would be somehow substituted with a term in the target language before the translation being applied, the types would not match (we would be applying a translated term to an untranslated one). Continuing, then, we would obtain $(\lambda x.P)[N]$, which should reduce to a *Left* or a *Right*.

On the other hand, if M reduces to something of the form *Right* (x), the approach above would not work: after reducing the *case*, we will be left with $x[N]$, which does not reduce any further, and so does not reduce to a *Left* or a *Right*. Instead, it is clear that in this case we should be able to tell that there is nothing more to be done and we should add the *Right* ourselves: this corresponds to replacing the question mark with *Right* $(x[N])$. However, this approach does not work for λ-abstractions: note that, by adding the *Right* ourselves, we would enclose the application in it, but we can not reduce under *Right*, so no further reduction will take place (where it clearly should, according to the (β) rule). Furthermore, even if we were to add such reduction under *Right*, what if the application inside reduces to a *Left*? Should we treat *Right* $(\textit{Left}\ \dots)$ as a throw or not?

Instead, what emerges from here is that the ability to discriminate terms according to *Left* of *Right* is not enough. There are clearly different actions to be taken depending on whether a term reduces to a variable or a λ-abstraction, despite the fact that both reside under a *Right*. Therefore, this suggests the need for a *case* statement that can discriminate according to more general term structures, which could potentially eliminate the need for *Left* and *Right* altogether. This is what we present in the next section.

5.3 Extensions

This section combines the evaluator in Chapter 3 with the observations in the previous section and introduces a more general *case* construct, as well as any other constructs from the evaluator that need a formal definition. Call the extended system $HM+$. The extensions are as follows:

$$M, N ::= \ldots \mid \textit{Thr}\ (M, n) \mid \textit{Stuck}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i}$$
$$\mid \textit{case}\ M\ \textit{of} \begin{cases} \textit{Thr}\ (x_t, n_t) \to M_t \\ \textit{Abs}\ x_a \to M_a \\ \textit{VarApp}\ x_v \to M_v \\ \textit{StuckApp}\ x_s \to M_s \end{cases}$$
$$\mid \textit{if}\ B\ \textit{then}\ M\ \textit{else}\ N \mid \textit{lookup}\ n\ \textit{in}\ \overline{\textit{catch}\ n_i(x) = N_i}$$

$$B ::= n \in \overline{n_i} \mid \overline{n_i} \in M$$

Figure 5.1: $HM+$ syntax extensions

5.3.1 To the syntax of terms

The additions to the syntax can be found in figure 5.1.

We extend the syntax of terms with a set of names ranged over by $n,\ m,\ \ldots$

For simplicity, we also add special variables x_t, x_a, x_v, x_s (which are specific to this system and *do not appear in* λ^{try}) and the name n_t (which similarly *does not appear in* λ^{try}). All of these are only allowed to appear within the *case* construct.

The new constructs are heavily inspired by the Haskell evaluator in Chapter 3 and are explained below:

1. *case* M *of* $\begin{cases} \textit{Thr}\ (x_t, n_t) \to M_t \\ \textit{Abs}\ x_a \to M_a \\ \textit{VarApp}\ x_v \to M_v \\ \textit{StuckApp}\ x_s \to M_s \end{cases}$:

   This is the most important addition, so we explain it first. With regards to syntax, it is closely related to the simpler *case* construct we saw in section 5.2. In each of the four branches, everything on the left hand side of the arrow is fixed (including the special variables x_t, x_a, x_v, x_s and the special name n_t that we have already mentioned, which do not appear in λ^{try}). Note that the things on the left hand side of the arrows are *not* terms (despite the fact that *Thr* (x_t, n_t) looks like one) - they are just part of the syntax for the *case* construct. The arrows themselves are part of the syntax and have no relation to reduction rules, while the terms M_t, M_a, M_v and M_s represent other terms produced by the syntax (they are differentiated with a subscript, in order to not give the impression that they have to be the same, or perhaps identical to the M immediately after the *case* keyword). We consider x_t and n_t to be bound in M_t, x_a to be bound in M_a, and similarly for the last two branches.

   The motivation for the form of this construct stems from the observations in section 5.2: if we try to force the translations of λ^{try} terms into *Left*'s and

Right's, we saw that we can not define a proper translation - specifically, we could not translate application in a way that would guarantee the production of a *Left* or a *Right*.

Consider the problem of translating the application MN. In section 5.2, we tried to reduce the translation of M until a *Left* or *Right* was reached, but we came to the conclusion that this was not enough: as such, we deduced that the *case* construct needed more specific information about the structure of the term that the translation of M reduced to. Knowing it was a *Right* was not enough: we actually needed to distinguish between, for example, a variable or an abstraction.

This leads to the question: what *termination cases* does the *case* statement need to be aware of? The reason for this is the same as before: we need to be able to tell when the translation of M has reduced to something that represents a throw, and when it has not (however, most importantly, forcing the *Left* and *Right* divide does not work, as already argued).

If simplifying matters with *Left* and *Right* is not good enough, perhaps we should look at what λ^{try} terms reduce to as they are: if reduction ever finishes, then the final result is a term in normal form. If we can identify the structure of such terms, it would give us a good (sufficient) candidate for what our *case* structure needs to consider: we could make it distinguish between final reduction results of the translations of λ^{try} terms in normal form. This would definitely include differentiating between variables and abstractions, which would solve our previous problem.

Taking into consideration the reduction rules of λ^{try}, we state (but do not prove) that λ^{try} terms that are in normal form (with respect to call-by-name reduction), denoted by $HLTry$, are given by the following syntax:

$$HLTry ::= \textit{throw}\ n(N) \mid HLTry'$$
$$HLTry' ::= xM_1...M_n \mid \lambda x.M \mid (\textit{try}\ HLTry';\ \textit{CB})M_1...M_n$$

where $n \geq 0$ ($n = 0$ implies there is no application), the M's are any λ^{try} terms, `CB` stands for the catch block and $HLTry'$ in the last case must contain at least one of the names defined in the catch block.

We can now think about what we would like the translation of an $HLTry$ term to reduce to, and make our *case* be able to distinguish between all the possible cases. The solution that we have identified is as follows:

(a) *throw* $n(N)$: We have already seen that there is no easy way to translate this term without some extension to the syntax. Thus, we add the *Thr* term to the syntax for this specific purpose, and this is the first branch of our *case* construct. With regards to x_t and n_t, they act in a similar manner to the x in the previous case construct from section 5.2: if the translation of the first term in the application reduces to a *Thr* (M, n), they make sure that the M and n are available to the term M_t (M will substitute x_t, and n will substitute n_t).

(b) $\lambda x.M$: It is not unreasonable to expect the translation of this to also be an abstraction. Therefore, we make the abstraction the second branch in the construct (which we denote by *Abs*). Note that the word *Abs* is fixed and does not appear anywhere else in the syntax: it only acts as a marker for this branch. Thus, it is not yet clear what the variable x_a does, as there is no obvious "pattern matching" happening anywhere (as was the case for the *Thr* construct). Instead, since we still need to use the result of the reduction on the right hand side of the arrow (in order to have preservation of reduction), we will see that it suffices if *the whole abstraction* substitutes x_a (because we only care that it is an abstraction, but not about specific information in it, like, for example, what variable is being abstracted on).

The exact operational workings (which is the place where the actual recognising of the structure of terms takes place) will be made clear once we give the reduction rules. So far, we are only justifying the structure of the *case* construct.

(c) $xM_1...M_n$: It would be satisfying if the translation of this term would eventually reduce (if we denote the translation by square brackets) to $x[M_1]...[M_n]$. As we will see, this is possible. This corresponds to the third branch in our construct, denoted by *VarApp* (because it is a variable potentially followed by a series of applications). The variable x_v behaves exactly the same as the x_a in the previous branch.

(d) (*try* $HLTry'$; *CB*)$M_1...M_n$: This case is less obvious, as we have not tackled the translation of the *try* yet. If we translate a *try* that can no longer reduce, we would like the translation itself to reduce to something that is in normal form and has a recognisable structure for the *case* construct (like the other translations so far). As it is not obvious what this should be, we introduce a new term to the syntax for this specific purpose: the *Stuck* term. This denotes a *try* that is *stuck* and can not further reduce, and is easy for the *case* statement to recognise.

This being said, we will have to make sure that the translation of such a *try* reduces to a *Stuck*. The whole purpose of the *Stuck* is to mark the last type of λ^{try} term in normal form, which we have no obvious way to distinguish. With the addition of the *Stuck*, it would be desirable for the translation of (*try* $HLTry'$; *CB*)$M_1...M_n$ to eventually reduce to (if translation is denoted by square brackets) (*Stuck* P; *CB*$'$)$[M_1]...[M_n]$ for some P and *CB*$'$, which we will see is possible. Finally, this gives the last branch in our *case* statement, which is denoted by *StuckApp* (because it is a *Stuck* term potentially followed by a series of applications). The x_s variable behaves similarly to the previous 2 branches.

We will see that the 4 branches we have identified are enough to model all of the reduction rules introduced by λ^{try}, and both application and *try* terms will be translated to a *case* construct in our system. They will both reduce the translation of a term until it corresponds to one of the four branches above (which we will prove always happens), and the branch will then be followed. In practical terms, we only care if reduction finishes with a *Thr* or not (and indeed the last 3 branches in the *case* construct will always do the same thing in our translation), but, in order to be able to tell this, we needed to identify what else reduction could finish with (which yielded the 4 branches above). Had we not done this, it would not have been possible to formulate proper reduction rules (that we will see), because it is hard to design a rule that says "in all other cases, do this" - at what point will we definitely know that we are not dealing with a throw?

Something worth noting is also the fact that there is no need for *Left* and *Right* anywhere in the syntax. Despite the fact that we tried to use them to achieve the same purpose (namely, figuring out when we are dealing with a throw and when we are not), it was not possible to do so without further considering the structure of the terms - and once we consider the structure of the terms, which is the current approach, we already have enough power to represent the λ^{try} reduction rules.

2. *Thr* (M, n): This represents the result of translating a *throw* and can be seen as the equivalent of a *Left* term from the evaluator. The pair consists of the term to be passed to the handler and the name of the handler. The role of this term is explained together with the *case* construct.

3. *Stuck* $M;\ \overline{\textit{catch}\ n_i(x) = N_i}$: This is used to mark a *try* term that can no longer participate in reduction; we would like the translation of such a term to reduce to this. The reason for having this term is explained together with the *case* construct. Note that we could have used any other term instead of this, as long as it is recognisable by the *case* statement and it does not participate in any reduction (other than in the *case* statement) - for example, a special constant defined specifically for this purpose. We have chosen to keep the term M and the catch block for informative purposes.

4. *if* B *then* M *else* N: This is the usual if statement - in our case, it will only be used with tests that check whether a name n occurs in a set of names or any name in a set of names occurs in a $HM+$ term. The reason for having this term is that, when translating *try*, we will need to reduce to different things depending on whether a name occurs somewhere: the most obvious way to achieve this is to introduce a dedicated boolean construct such as this, and

$$(I_1) \quad \textit{if } n \in \overline{n_i} \textit{ then } M \textit{ else } N \to M, \text{ if } n \in \{n_1, \ldots, n_n\}$$
$$(I_2) \quad \textit{if } n \in \overline{n_i} \textit{ then } M \textit{ else } N \to N, \text{ if } n \notin \{n_1, \ldots, n_n\}$$
$$(I_3) \quad \textit{if } \overline{n_i} \in M \textit{ then } N \textit{ else } P \to N, \text{ if any } n_i \in M$$
$$(I_4) \quad \textit{if } \overline{n_i} \in M \textit{ then } N \textit{ else } P \to P, \text{ if no } n_i \in M$$

$$(C_1) \quad (\textit{case Thr } (M, m) \textit{ of Thr } (x_t, n_t) \to M_t;\ \ldots) \to M_t[M/x_t][m/n_t]$$
$$(C_2) \quad (\textit{case } \lambda y.M \textit{ of } \ldots;\ \textit{Abs } x_a \to M_a;\ \ldots) \to M_a[\lambda y.M/x_a]$$
$$(C_3) \quad (\textit{case } yM_1 ... M_n \textit{ of } \ldots;\ \textit{VarApp } x_v \to M_v;\ \ldots) \to M_v[yM_1 ... M_n/x_v]$$
$$(C_4) \quad (\textit{case } (\textit{Stuck } M;\ \overline{\textit{catch } n_i(x) = N_i})M_1 ... M_n \textit{ of } \ldots;\ \textit{StuckApp } x_s \to M_s) \to M_s[(\textit{Stuck } M;\ \overline{\textit{catch } n_i(x) = N_i})M_1 ... M_n/x_s]$$

$$(L_1) \quad \textit{lookup } n \textit{ in } \overline{\textit{catch } n_i(x) = N_i};\ \textit{catch } m(x) = M \to \lambda x.M, \text{ if } n = m$$
$$(L_2) \quad \textit{lookup } n \textit{ in } \overline{\textit{catch } n_i(x) = N_i};\ \textit{catch } m(x) = M \to \textit{lookup } n \textit{ in } \overline{\textit{catch } n_i(x) = N_i}, \text{ if } n \neq m$$
$$(L_3) \quad \textit{lookup } n \textit{ in catch } m(x) = M \to \lambda x.M, \text{ if } n = m$$

$$(\textit{context}) \quad M \to N \Rightarrow \textit{case } M \textit{ of } \cdots \to \textit{case } N \textit{ of } \ldots$$

Figure 5.2: $HM+$ reduction rules extensions

have reduction rules for it that take into account name occurrence (in λ^{try}, this was part of the reduction rules for *try*, but we do not have that here, as the rules for the *case* statement will be more general and it will be unaware of the fact that it is the translation of a *try*).

5. *lookup* n *in* $\overline{\textit{catch } n_i(x) = N_i}$: This term is used to formalise the looking up of a name in a catch block in order to retrieve the corresponding handler. Again, in λ^{try}, this was part of the reduction rules for *try* - however, our *case* statement will not be aware of the fact that it is the translation of a *try* and it has no concept of handler lookup. As such, we have to introduce this construct, with dedicated reduction rules, to model it.

5.3.2 To reduction rules

The extensions to the reduction rules can be found in figure 5.2.

We use square brackets for substitution in $HM+$. We do not define it formally, but instead give a straightforward informal definition: substitution is defined similarly to λ^{try}, where all of the new constructs (including booleans) simply pass the substitution on to the subterms inside (including booleans). The *Stuck* and *lookup* terms are treated similarly to the *try* term in λ^{try} with regards to bound and free variables and names. The x's (and the n_t) in the *case* term act as binders for the terms on the right hand side of the *case* arrows (like the x in $\lambda x.M$ for the M), and note that, by construction, they can not appear free anywhere in our system. We assume Barendregt's Convention and the fact that α-conversion takes place silently wherever necessary.

We use the same notation to define *name substitution*. *This is only defined where the name being substituted is* n_t, and is denoted by $M[m/n_t]$ for some name m, meaning that m replaces n_t in M. As before, all of the constructs pass this substitution on to the subterms inside (taking into account that n_t acts as a binder, which is also mentioned above) - the only places in which it has a meaningful effect are *Thr* (M, n), the names in the boolean tests, and the n in the *lookup* construct (in which any free occurrence of the name n_t would be replaced by m given the substitution $[m/n_t]$). Note that we could have defined substitution in a more general way (with the ability to substitute any name), but, as we will see, the current definition suffices for our translation. Furthermore, this has certain advantages, such as not having to worry about Barendregt's Convention and α-conversion in this case (this is because, as will be made clear by the translation, the name n_t will never occur free in the translation of any λ^{try}-term).

We assume the CBN reduction strategy throughout. Note that we keep the normal (β) reduction rule, which, in $HM+$, uses the notion of variable substitution as we have defined it above.

The extensions are explained below:

If construct

The meaning of these rules is straightforward: if the test is true, we reduce to the first term, and if it is false, we reduce to the second one. Of course, we assume that there is some procedure to check whether a name appears in a set of names or any member of a set of names appears in a $HM+$ term. This approach is similar to the one in the reduction rules for *try* terms in λ^{try} (which are also conditioned by name occurrence tests), and occurrence can easily be checked in our system: when checking if a name appears in a term, all of the constructs pass the check on to the subterms inside, taking into account any new occurrences introduced by themselves (such as the name in *Thr* or the names in the boolean tests).

Case construct

There are 4 rules for the case construct, one corresponding to each branch (we only show the relevant branch in each rule). The last two rules have $n \geq 0$, where $n = 0$ implies there is no application at all. We also extend contexts to allow reduction under the *case* statement (this is the (*context*) rule).

These rules make clear exactly how the *case* statement operates and are consistent with the explanations given in subsection 5.3.1. They say that, when dealing with the term *case* M *of* $\ldots$, we are allowed to reduce M as much as we want, and, if such reduction leads to a term that has one of four forms, we can reduce the whole *case* construct to a term of our choosing, in which the result of the reduction of M is also available (in the form of variables that are substituted).

For example, in the first rule, if M reduces to a term of the form *Thr* (N, m) for some N and m, then the *case* construct reduces to the term M_t, in which N substitutes x_t and m substitutes n_t. This allows us to provide special treatment for the throw. Note also that, in a sense, the *Thr* is deconstructed, as the pieces of information contained in it (N and m) are available separately. As explained in 5.3.1, this is not the case for the other 3 branches: if reducing M leads to one of their corresponding forms, then the result is not deconstructed in any way, but rather it is used as a whole in the terms on the right hand side of the arrows (M_a, M_v and M_s). This is because we do not care about specific information in it - we are only interested in the fact that a specific form has been reached (this suffices for the translation).

It is important to clarify the meaning of n in the reduction rules: it signifies the number of applications that a term is followed by (for example, in $yM_1...M_n$). In this sense, one could argue that, in fact, the third and fourth *case* rules each stand for an infinity of rules, one corresponding to each n from 0 to infinity. However, this is not our intention: the terms following the keyword *case* are meant in a "pattern matching" way, meaning that the rule is to be applied if M in *case* M *of* $\ldots$ has that specific structure (note that a term M can not satisfy more than one of the patterns in the rules). For the third rule, for example, we mean that the rule is to be applied if M consists of any variable potentially followed by any number of applications - this form is captured by the rule and is available for use on the right hand side of the reduction arrow, where we use it in the substitution. The "pattern matching" reasoning is similar for all four branches. Thus, implicitly, our system has the capability to perform a form of pattern matching on its terms, and it can distinguish the number of applications in an application chain.

Lookup construct

The rules model the looking up of a name in a context. They serve the only purpose of guaranteeing that, if n is defined in the catch block, then

$$\textit{lookup } n \textit{ in } \overline{\textit{catch } n_i(x) = N_i} \rightarrow^* \lambda x.N_j$$

where n_j is the unique name defined in the catch block that satisfies $n_j = n$. Denote this fact by (L). This represents the looking up of a handler in a list of handlers. Note that the name that is being looked up does not have to be defined in the catch block: if this is the case, then reduction simply does not proceed further and we are stuck with the *lookup* term (however, in our translation, we will use the *if* construct to make sure that we only ever do a lookup where the name is defined).

Notes

Importantly, note that the forms that are being considered in the reduction rules for the *case* construct (the terms following the word *case*) are all in normal form with respect to reduction in $HM+$. Thus, there is no ambiguity - once one of those forms has been reached (and not earlier!), exactly one of the rules will be able to be applied (and no other rules, since we can not reduce any further inside the *case*). The rules for the *if* construct and the *lookup* construct also do not introduce ambiguity, so we can safely conclude that the same property holds for the whole of the reduction system: at any point in the reduction of a term M, we can apply **at most one** reduction rule (in other words, reduction paths can not split). This can be proven formally by induction on the structure of contexts.

Also note that there are deliberately no reduction rules for the *Stuck* construct (other than within a *case*). This is because it is only used as a marker for a λ^{try} *try* term that can no longer reduce: it is easily recognisable (as a pattern) and in normal form, as explained in 5.3.1.

This concludes the additions to the reduction rules.

5.4 Translation

We now give the translation from λ^{try} to $HM+$, which we denote by $\lfloor\cdot\rceil$.

$$\lfloor x\rceil = x$$
$$\lfloor \lambda x.M\rceil = \lambda x.\lfloor M\rceil$$

$$\lfloor MN\rceil = \textit{case}\ \lfloor M\rceil\ \textit{of}\ \begin{cases} \textit{Thr}\ (x_t, n_t) \to \textit{Thr}\ (x_t, n_t) \\ \textit{Abs}\ x_a \to x_a\lfloor N\rceil \\ \textit{VarApp}\ x_v \to x_v\lfloor N\rceil \\ \textit{StuckApp}\ x_s \to x_s\lfloor N\rceil \end{cases}$$

$$\lfloor \textit{throw}\ n(N)\rceil = \textit{Thr}(\lfloor N\rceil, n)$$

$$\lfloor \textit{try}\ M;\ \overline{\textit{catch}\ n_i(x) = N_i}\rceil =$$
$$\textit{case}\ \lfloor M\rceil\ \textit{of}\ \begin{cases} \textit{Thr}\ (x_t, n_t) \to \textit{if}\ n_t \in \overline{n_i}\ \textit{then}\ (\textit{lookup}\ n_t\ \textit{in}\ \overline{\textit{catch}\ n_i(x) = \lfloor N_i\rceil})x_t\ \textit{else Thr}\ (x_t, n_t) \\ \textit{Abs}\ x_a \to \textit{if}\ \overline{n_i} \in x_a\ \textit{then Stuck}\ x_a;\ \overline{\textit{catch}\ n_i(x) = \lfloor N_i\rceil}\ \textit{else}\ x_a \\ \textit{VarApp}\ x_v \to \textit{if}\ \overline{n_i} \in x_v\ \textit{then Stuck}\ x_v;\ \overline{\textit{catch}\ n_i(x) = \lfloor N_i\rceil}\ \textit{else}\ x_v \\ \textit{StuckApp}\ x_s \to \textit{if}\ \overline{n_i} \in x_s\ \textit{then Stuck}\ x_s;\ \overline{\textit{catch}\ n_i(x) = \lfloor N_i\rceil}\ \textit{else}\ x_s \end{cases}$$

Although inspired by our previous evaluator, there is a very important difference between it and our translation: unlike the evaluator, which only evaluates terms when needed, the translation is applied to everything from the beginning (note that it is passed down recursively onto each subterm). Thus, like the translation from λ^{try} to $\lambda\mu$, it is a translation in the proper sense of the word: the constructs introduced by λ^{try} do not appear anywhere in the translation, other than the fact that we have chosen to keep the same appearance for the catch block (the same can not be said about the evaluator, where λ^{try} was part of the target language).

Another important observation is that the RHS's of the last 3 branches of each *case* construct in the translation are practically identical. As we have already mentioned in 5.3.1, we only need to give special treatment to the *throw* terms: the other branches serve more as termination criteria, as once we have reached a term of one of the forms that they define, we definitely know that we are not dealing with a throw (and, knowing this information, we proceed in the same way for all three of them). The *case* construct makes sure we reduce the argument to the *case* until there is no ambiguity left about which rule to apply.

The non-trivial cases are application and *try* terms:

1. MN: The intuition behind the translation of MN is as follows: as we have already explained in previous sections, we need to reduce $[M]$ until we know whether we are dealing with a throw or not. To this extent, we make $[M]$ the argument of a case construct, which accomplishes just that. Then, if we end up with a *Thr*, we simply ignore $[N]$ and "return" the *Thr*. This corresponds to the (*throw*) reduction rule from λ^{try}. If we end up with something that is not a throw, we apply the result to $[N]$, which, in the case of the abstraction, will correspond to the (β) rule (and in the other two cases, reduction of the term on the right hand side of the arrow after substituting x_v or x_s will simply not be able to proceed further).

2. *try* $M;\ \overline{\textit{catch}\ n_i(x) = N_i}$: Similarly, when translating this term, we need to reduce $[M]$ until we know whether we are dealing with a throw or not.

   If, in the *case* statement, we end up with a *Thr* (first branch), then we check if it can be caught by the current *try* (or *case*) block (this is the purpose of the *if* statement). If it can, then we simply use the *lookup* construct to find the relevant handler, and we apply the payload of the *Thr* to it. If it can not, we just "return" the *Thr* itself, which symbolizes upward propagation of it (until the proper *case* block can catch it).

   On the other hand, if we end up with something that is not a throw, we have to check whether the *try* (or *case*) block can be escaped. λ^{try} rules tell us that this is only possible if none of the names defined in the catch block appears in the result, and this is exactly what the *if* statements in the last 3 branches test. If no such occurrence exists, we are free to "return" the result of reducing $[M]$. If any name does, however, occur even after we have reduced $[M]$ as much as we could, then the *try* block can not be escaped. As we have explained previously, the solution in this case is to wrap the reduction result in a *Stuck* term, whose only purpose is to act as a marker for any future *case* blocks.

As we will show, all of the reduction rules in λ^{try} are incorporated by the new *case* construct.

5.5 Examples

We present here some examples of λ^{try} terms, their translation and the complete reduction path for the translation.

1. $(\lambda x.x)(\lambda y.y)$:

$$\lfloor(\lambda x.x)(\lambda y.y)\rceil = \textit{case } \lambda x.x \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a(\lambda y.y) \\ \textit{VarApp } x_v \to x_v(\lambda y.y) \\ \textit{StuckApp } x_s \to x_s(\lambda y.y) \end{cases}$$

$\to (x_a(\lambda y.y))[(\lambda x.x)/x_a]$ (by rule (C_2))
$= (\lambda x.x)(\lambda y.y)$ (by def. subst.)
$\to \lambda y.y$ (by normal application rule (β))

Note that the *case* statement acts as a wrapper: after making sure that we are not dealing with a throw, it delegates to normal application (recall that it still exists in $HM+$, as it is an extension of Hindley-Milner, except that it uses the notion of substitution that we have extended to include the new constructs).

2. $(\textit{throw } n(x))y$:

$$\lfloor(\textit{throw } n(x))y\rceil = \textit{case Thr } (x, n) \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a y \\ \textit{VarApp } x_v \to x_v y \\ \textit{StuckApp } x_s \to x_s y \end{cases}$$

$\to (\textit{Thr } (x_t, n_t))[x/x_t][n/n_t]$ (by rule (C_1))
$= \textit{Thr } (x, n)$ (by def. subst.)

Note that the y in the application is discarded.

3. $\textit{try } (\textit{throw } n(z))y;\ \textit{catch } n(x) = x$:

$\lfloor \textit{try } (\textit{throw } n(z))y;\ \textit{catch } n(x) = x\rceil$

$$= \textit{case } \lfloor(\textit{throw } n(z))y\rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{if } n_t \in \{n\} \textit{ then } (\textit{lookup } n_t \textit{ in catch } n(x) = x)x_t \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \to \dots \\ \textit{VarApp } x_v \to \dots \\ \textit{StuckApp } x_s \to \dots \end{cases}$$

$$\to^* \textit{case Thr } (z, n) \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{if } n_t \in \{n\} \textit{ then } (\textit{lookup } n_t \textit{ in catch } n(x) = x)x_t \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \to \dots \\ \textit{VarApp } x_v \to \dots \\ \textit{StuckApp } x_s \to \dots \end{cases}$$

(as per previous example, reduction happening under the *case* here)

$\rightarrow$ (*if* $n_t \in \{n\}$ *then* (*lookup* n_t *in catch* $n(x) = x)x_t$ *else Thr* $(x_t, n_t))[z/x_t][n/n_t]$ (by rule (C_1))

$\rightarrow$ *if* $n \in \{n\}$ *then* (*lookup* n *in catch* $n(x) = x)z$ *else Thr* (z, n) (by def. subst.)

$\rightarrow$ (*lookup* n *in catch* $n(x) = x)z$ (by rule (I_1))
$\rightarrow (\lambda x.x)z$ (by (L))
$\rightarrow z$ (by (β))

Note that the name n_t was substituted by the name to which the throw occurred (all occurrences on the right hand side of the first branch arrow), and x_t was substituted by the payload of the throw, which was z.

Also note the role of the *if* test in determining whether the throw was to a name defined in the current *try* block (which, in this case, it was) and the role of the *lookup* in retrieving the handler. Normal application takes place in the end. If the throw had been to a different name, then the *else* branch of the *if* would have been followed, and the final result would have been *Thr* (z, n).

4. *try* $\lambda y.y;$ *catch* $n(x) = x$:

$\lfloor$*try* $\lambda y.y;$ *catch* $n(x) = x\rceil$

$$= \textit{case } \lambda y.y \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \rightarrow \ldots \\ \textit{Abs } x_a \rightarrow \textit{if } \{n\} \in x_a \textit{ then Stuck } x_a;\ \textit{catch } n(x) = x \textit{ else } x_a \\ \textit{VarApp } x_v \rightarrow \ldots \\ \textit{StuckApp } x_s \rightarrow \ldots \end{cases}$$

$\rightarrow$ (*if* $\{n\} \in x_a$ *then Stuck* $x_a;$ *catch* $n(x) = x$ *else* $x_a)[(\lambda y.y)/x_a]$ (by rule (C_2))

$\rightarrow$ *if* $\{n\} \in \lambda y.y$ *then Stuck* $\lambda y.y;$ *catch* $n(x) = x$ *else* $\lambda y.y$ (by def. subst.)
$\rightarrow \lambda y.y$ (by rule (I_4))

Note that x_a is replaced by the whole abstraction. As the name n does not occur in it, the reduction rule for the *if* statement says that the *else* branch should be taken, so we reduce to the abstraction itself. This corresponds to the *try* block being escaped.

If, instead of $\lambda y.y$, we had $\lambda y.$*throw* $n(z)$, a different path would have been taken. The *if* test would have checked whether the name n occurred in $\lambda y.$*Thr* (z, n) (which it does), the *then* branch would have been followed, and the final result would have been *Stuck* $\lambda y.$*Thr* $(z, n);$ *catch* $n(x) = x$. This is how the *Stuck* marker is introduced - it corresponds to a *try* that can no longer reduce, and, indeed, in λ^{try}, *try* $\lambda y.$*throw* $n(z);$ *catch* $n(x) = x$ can not reduce further, because the throw is under an abstraction.

5. xyz:

$$\lfloor xyz \rceil = \textit{case } \lfloor xy \rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a z \\ \textit{VarApp } x_v \to x_v z \\ \textit{StuckApp } x_s \to x_s z \end{cases}$$

$$= \textit{case } (\textit{case } x \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a y \\ \textit{VarApp } x_v \to x_v y \\ \textit{StuckApp } x_s \to x_s y \end{cases}) \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a z \\ \textit{VarApp } x_v \to x_v z \\ \textit{StuckApp } x_s \to x_s z \end{cases}$$

$$\to \textit{case } ((x_v)y)[x/x_v] \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a z \\ \textit{VarApp } x_v \to x_v z \\ \textit{StuckApp } x_s \to x_s z \end{cases}$$

(by rule (C_3) and reducing under the first *case* - note that the term x matches the pattern for the *VarApp* branch with $n = 0$)

$$= \textit{case } xy \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a z \\ \textit{VarApp } x_v \to x_v z \\ \textit{StuckApp } x_s \to x_s z \end{cases} \quad \text{(by def. subst.)}$$

$\to (x_v z)[(xy)/x_v]$ (by rule (C_3) - note that the term xy matches the pattern for the *VarApp* rule with $n = 1$)

$= xyz$ (by def. subst.)

Note that terms with different numbers of applications can match the same pattern $(xM_1...M_n)$, so the same rule (C_3) ends up being applied. The final reduction result is the same as the term we started with, but note that it has been passed through two *case* statements which would have detected any throws. Since they did not, they delegated back to normal application.

5.6 Preservation of Reduction

We show here that the translation presented in section 5.4 satisfies a version of reduction preservation. Consider the following syntax, which generates a subset of

$HM+$ terms that are in normal form:

$$H ::= xM_1...M_n \mid (\textit{Stuck } M;\ \overline{\textit{catch } n_i(x) = N_i})M_1...M_n \mid \lambda x.M \mid \textit{Thr } (M, n)$$

where $n \geq 0$ and $n = 0$ implies there is no application.

Theorem 5.6.1 (Translation Preserves Reduction) *If λ^{try} terms P and Q satisfy $P \to Q$ and Q has a normal form, then there is a unique $HM+$ term H given by the syntax above such that $\lfloor Q \rceil \to^* H$ and $\lfloor P \rceil \to^* H$.*

The form of this result is motivated by a key difference between λ^{try} and $HM+$: reduction paths in λ^{try} can split (for example, in the term *try* M; *CB*, if M can reduce further, but does also not contain any of the names in the catch block, then we can either further reduce M under the *try*, or first escape the *try* and reduce M outside of it), while in $HM+$, as we have already argued, they can not (M would be first reduced as much as possible before checking if it can escape the *case*). As such, our preservation result can only talk about final reduction results, not about the specific paths.

The proof is based on some auxiliary lemmas which we prove first.

We begin by showing that substitution is preserved by the translation. For clarity, we denote substitution in the original λ^{try} with curly brackets and substitution in $HM+$ with square brackets:

Lemma 5.6.2 *For any λ^{try} terms M and N and variable y, we have that $\lfloor M\{N/y\} \rceil = \lfloor M \rceil[\lfloor N \rceil/y]$.*

The proof is by induction on the structure of λ^{try} terms:

(y):
$\lfloor y\{N/y\} \rceil$
$= \lfloor N \rceil$ (def. subst. λ^{try})
$= y[\lfloor N \rfloor/y]$ (def. subst. $HM+$)
$= \lfloor y \rceil[\lfloor N \rfloor/y]$ (def. transl.)
$\square$

$(x \neq y)$:
$\lfloor x\{N/y\} \rceil$
$= \lfloor x \rceil$ (def. subst. λ^{try})
$= x$ (def. transl.)
$= x[\lfloor N \rfloor/y]$ (def. subst. $HM+$)
$= \lfloor x \rceil[\lfloor N \rfloor/y]$ (def. transl.)
$\square$

$(\lambda y.M)$:
$\lfloor (\lambda y.M)\{N/y\} \rceil$

$= \lambda y.\lfloor M \rceil$ (def. subst. λ^{try} and transl.)
$= (\lambda y.\lfloor M \rceil)[\lfloor N \rceil/y]$ (def. subt. $HM+$)
$= \lfloor \lambda y.M \rceil[\lfloor N \rceil/y]$ (def. transl)
□

$(\lambda x.M, x \neq y)$:
$\lfloor (\lambda x.M)\{N/y\} \rceil$
$= \lambda x.\lfloor M\{N/y\} \rceil$ (def. subst. λ^{try} and transl.)
$= \lambda x.\lfloor M \rceil[\lfloor N \rceil/y]$ (IH)
$= (\lambda x.\lfloor M \rceil)[\lfloor N \rceil/y]$ (def. subst. $HM+$)
$= \lfloor \lambda x.M \rceil[\lfloor N \rceil/y]$ (def. transl.)
□

(PQ):
$\lfloor (PQ)\{N/y\} \rceil$
$= \lfloor P\{N/y\}Q\{N/y\} \rceil$ (def. subst. λ^{try})

$$= \textit{case } \lfloor P\{N/y\} \rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a\lfloor Q\{N/y\} \rceil \\ \textit{VarApp } x_v \to x_v\lfloor Q\{N/y\} \rceil \\ \textit{StuckApp } x_s \to x_s\lfloor Q\{N/y\} \rceil \end{cases} \quad \text{(def. transl.)}$$

$$= \textit{case } \lfloor P \rceil[\lfloor N \rceil/y] \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a\lfloor Q \rceil[\lfloor N \rceil/y] \\ \textit{VarApp } x_v \to x_v\lfloor Q \rceil[\lfloor N \rceil/y] \\ \textit{StuckApp } x_s \to x_s\lfloor Q \rceil[\lfloor N \rceil/y] \end{cases} \quad \text{(IH)}$$

$$= \textit{case } \lfloor P \rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a\lfloor Q \rceil \\ \textit{VarApp } x_v \to x_v\lfloor Q \rceil \\ \textit{StuckApp } x_s \to x_s\lfloor Q \rceil \end{cases} [\lfloor N \rceil/y]$$ (def. subst. $HM+$, note that the binders in the branches can not be equal to y by construction)

$= \lfloor PQ \rceil[\lfloor N \rceil/y]$ (def. transl)
□

(*throw* $n(M)$):
$\lfloor$*throw* $n(M)\{N/y\} \rceil$
$= \lfloor$*throw* $n(M\{N/y\}) \rceil$ (def. subst. λ^{try})
$=$ *Thr* $(\lfloor M\{N/y\} \rceil, n)$ (def. transl.)
$=$ *Thr* $(\lfloor M \rceil[\lfloor N \rceil/y], n)$ (IH)
$=$ (*Thr* $(\lfloor M \rceil, n))[\lfloor N \rceil/y]$ (def. subst. HM+)
$= \lfloor$*throw* $n(M) \rceil[\lfloor N \rceil/y]$ (def. transl)
□

(*try* M; $\overline{\textit{catch } n_i(x) = N_i}, x \neq y$):

$\lfloor(\textit{try } M;\ \overline{\textit{catch } n_i(x) = N_i})\{N/y\}\rceil$
$= \lfloor(\textit{try } M\{N/y\};\ \overline{\textit{catch } n_i(x) = N_i\{N/y\}})\rceil$ (def. subst. λ^{try})

$$= \textit{case } \lfloor M\{N/y\}\rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{if } n_t \in \overline{n_i} \textit{ then } (\textit{lookup } n_t \textit{ in } \overline{\textit{catch } n_i(x) = \lfloor N_i\{N/y\}\rceil})x_t \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \to \textit{if } \overline{n_i} \in x_a \textit{ then Stuck } x_a;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\{N/y\}\rceil} \textit{ else } x_a \\ \textit{VarApp } x_v \to \textit{if } \overline{n_i} \in x_v \textit{ then Stuck } x_v;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\{N/y\}\rceil} \textit{ else } x_v \\ \textit{StuckApp } x_s \to \textit{if } \overline{n_i} \in x_s \textit{ then Stuck } x_s;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\{N/y\}\rceil} \textit{ else } x_s \end{cases}$$

(def. transl)

$$= \textit{case } \lfloor M\rceil[\lfloor N\rceil/y] \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{if } n_t \in \overline{n_i} \textit{ then } (\textit{lookup } n_t \textit{ in } \overline{\textit{catch } n_i(x) = \lfloor N_i\rceil[\lfloor N\rceil/y]})x_t \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \to \textit{if } \overline{n_i} \in x_a \textit{ then Stuck } x_a;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\rceil[\lfloor N\rceil/y]} \textit{ else } x_a \\ \textit{VarApp } x_v \to \textit{if } \overline{n_i} \in x_v \textit{ then Stuck } x_v;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\rceil[\lfloor N\rceil/y]} \textit{ else } x_v \\ \textit{StuckApp } x_s \to \textit{if } \overline{n_i} \in x_s \textit{ then Stuck } x_s;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\rceil[\lfloor N\rceil/y]} \textit{ else } x_s \end{cases}$$

(IH)

$$= \textit{case } \lfloor M\rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{if } n_t \in \overline{n_i} \textit{ then } (\textit{lookup } n_t \textit{ in } \overline{\textit{catch } n_i(x) = \lfloor N_i\rceil})x_t \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \to \textit{if } \overline{n_i} \in x_a \textit{ then Stuck } x_a;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\rceil} \textit{ else } x_a \\ \textit{VarApp } x_v \to \textit{if } \overline{n_i} \in x_v \textit{ then Stuck } x_v;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\rceil} \textit{ else } x_v \\ \textit{StuckApp } x_s \to \textit{if } \overline{n_i} \in x_s \textit{ then Stuck } x_s;\ \overline{\textit{catch } n_i(x) = \lfloor N_i\rceil} \textit{ else } x_s \end{cases} \quad [\lfloor N\rceil/y]$$

(def. subst $HM+$, note that the binders can not be equal to y by construction)

$= \lfloor \textit{try } M;\ \overline{\textit{catch } n_i(x) = N_i}\rceil[\lfloor N\rceil/y]$ (def. transl.)
□

In the last case, the proof for $x = y$ is similar, except that there is no substitution in the catch block, so we do not show it here. This concludes the proof of Lemma 5.6.2.

Lemma 5.6.3 *If a λ^{try} term P is in normal form, then there exists a $HM+$ term H given by the previous H syntax such that $\lfloor P\rceil \to^* H$.*

Recall the syntax of λ^{try} terms in normal form:

$$HLTry ::= \textit{throw } n(N) \mid HLTry'$$
$$HLTry' ::= xM_1...M_n \mid \lambda x.M \mid (\textit{try } HLTry';\ \textit{CB})M_1...M_n$$

where the $HLTry'$ in the last case must contain at least one of the names defined in the catch block. The proof is in two parts, one for each case in the definition of $HLTry$:

1. ($\textit{throw } n(N)$):
   Then $\lfloor \textit{throw } n(N)\rceil = \textit{Thr } (\lfloor N\rceil, n)$ and this is our H. □

2. ($HLTry'$):
   In this case the proof proceeds by induction on the structure of $HLTry'$, but we make the proposition stronger: we also require that the H that we find is not a *Thr*, and also that it contains all of the names that the original λ^{try} term contained.

We make use of the following result, denoted by (*): if a λ^{try} term M contains name n, then $\lfloor M \rceil$ also contains n (translation does not eliminate names). This can easily be proven by induction on the structure of λ^{try} terms. Then:

$(xM_1...M_n)$:
If $n = 0$, then $\lfloor x \rceil = x$ and we have found our H: x.

Otherwise, consider the case for $n = 1$: $\lfloor xM \rceil = \textit{case } x \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \rightarrow \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \rightarrow x_a \lfloor M \rceil \\ \textit{VarApp } x_v \rightarrow x_v \lfloor M \rceil \\ \textit{StuckApp } x_s \rightarrow x_s \lfloor M \rceil \end{cases}$

$\rightarrow x \lfloor M \rceil$ (by rule (C_3))

So our H is $x \lfloor M \rceil$ (note that, by (*), it contains all the names that xM contains). The pattern is clear and it is not hard to prove by induction on n that $\lfloor xM_1...M_n \rceil \rightarrow^* x \lfloor M_1 \rceil ... \lfloor M_n \rceil$, which is the H we were seeking. □

$(\lambda x.M)$:
In this case $\lfloor \lambda x.M \rceil = \lambda x. \lfloor M \rceil$ and this is our H (again by (*), note that it contains all the names the original term contains). □

$((\textit{try } HLTry';\ \textit{CB})M_1...M_n)$:
Consider first the case when $n = 0$. Then $\lfloor \textit{try } HLTry';\ \underline{\textit{CB}} \rceil$

$$= \textit{case } \lfloor HLTry' \rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \rightarrow \textit{if } n_t \in \overline{n_i} \textit{ then } (\textit{lookup } n_t \textit{ in } \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil})x_t \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \rightarrow \textit{if } \overline{n_i} \in x_a \textit{ then Stuck } x_a;\ \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil} \textit{ else } x_a \\ \textit{VarApp } x_v \rightarrow \textit{if } \overline{n_i} \in x_v \textit{ then Stuck } x_v;\ \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil} \textit{ else } x_v \\ \textit{StuckApp } x_s \rightarrow \textit{if } \overline{n_i} \in x_s \textit{ then Stuck } x_s;\ \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil} \textit{ else } x_s \end{cases}$$

$$\rightarrow^* \textit{case } H' \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \rightarrow \textit{if } n_t \in \overline{n_i} \textit{ then } (\textit{lookup } n_t \textit{ in } \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil})x_t \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \rightarrow \textit{if } \overline{n_i} \in x_a \textit{ then Stuck } x_a;\ \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil} \textit{ else } x_a \\ \textit{VarApp } x_v \rightarrow \textit{if } \overline{n_i} \in x_v \textit{ then Stuck } x_v;\ \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil} \textit{ else } x_v \\ \textit{StuckApp } x_s \rightarrow \textit{if } \overline{n_i} \in x_s \textit{ then Stuck } x_s;\ \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil} \textit{ else } x_s \end{cases}$$

for some H' that is not a *Thr* (by the IH). Note that any such H' would correspond to exactly one of the last 3 branches in the *case*, which all do the exact same thing. Thus, by an application of one of rules (C_2), (C_3) or (C_4), the above will reduce to:

$$\textit{if } \overline{n_i} \in H' \textit{ then Stuck } H';\ \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil [H'/x_b]} \textit{ else } H'$$

where x_b is one of the binders. Note that, by construction, the binders do not occur free in $\lfloor M \rceil$ for any M , so we can disregard the substitution. Also note that we know $HLTry'$ must contain some name n_i defined in the catch block - therefore, by the IH, H' also contains it, which makes the test $\overline{n_i} \in H'$ true. So by rule (I_3) the above reduces further to:

Stuck H'; $\overline{\textit{catch } n_i(x) = \lfloor N_i \rceil}$. This is the H that we were seeking (remember that the syntax for H generates a subset of the general syntax for terms M). Note that by the IH and (*), it also contains all the names that *try* $HLTry'$; *CB* contains, as required.

For the case when $n \geq 0$, the proof proceeds similarly to the case $xM_1...M_n$, except that the *StuckApp* branch in the *case* will be used instead of the *VarApp* one. It is easy to show by induction on n that: $\lfloor(\textit{try } HLTry';\ \textit{CB})M_1...M_n\rceil$ $\rightarrow^* (\textit{Stuck } H';\ \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil})\lfloor M_1\rceil...\lfloor M_n\rceil$, where H' is as in the case $n = 0$. This is the H that we were seeking, and by the IH and (*) we know that it also contains all of the names $(\textit{try } HLTry';\ \textit{CB})M_1...M_n$ contains, as required.

In each case we have found a suitable H. This completes the proof of Lemma 5.6.3.

Lemma 5.6.4 *If P and Q are λ^{try} terms such that $P \rightarrow Q$ and there is a term H given by the H syntax such that $\lfloor Q\rceil \rightarrow^* H$, then $\lfloor P\rceil \rightarrow^* H$.*

The proof is by induction on the structure of $\rightarrow$:

(β): Then $P = (\lambda x.M)N$, $Q = M\{N/x\}$ and there is a suitable H as described in the lemma. We have:

$\lfloor(\lambda x.M)N\rceil$

$$= \textit{case } \lambda x.\lfloor M\rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \rightarrow \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \rightarrow x_a\lfloor N\rceil \\ \textit{VarApp } x_v \rightarrow x_v\lfloor N\rceil \\ \textit{StuckApp } x_s \rightarrow x_s\lfloor N\rceil \end{cases} \quad \text{(def. transl)}$$

$\rightarrow (\lambda x.\lfloor M\rceil)\lfloor N\rceil$ (by rule (C_2), also recall that x_a can not appear free in $\lfloor N\rceil$)
$\rightarrow \lfloor M\rceil[\lfloor N\rceil/x]$ (reduction in $HM+$)
$= \lfloor M\{N/x\}\rceil$ (by Lemma 5.6.2)
$= \lfloor Q\rceil$ (by assumption)
$\rightarrow^* H$ (by assumption)
$\square$

(*throw*) : Then $P = (\textit{throw } n(N))M$ and $Q = \textit{throw } n(N)$. We have:

$\lfloor(\textit{throw } n(N))M\rceil$

$$= \textit{case Thr } (\lfloor N\rceil, n) \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \rightarrow \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \rightarrow x_a\lfloor M\rceil \\ \textit{VarApp } x_v \rightarrow x_v\lfloor M\rceil \\ \textit{StuckApp } x_s \rightarrow x_s\lfloor M\rceil \end{cases} \quad \text{(by def. transl.)}$$

$\rightarrow \textit{Thr } (\lfloor N\rceil, n)$ (by rule (C_1))

$= \lfloor Q \rceil$ (by assumption and def. transl.)
$\rightarrow^* H$ (by assumption)
$\square$

(*try-throw*): Then $P = \textit{try throw } n_l(N);\ \textit{CB};\ \textit{catch } n_l(x) = M_l$ and $Q = M_l\{N/x\}$. We have:

$\lfloor \textit{try throw } n_l(N);\ \textit{CB};\ \textit{catch } n_l(x) = M_l \rceil$
$= \textit{case Thr } (\lfloor N \rceil, n_l) \textit{ of}$
$\textit{Thr } (x_t, n_t) \rightarrow \textit{if } n_t \in \overline{n_i} \cup \{n_l\} \textit{ then } (\textit{lookup } n_t \textit{ in } \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil};\ \textit{catch } n_l(x) = \lfloor M_l \rceil)x_t \textit{ else Thr } (x_t, n_t); \ldots$ (by def. transl., only showing relevant branch)
$\rightarrow^* (\textit{lookup } n_l \textit{ in } \overline{\textit{catch } n_i(x) = \lfloor N_i \rceil};\ \textit{catch } n_l(x) = \lfloor M_l \rceil)\lfloor N \rceil$ (by rule (C_1) and then rule (I_1), given that $n_l \in \overline{n_i} \cup \{n_l\}$)
$\rightarrow^* (\lambda x.\lfloor M_l \rceil)\lfloor N \rceil$ (by (L))
$\rightarrow \lfloor M_l \rceil[\lfloor N \rceil/x]$ (reduction in $HM+$)
$= \lfloor M_l\{N/x\} \rceil$ (by Lemma 5.6.2)
$= \lfloor Q \rceil$ (by assumption)
$\rightarrow^* H$ (by assumption)
$\square$

(*try-normal*): then $P = \textit{try } N;\ \textit{CB}$ and $Q = N$, where $\overline{n_i \notin N}$. As explained previously, the existence of this case is what dictates the form of the result that we are trying to prove. We have:

$\lfloor \textit{try } N;\ \textit{CB} \rceil$

$$= \textit{case } \lfloor N \rceil \textit{ of} \begin{cases} \textit{Thr } (x_t, n_t) \rightarrow \textit{if } n_t \in \overline{n_i} \textit{ then } \ldots \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \rightarrow \textit{if } \overline{n_i} \in x_a \textit{ then } \ldots \textit{ else } x_a \\ \textit{VarApp } x_v \rightarrow \textit{if } \overline{n_i} \in x_v \textit{ then } \ldots \textit{ else } x_v \\ \textit{StuckApp } x_s \rightarrow \textit{if } \overline{n_i} \in x_s \textit{ then } \ldots \textit{ else } x_s \end{cases} \quad \text{(by def. transl.)}$$

$$\rightarrow^* \textit{case } H \textit{ of} \begin{cases} \textit{Thr } (x_t, n_t) \rightarrow \textit{if } n_t \in \overline{n_i} \textit{ then } \ldots \textit{ else Thr } (x_t, n_t) \\ \textit{Abs } x_a \rightarrow \textit{if } \overline{n_i} \in x_a \textit{ then } \ldots \textit{ else } x_a \\ \textit{VarApp } x_v \rightarrow \textit{if } \overline{n_i} \in x_v \textit{ then } \ldots \textit{ else } x_v \\ \textit{StuckApp } x_s \rightarrow \textit{if } \overline{n_i} \in x_s \textit{ then } \ldots \textit{ else } x_s \end{cases} \quad \text{(by assumption)}$$

We know H has one of 4 forms by assumption. We consider 2 possibilities.

1. H is a *Thr* (M, n). Then the above will reduce, by rule (C_1), to:

   $\textit{if } n \in \overline{n_i} \textit{ then } \ldots \textit{ else Thr } (M, n)$

   Note that translation does not introduce any new names (other than the n_t name binder in *case*, which we assume to not exist in λ^{try}). Since we know $\overline{n_i \notin N}$, this means that $\overline{n_i \notin \lfloor N \rceil}$.

Note also that reduction does not introduce any new names, other than through the silent α-conversion, which we assume always introduces *fresh* names.

Therefore, if the n above was equal to any n_i, because of the previous fact, it could not have been introduced by reduction, so it must have been in $\lfloor N \rceil$ from the beginning. But we also know this to be impossible (as reasoned above).

Thus, we can safely conclude that the n is not equal to any n_i, and the *else* branch will be taken, further reducing to *Thr* (M, n). But this is just H, as required. □

2. H is not a *Thr* (M, n). One of the other *case* branches will be therefore taken, the above will reduce to (using one of the rules (C_2), (C_3) or (C_4)):

   if $\overline{n_i} \in H$ *then* $\ldots$ *else* H

   However, using the same reasoning as in the previous case, we can conclude that none of the names n_i can appear in H. So, as before, the *else* branch will be taken and the above will further reduce to H, as required. □

$(M \to N \Rightarrow MR \to NR)$: This is the first inductive case. We have that $P = MR$, $Q = NR$ and $\lfloor NR \rceil \to^* H$. We assume (IH) that, if $\lfloor N \rceil \to^* H_2$, then $\lfloor M \rceil \to^* H_2$.

We know $\lfloor NR \rceil \to^* H$, which is to say that:

$$\textit{case } \lfloor N \rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a \lfloor R \rceil \\ \textit{VarApp } x_v \to x_v \lfloor R \rceil \\ \textit{StuckApp } x_s \to x_s \lfloor R \rceil \end{cases} \to^* H$$

Given the syntax that generates H, it is clear that the above is not possible unless $\lfloor N \rceil$ reduces until one of the rules (C_1) to (C_4) can be applied. Because of the direct correspondence between the rules and the H syntax, this means that there exists some H_2 such that $\lfloor N \rceil \to^* H_2$. Thus:

$$\textit{case } \lfloor N \rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a \lfloor R \rceil \\ \textit{VarApp } x_v \to x_v \lfloor R \rceil \\ \textit{StuckApp } x_s \to x_s \lfloor R \rceil \end{cases}$$

$$\to^* \textit{case } H_2 \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a \lfloor R \rceil \\ \textit{VarApp } x_v \to x_v \lfloor R \rceil \\ \textit{StuckApp } x_s \to x_s \lfloor R \rceil \end{cases} \to^* H$$

This also implies that we can now use the IH to obtain that $\lfloor M \rceil \to^* H_2$. With this in mind, we have that:

$$\textit{case } \lfloor M \rceil \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a \lfloor R \rceil \\ \textit{VarApp } x_v \to x_v \lfloor R \rceil \\ \textit{StuckApp } x_s \to x_s \lfloor R \rceil \end{cases} \to^* \textit{case } H_2 \textit{ of } \begin{cases} \textit{Thr } (x_t, n_t) \to \textit{Thr } (x_t, n_t) \\ \textit{Abs } x_a \to x_a \lfloor R \rceil \\ \textit{VarApp } x_v \to x_v \lfloor R \rceil \\ \textit{StuckApp } x_s \to x_s \lfloor R \rceil \end{cases}$$

But we know from the previous reasoning that the above reduces to H, as required. □

$(M \to N \Rightarrow \textit{try } M;\ \textit{CB} \to \textit{try } N;\ \textit{CB})$: This is the second inductive case. The proof is completely identical to the one above, except that the content of the *case* construct is different, so we do not show it here. □

This completes the proof of Lemma 5.6.4.

Main result

We can now proceed with the proof of Theorem 5.6.1: If λ^{try} terms P and Q satisfy $P \to Q$ and Q has a normal form, then there is a unique $HM+$ term H given by the H syntax such that $\lfloor Q \rceil \to^* H$ and $\lfloor P \rceil \to^* H$. The proof follows from Lemma 5.6.3 and Lemma 5.6.4.

Assume $P \to Q$. By assumption, we know Q has some normal form Q_H. Thus:

$P \to Q \to Q_1 \to \cdots \to Q_n \to Q_H$

Applying Lemma 5.6.3 to Q_H, we have that there exists a term H as described by the H syntax such that $\lfloor Q_H \rceil \to^* H$.

Then, applying Lemma 5.6.4 to Q_n and Q_H, we also have that $\lfloor Q_n \rceil \to^* H$.

Continuing, if we apply Lemma 5.6.4 to Q_{n-1} and Q_n, we also have that $\lfloor Q_{n-1} \rceil \to^* H$.

We can keep this process up until we reach both P and Q, which will yield $\lfloor P \rceil \to^* H$ and $\lfloor Q \rceil \to^* H$, as required. Recall also that reduction in $HM+$ is deterministic and H is in normal form: this implies that H is unique (there is no other term in normal form that P or Q can reduce to). This completes the proof of Theorem 5.6.1.

Chapter 6

Conclusion

6.1 Evaluation

- **Investigation:** One question of particular importance for this project has been what kind of extensions would be necessary in order to be able to implement λ^{try} in Haskell. We have shown that this boils down to being able to translate λ^{try} to System FC in a way that satisfies some form of preservation of reduction and assignable types. We believe to have brought strong evidence that λ^{try} can not be translated to Hindley-Milner in such a way without extending Hindley-Milner, and, because of the relationship between it and System FC, we believe this is a strong indicator that the same holds for System FC. However, until proven otherwise (which we have not been able to do), the possibility (of translation without extension) remains.

- **Evaluator:** We believe the evaluator that we have introduced accurately models λ^{try} via means of the `Either` data type, as we have proved that it preserves reduction. This shows that there is some connection between λ^{try} and the traditional way that exceptions are handled in Haskell (through data types, for example by representing an exception through a `Left`). As one of the more interesting parts of the evaluator can also be expressed in terms of monadic operators (`Either a` is a monad in Haskell), this implies that there might be a link between λ^{try} and monads. We would have liked to explore this idea further (and perhaps through a more theoretical lens), however, as we have turned the evaluator into a proper language extension, it seems the connection with monads was lost.

- **Extension and Translation:** The extension and translation we have presented highlight a possible solution to the meaning of the constructs in λ^{try}. It seems that throwing and catching as modeled there are indeed not expressible in standard calculi (such as Hindley-Milner) - in both λ^{try} and $\lambda\mu$ (into which it can be translated), there is an implicit *case* construct that allows dealing with throws. This is present, for example, in the reduction rules for the application: λ^{try} specifically treats application one way if the first term in it is a *throw* and another way if it is an abstraction, while $\lambda\mu$ treats application one way if the first term is a λ-abstraction and another way if it is a μ-abstraction. This can be seen as a form of *case* construct - our extension and translation have made this fact clear by having the *case* explicitly as part of the syntax, instead of the reduction rules.

The translation seems feasible, as we have proved that it preserves reduction. We believe our extension is a nice bridge between System FC as it is and λ^{try}: it is general enough to be reasonably be implemented in GHC as part of the former, and specific enough to accurately model the latter (as opposed to blindly adding λ^{try} to System FC directly, which would obviously give us a way to translate λ^{try}, but the GHC implementation of which we believe would be of questionable feasibility).

On the other hand, it can also be said that the extension we have ended up with is rather verbose and complex. This has been an obstacle in tackling an actual GHC implementation, which has led us to deal with mostly the theoretical side. In this respect, the implementation obtained by Fisher [4] is more practical (even if it is based on a library with data types that model CDC, meaning that it can be considered a form of evaluator, rather than making Haskell do reduction the way that is stipulated in λ^{try}).

6.2 Conclusion

We explored the possibility of adding the feature of exception handling by name to Haskell, based on van Bakel's λ^{try} [1]. We defined a Haskell evaluator for λ^{try} based on the `Either` data type and monad and proved that it preserves reduction. We have learned that adding features that precisely model λ^{try} to Haskell would imply translating λ^{try} to System FC, on which Haskell is based on, in a way that preserves reduction and types in some form. We have also deduced that λ^{try} is most likely not able to be translated to Hindley-Milner or System FC without extending those systems as well. Using our evaluator, we have developed an extension to Hindley-Milner that allows λ^{try} to be mapped to it and we have proved that it preserves reduction. This project takes the form of an investigation and it brings some clarity to the issue of implementing λ^{try} as part of Haskell's internal reduction engine.

6.2.1 Future Work

- **Compacting the translation**: Our *case* construct has 4 branches - however, as we have seen, this level of expressive power is not needed, because the last 3 serve as termination criteria and they always do the same thing. A nice improvement would be to replace them with a single branch, to which the 3 different reduction rules (each of which previously corresponded to a different branch) would now apply.

- **Proving type preservation:** We have not proved any kind of type preservation results for our evaluator and our translation. Doing so would provide more insight into the problem and would be necessary if our extension is to be implemented in GHC.

- **Tackling GHC implementation:** As already mentioned, we have not made any significant progress with regards to implementing our extension in GHC. This is, of course, a very important practical issue and we would have liked to explore this side more. We believe an implementation could be developed one step at a time (for example, we could first try to add a simple *case* construct, that deals only with normal application, without any of the new λ^{try} constructs, and work from there).

- **Implementing $\lambda\mu$ in GHC:** Because we have tried to avoid the path taken in Fisher's previous work [4] and come up with a direct translation, we have not focused very much on $\lambda\mu$ in this project. However, it is known that λ^{try} can be translated to $\lambda\mu$, and $\lambda\mu$ is much more compact than our extension. Furthermore, as van Bakel remarks in [1], only a small subset of $\lambda\mu$ is used to translate λ^{try} - we can see in the translation that, for example, the only kinds of μ-abstractions that are used start with $\mu n.[n]$ and $\mu_{\_}.[n]$. This raises the question of whether it would not be easier to add the relevant parts of $\lambda\mu$ to System FC and then translate λ^{try} using those.

- **Exploring alternative routes:** Although we do not think it is unreasonable to implement our extension in GHC, it remains a possibility that alternative, simpler routes exist. We started from an evaluator based on `Either`, as the `Left` and `Right` categorisation seemed promising with regards to handling the throws, and we also found a monadic connection on the way. However, in the process of developing our extension, we have had to give up the `Left` and `Right` distinction and replace them by more complicated machinery. We believe it is worth investigating if an evaluator could be implemented using different concepts, which could, in turn, give rise to other forms of extensions.

Bibliography

[1] S. van Bakel. "Exception Handling and Classical Logic". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*. PPDP '19. Porto, Portugal: Association for Computing Machinery, 2019. ISBN: 9781450372497. DOI: 10.1145/3354166.3354186. URL: https://doi.org/10.1145/3354166.3354186 (visited on May 28, 2020).

[2] A. Baciu. "Exception Handling for Haskell via Delimited Continuations with Preservation of Types". Master's thesis. Imperial College London, 2017.

[3] J. R. Griffiths. "A Curry-Howard Correspondence for Failing Exceptions". Master's thesis. Imperial College London, 2018.

[4] W. S. Fisher. "Exception Handling in Haskell Using the λ^{try} Calculus". Master's thesis. Imperial College London, 2016.

[5] Wikipedia. URL: https://en.wikipedia.org/wiki/Formal_system (visited on May 29, 2020).

[6] S. van Bakel. *Type Systems for Programming Languages*. Course notes (revised edition 2016). 2001. URL: https://www.doc.ic.ac.uk/~svb/TSfPL/notes.pdf (visited on Jan. 19, 2020).

[7] M. Parigot. "Classical Proofs as Programs". In: *Proceedings of the Third Kurt Gödel Colloquium on Computational Logic and Proof Theory*. KGC '93. Berlin, Heidelberg: Springer-Verlag, 1993, pp. 263–276. ISBN: 3540571841.

[8] T. Crolard. "A confluent λ-calculus with a catch/throw mechanism". In: *Journal of Functional Programming* 9.6 (1999), pp. 625–647. DOI: 10.1017/S0956796899003512. URL: http://cedric.cnam.fr/sys/crolard/publications/jfp.pdf (visited on June 3, 2020).

[9] R. K. Dyvbig, S. Peyton Jones, and A. Sabry. "A Monadic Framework for Delimited Continuations". In: *J. Funct. Program.* 17.6 (Nov. 2007), pp. 687–730. ISSN: 0956-7968. DOI: 10.1017/S0956796807006259. URL: https://doi.org/10.1017/S0956796807006259 (visited on May 28, 2020).

[10] Haskell Wiki. URL: https://wiki.haskell.org/Exception (visited on May 18, 2020).

[11] GHC website. URL: https://www.haskell.org/ghc/ (visited on Apr. 15, 2020).

[12] S. Peyton Jones. *The Haskell 98 Language Report*. 2002. URL: https://www.haskell.org/onlinereport/index.html (visited on Apr. 15, 2020).

[13] Wikipedia. *System F*. URL: https://en.wikipedia.org/wiki/System_F (visited on Apr. 15, 2020).

[14] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. "System F with Type Equality Coercions". In: *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. TLDI '07. Nice, Nice, France: Association for Computing Machinery, 2007, pp. 53–66. ISBN: 159593393X. DOI: 10.1145/1190315.1190324. URL: https://doi.org/10.1145/1190315.1190324 (visited on May 28, 2020).

[15] GHC documentation. URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/hsc-main (visited on Apr. 15, 2020).

[16] LLVM Project website. URL: https://llvm.org (visited on Apr. 15, 2020).

[17] P. Nogueira. *A Short Introduction to Systems F and F_ω*. 2006. URL: https://babel.ls.fi.upm.es/~pablo/Papers/Notes/f-fw.pdf (visited on May 28, 2020).

[18] Wikipedia. *Hindley-Milner type system*. URL: https://en.wikipedia.org/wiki/Hindley-Milner_type_system (visited on Apr. 16, 2020).

[19] R. A. Eisenberg. *System FC, as implemented in GHC*. Tech. rep. MS-CIS-15-09. University of Pennsylvania, 2015. URL: https://repository.brynmawr.edu/compsci_pubs/15/ (visited on May 28, 2020).

[20] GHC documentation. URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-syn-type (visited on Apr. 16, 2020).