Continuations in $\mathcal{X}$

Charles Collicutt

Project Supervisor: Steffen van Bakel
Second Marker: Maria Vigliotti

Department of Computing, Imperial College London, 180 Queen's Gate, London, SW7 2BZ
cac04@doc.ic.ac.uk

June 19, 2007

Contents

Abstract

A one-to-one correspondence, known as the *Curry-Howard Isomorphism* after its discoverers, has been shown to exist between the simply typed λ-calculus and implicative intuitionistic logic in a natural deduction framework. This correspondence has allowed researchers to analyse the links between intuitionistic logic and function abstraction and application. $\mathcal{X}$ is a logical calculus that exhibits a Curry-Howard Isomorphism with implicative classical logic. Research has shown that classical logic is linked to control in computation. Unlike most logical calculi, $\mathcal{X}$ has no notion of function abstraction or application nor does it include variables or substitution. Instead, $\mathcal{X}$ describes structures called "circuits" that consist of named "plugs" and "sockets" that can be wired together. There is a notion of "flow" through these circuits, which leads to the intuition that, rather than modelling functional programming like the λ-calculus, $\mathcal{X}$ models control flow and continuations. In this project I investigate the representation of continuations in $\mathcal{X}$. By constructing a translation from $\mathcal{X}$ to an abstract machine that models the saving and restoration of evaluation contexts, I reveal how reduction in $\mathcal{X}$ corresponds to operations on continuations.

Acknowledgements

I would like to thank Dr. van Bakel for his support and encouragement and for providing me with the most interesting and enjoyable part of my course.

I would like to thank Alex Summers for his advice and explanations.

1 Introduction

The λ-calculus [4] has proved very useful for the investigation of function abstraction and application. It inspired the functional programming paradigm, including the languages LISP, ML and Haskell. Through the Curry-Howard Isomorphism [8], a one-to-one correspondence between implicative intuitionistic logic and the simply typed λ-calculus has been established. This gives us an insight into the relationship between logic and computation. However, intuitionistic logic is not as expressive as classical logic. Recently, various new calculi have been invented that exhibit a Curry-Howard Isomorphism with fragments of classical logic. Investigations of these calculi have revealed a link between classical logic and control and continuations [7, 9, 12, 1, 2, 5].

$\mathcal{X}$ is a language that exhibits the Curry-Howard Isomorphism with the implicative fragment of classical logic [16]. Whereas typing for the λ-calculus is based on natural deduction, $\mathcal{X}$ is inspired by Gentzen's sequent calculus [14]. Function abstraction and application do not appear in $\mathcal{X}$ (although they can be interpreted) and it does not have variables or substitution. Instead, $\mathcal{X}$ describes structures called "circuits" (or "nets") whose component parts can be connected together. A circuit may contain a number of "plugs" that can be connected to the "sockets" of another circuit (or vice versa.) We have a notion of "flow" through the circuits - that is to say, the connections between plugs and sockets have a direction. Thus, intuitively, $\mathcal{X}$ seems to model contexts or continuations. The purpose of my project has been to elucidate this idea: I will show how continuations are represented in $\mathcal{X}$ and how reductions in $\mathcal{X}$ are related to operations on continuations. This will provide us with a further insight into the link between classical logic and computation.

Parigot extended the λ-calculus to create a new calculus which would correspond to a fragment of classical logic [13]. He called his calculus $\lambda\mu$. However, unlike $\mathcal{X}$, Parigot's $\lambda\mu$ is still based on a natural deduction framework. Bierman has shown [5] that $\lambda\mu$ can be interpreted as "a typed λ-calculus which is able to save and restore the runtime environment." He provides an abstract machine for $\lambda\mu$ which demonstrates its ability to represent the saving and restoring of continuations. So for my project I have investigated representations of $\lambda\mu$ in $\mathcal{X}$ (and vice versa) and have produced a translation from $\mathcal{X}$ to Bierman's abstract machine via $\lambda\mu$. Having proved the completeness and consistency of this translation, I have used it to discover then illustrate the representation of continuations in $\mathcal{X}$.

2 What is a continuation?

Continuations are a concept used to *dynamically* modify the control flow of a program. In order to explain what I mean by this, it will first be necessary to recall some better known *static* methods of modifying a program's control flow. By "static method" I mean a method whose effects can be completely analysed statically, i.e. without running the program. Because such analysis depends only on the source code of the program (and not the context in which it is run) I will use "lexical" as a synonym for "static".

The most general lexical method of altering the control flow of a program is the `goto` statement. This statement causes control to jump to a label elsewhere in the program: the target label may appear anywhere in the program. All that happens in practice is that the *program counter* is overwritten with the address of the instruction referred to by the label. Thus the `goto` statement can be used to cause control to jump to any point in the program. For example, consider the C program in Listing 1.

Listing 1: goto in C

```
                int i = 0;
firstlabel:
                ++i;
                printf("%d", i);
                goto thirdlabel;
secondlabel:
                --i;
                printf("%d", i);
thirdlabel:
                ++i;
                goto firstlabel;
```

When compiled and run, this program will output all the odd natural numbers in ascending order. The code between the second and third labels will never be executed: control jumps to the third label then back to the first *ad infinitum*.

In order to better structure our control of the program, we can introduce the notion of structure into the program's code. Typically this is done by dividing the program into nested *blocks*. Now we have an outermost block, consisting of the program in its entirety, within which there are blocks of code that may themselves contain further blocks. Most trivially, we can use these blocks to implement loops and conditionals. For example, the C program in Listing 2 will output the odd natural numbers less than 20 in ascending order and it will do so five times.

Listing 2: Structuring with blocks

```
int i = 0;
int j = 0;

while (i < 5) {
        while (j < 20) {
                if (j % 2 != 0) printf("%d", j);
                ++j;
        }
        j = 0;
        ++i;
}
```

Here the blocks are denoted by curly braces and are used to determine the scope of `while` loops. Now that the program is lexically structured into blocks, we can use that structure to implement a more structured (i.e. restricted) version of `goto`. In fact, the `while` loop is itself a combination of a conditional and a structured `goto`. However, more interestingly, in C we can use the `break` statement to make control jump outside the current block. Listing 3 contains an alternative implementation of the program in Listing 2 using `break` statements.

Listing 3: Breaking out of blocks

```
int i = 0;
int j = 0;

while (1) {
        while (1) {
                if (j % 2 != 0) printf("%d", j);
                ++j;
                if (j > 19) break;
        }
        j = 0;
        ++i;
        if (i > 4) break;
}
```

Here `break` is acting like `goto` but it is restricted to jumping “outwards”. It cannot cause control to jump to any arbitrary point in the program as `goto` can, instead it can only cause control to jump into an outer block. This is even more obvious in Java, in which the `break` statement is more flexible than in C. In Java, you can `break` to a label in much the same way that you would use `goto` but you can only do so if the label is in an outer block. For example, if you were searching through a matrix for an instance of the integer 13 you might do something like Listing 4.

Listing 4: Breaking to a label

```
int i = 0;
int j = 0;
boolean found = false;

outside:
while (i < WIDTH) {
        while (j < HEIGHT) {
                if (matrix[i][j] == 13 {
                        found = true;
                        break outside;
                }
                ++j;
        }
        ++i;
}
```

Although the Java syntax requires you to place the label at the beginning of the block from which you wish to escape, it is clearly much the same as a `goto`. However, Listing 5 is not valid Java code.

Listing 5: Invalid Java Break

```
while (i < MAXI) {
        while (j < MAXJ) {
                break sideways;
        }
}

while (k < MAXK) {
        sideways:
        while (l < MAXL) {
        }
}
```

Here an attempt is made to jump not to an outer block but to another block at the same depth in the "nesting tree". While `goto` can perform such arbitrary jumps, the more restricted `break` cannot. For this reason, we might call `break` an "outgoing-only `goto`".

Continuations are also used to modify the control flow of a program but they do not do so statically. Their behaviour cannot be completely determined by a lexical analysis. Rather than dealing with lexical structures such as blocks, continuations operate on the control stack itself. So it will now be useful to investigate the idea of a control stack.

If we ignore for the moment the method of passing arguments and returning results, a function call consists of two things. Some control information, such as the return address and the stack base pointer, is pushed onto the stack and control jumps to the beginning of the function code. When the function has completed, the control information is used to jump back to the point in the program at which the function was called and to pop everything that the function pushed onto the stack. So we can view the stack as being divided into *frames*. Every time a function is called another frame is added to the end of

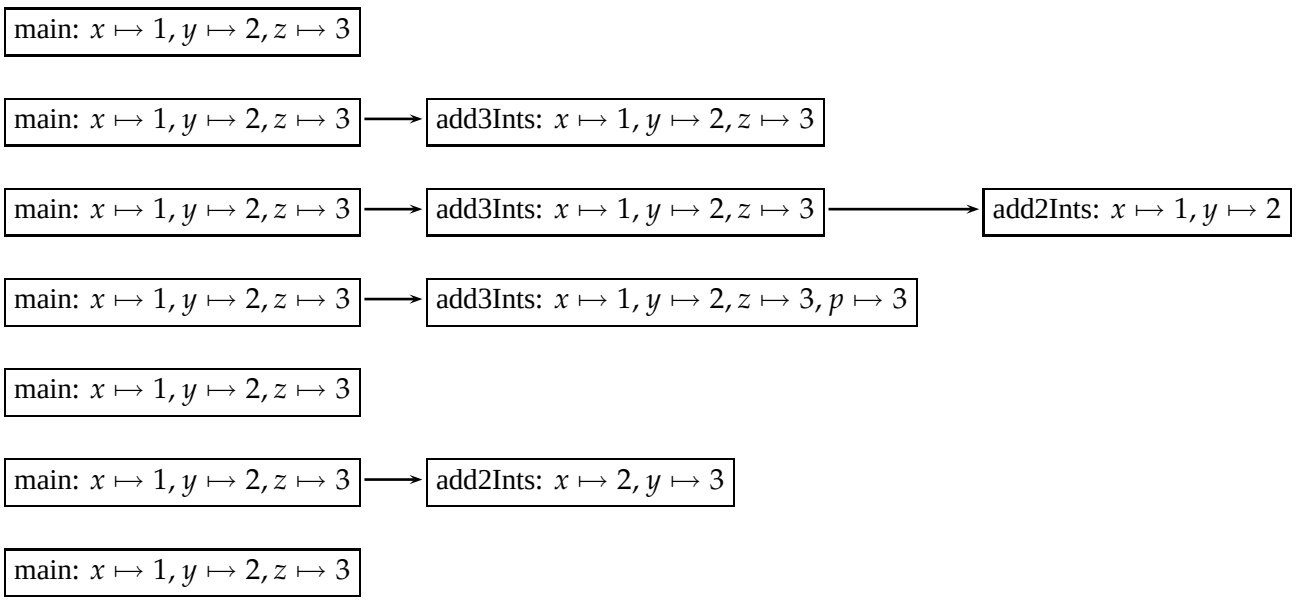
the stack and every time a function returns a frame is removed from the end of the stack.

But rather than considering the actual block of memory used as a stack, consider instead the slightly more abstract idea of a control stack. It can be thought of as a linked list with each node corresponding to a frame. Each frame contains the information local to that function (the *context*) and a link back to the frame of the function in which it was called. Thus every time a function is called, another frame is added to the end of the list; and every time a function returns, its frame is removed from the end of the list. Consider the C program in Listing 6.

Listing 6: A C program with nested function calls

```
int add2Ints(int x, int y) {
        return x + y;
}

int add3Ints(x, y, z) {
        int p = addTwoIntegers(x, y);
        return p + z;
}

int main() {
        int x = 1;
        int y = 2;
        int z = 3;

        add3Ints(x, y, z);
        add2Ints(y, z);

        return 0;
}
```

As this program runs it would have a control stack that grows and shrinks like this:



If we make an analogy between lexical blocks and frames in a control stack, we can see that the `break` statement is analogous to an *exception*. In languages that support this feature, it is possible to *throw* (or *raise*) an exception. When this is done, execution in the current function stops and control returns to the calling function, i.e. control moves from the current frame to the next frame up the control stack. This is much the same as returning from a function as shown above. However, unlike a function return, if the exception is not *caught* it will propagate up the control stack. Control will continue to jump frame by frame up the control stack until either the exception is caught or the top of the stack is reached and the program terminates.

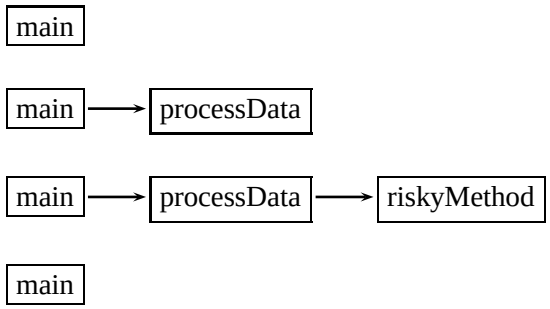
An exception may be caught by enclosing the code that might lead to the throwing of an exception in a *try* block. Attached to this try block should be a *catch* block containing an *exception handler*, which is a block of code to be executed in the event of an exception being caught. Exceptions are usually used to signal that an error has occurred, so the exception handler tends to be code designed to help the program recover from an error. Exceptions can be typed: in object-oriented languages they are usually implemented as instances of an *exception class* with a whole hierarchy of classes to represent different types of errors. A single try block may be associated with multiple catch blocks, each one intended to catch a different type of exception (perhaps one for an IO error, one for an array index out of bounds error and so on.) Listing 7 demonstrates the use of exceptions in Java.

Listing 7: Exceptions in Java

```
public class Example {
        private void riskyMethod() throws Exception {
                // Do something that might result in an error
                if (error == true) {
                        throw new Exception();
                }
        }

        private void processData throws Exception {
                try {
                        riskyMethod();
                        // Do some IO action here, e.g. read a file
                } catch (IOException e) {
                        // Deal with the IO error
                }
        }

        public static void main(String[] args) {
                try {
                        processData();
                } catch (Exception e) {
                        // Deal with the error
                }
                // Do something else
        }
}
```

In this example, the `main` method calls the `processData` method that in turn calls the `riskyMethod` method. If `riskyMethod` encounters an error it will throw an exception. Execution will immediately

cease in `riskyMethod` and the exception will be thrown again in `processData`. `processData` will not catch this exception (as its catch block only catches exceptions of type `IOException`) so execution will cease in `processData` and the exception will be thrown again in `main`. Here the exception will be caught and execution will continue in `main` after the try block. So, assuming `riskyMethod` does indeed encounter an error, the control stack will grow and shrink like this:



Thus function returns, which cause control to move up one frame, are analogous to `break`ing out of the innermost block (as `break` does in C.) Whereas exceptions, which can propagate up through the control stack and so cause control to jump many frames in one go, are analogous to labelled `break`s in Java, which can jump out of many nested blocks in one go.

Just as `break` can be thought of as an "outgoing-only `goto`", so exceptions can be thought of as "upwards-only continuations" [10]. As we have seen, exceptions can only cause control to move up the control stack. However, in languages that support first-class continuations, you can save a reference to any frame and use that reference to move control to that frame at any time. Also, such languages typically employ some form of garbage collection. This means that a frame is no longer automatically removed when the function that created it returns: now a frame is only removed when *all* references to it have been deleted, including both the implicit reference from the frame above it in the control stack and any other references that you might have created. Thus the control stack is no longer a stack at all but instead an arbitrary graph.

For a program without any `goto` statements, the lexical structure of the program can be described in terms of nested blocks forming a hierarchical tree. The use of the unrestricted `goto` statement destroys that structure by allowing control to jump to any arbitrary point in the tree. Analogously, for a program without continuations, the control flow of the program can be represented by a structured control stack. The use of first-class continuations destroys that structure by turning the control stack into an arbitrary control graph.

First-class continuations first appeared in the programming language LISP. In the Scheme dialect of LISP there is a function called `call-with-current-continuation`, usually abbreviated to `call/cc`. This function takes a single argument, which is itself a function that takes a single argument, and it passes to that function a reference to the current continuation. So `call/cc` calls its argument with the current continuation. You can then, for instance, store the reference to the continuation in a variable, which will allow you to invoke (restore) the continuation at another point in the program (in Scheme, as in some other languages with first-class continuations, a continuation can be invoked by calling it in the same way that you call a function.) Listing 8 demonstrates this function [18].

Listing 8: call/cc in Scheme

```
(define aContinuation #f)

(define (test)
        (let ((i 0))
                (call/cc (lambda (k) (set! aContinuation k)))
                (set! i (+ i 1))
                i
        )
)
```

In the first line of Listing 8, a variable called `aContinuation` is defined with a dummy value. In the rest of the listing, a function called `test` is defined. The whole body of the function sits within a `let` block, which defines the local variable `i` and initialises it to zero. The first line of the `let` block invokes the `call/cc` function.

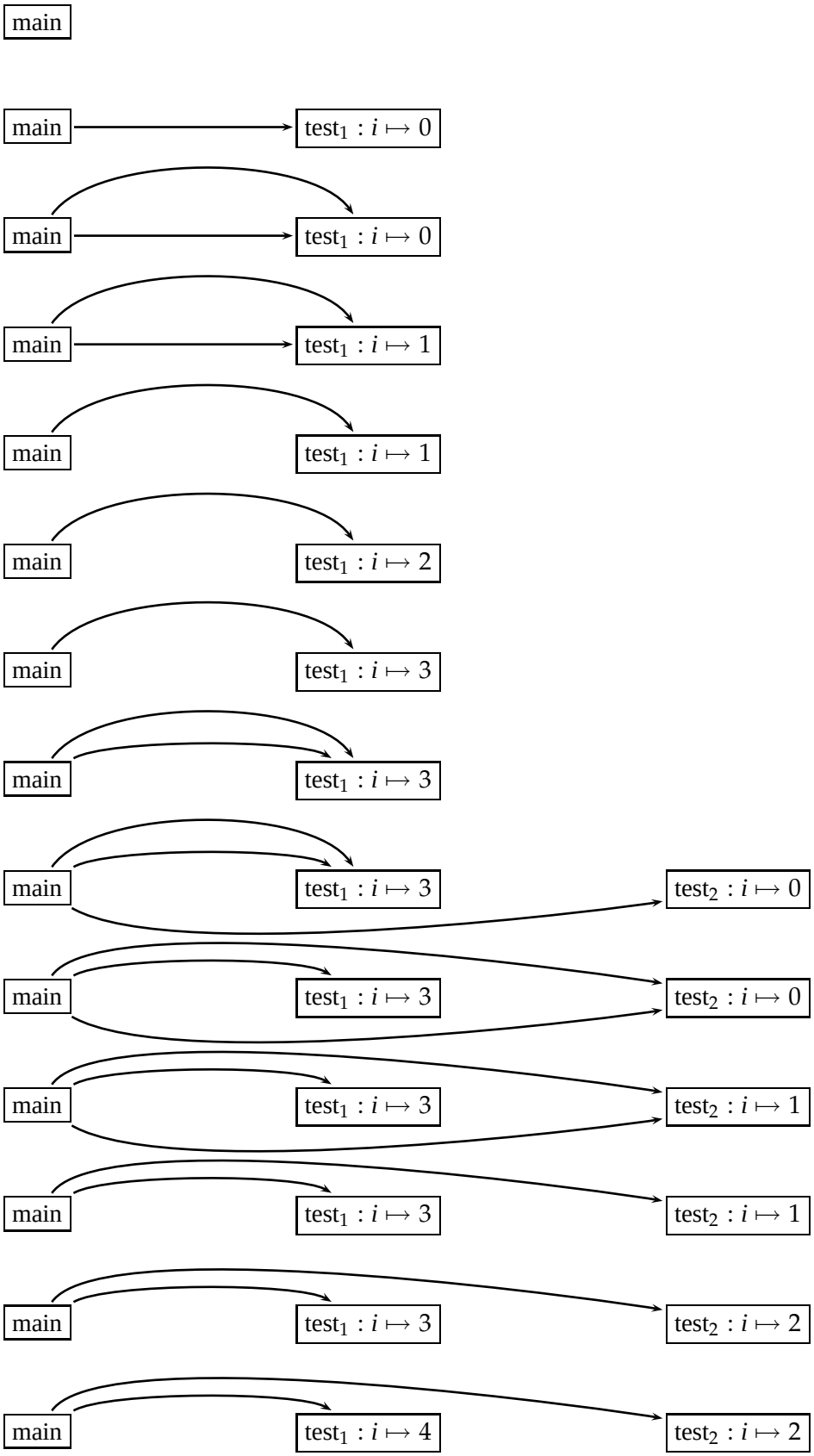
The `lambda` construction defines an anonymous function (the name is inspired by the λ-calculus) which is passed as an argument to `call/cc`, which calls its argument with a reference to the current continuation. The anonymous function uses the built-in `set!` command to assign the reference to the continuation to the global variable `aContinuation`. In the rest of the `test` function, the local variable `i` is incremented by one and then returned.

Assume the existence of an interactive Scheme interpreter which displays the return value of a function call. Listing 9 shows some results you could obtain having first run the program in Listing 8.

Listing 9: Running Listing 8 on an interpreter

```
> (test)
1
> (aContinuation)
2
> (aContinuation)
3
> (define anotherReference aContinuation)
> (test)
1
> (aContinuation)
2
> (anotherReference)
4
```

If we call the interpreter's frame `main`, then running Listing 9 would cause the control "stack" to develop as shown below.


main
main
test$_1$: $i \mapsto 0$
main
test$_1$: $i \mapsto 0$
main
test$_1$: $i \mapsto 1$
main
test$_1$: $i \mapsto 1$
main
test$_1$: $i \mapsto 2$
main
test$_1$: $i \mapsto 3$
main
test$_1$: $i \mapsto 3$
main
test$_1$: $i \mapsto 3$
test$_2$: $i \mapsto 0$
main
test$_1$: $i \mapsto 3$
test$_2$: $i \mapsto 0$
main
test$_1$: $i \mapsto 3$
test$_2$: $i \mapsto 1$
main
test$_1$: $i \mapsto 3$
test$_2$: $i \mapsto 1$
main
test$_1$: $i \mapsto 3$
test$_2$: $i \mapsto 2$
main
test$_1$: $i \mapsto 4$
test$_2$: $i \mapsto 2$

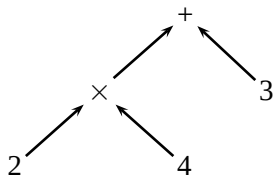
When discussing logical calculi, we can analyse continuations using the idea of an *evaluation context*. The intuition behind the use of contexts is that a continuation is essentially a "calculation with a hole in it." The hole represents the result of some sub-calculation that has yet to be achieved and the evaluation context describes what will happen to that result when it has been achieved (i.e. how it will be evaluated.) In the terms that we have been using so far to describe continuations, the hole represents the point in the control graph referred to by a continuation variable. For example, consider the following arithmetical calculation:

$$2 \times 4 + 3$$

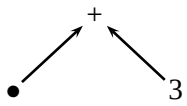
According to the normal rules of precedence, the multiplication will be performed before the addition. So we could say that there is a continuation here: 3 will be added to whatever the result of the multiplication turns out to be. We can represent this by a formula with a hole in it:

$$\bullet + 3$$

A graph of the original calculation might look like this:



In which case the formula with a hole would look like this:



This gives us a useful method of representing continuations in logical calculi as contexts and it is the method I will use in this project.

3 Classical and Intuitionistic Logic

Intuitionistic logic is the logical system developed by Heyting as a formal basis for Brouwer's intuitionistic mathematics [11]. According to Brouwer, in order to show $(A \vee B)$ you need a proof of A or a proof of B. For example, consider the following mathematical proof [3].

Theorem 3.1 *There are irrational numbers a and b such that a^b is rational.*

Proof: Consider $\sqrt{2}^{\sqrt{2}}$. If this is rational, we are done: we can let $a = b = \sqrt{2}$. Otherwise, it is irrational. Then we have

$$\left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = \sqrt{2}^{2} = 2,$$

which is rational. So, in this case, let a be $\sqrt{2}^{\sqrt{2}}$ and let b be $\sqrt{2}$. □

Classically, this proof is valid. However, according to Brouwer's intuitionism, it is not. We have proved the existence of a pair of numbers that have a certain property but we are unable to say *which* pair of numbers it is. An intuitionistic existence proof should provide an unconditional definition of the objects it asserts to exist. The above proof rests on the assumption that "either $\sqrt{2}^{\sqrt{2}}$ is rational or it is not" but Brouwer would argue that such a claim needs further justification of its own.

This has the consequence that the law of excluded middle $(A \vee \neg A)$ is an axiom in classical logic but not in intuitionistic logic. Thus there are proofs that are classically valid but intuitionistically unprovable. However, a translation from classical proofs to intuitionistic proofs, first defined by Gödel and Gentzen, links classical logic and intuitionistic logic.

Definition 3.2 (DOUBLE NEGATION TRANSLATION)
The translation $[\![\cdot]\!]$ is defined inductively as follows:

$$\begin{aligned}
[\![A]\!] &\triangleq \neg\neg A, \text{ if } A \text{ is atomic} \\
[\![A \Rightarrow B]\!] &\triangleq [\![A]\!] \Rightarrow [\![B]\!] \\
[\![A \wedge B]\!] &\triangleq [\![A]\!] \wedge [\![B]\!] \\
[\![A \vee B]\!] &\triangleq \neg(\neg[\![A]\!] \wedge \neg[\![B]\!])
\end{aligned}$$

Theorem 3.3 (DOUBLE NEGATION TRANSLATION)
Let A be a proposition composed of one or more atomic propositions and the logical connectives $\vee$, $\wedge$ and $\Rightarrow$. A is provable classically if and only if $[\![A]\!]$ is provable intuitionistically.

Thus intuitionistic logic makes a distinction between $\neg(\neg A \wedge \neg B)$ and $(A \vee B)$ that classical logic does not make. However, the calculi in which we are interested for the purposes of my project only correspond to implicative logic so we need only be concerned with the first two translation rules.

4 Natural Deduction and Gentzen's Sequent Calculus

In a deduction system with sequents, one can write $A \vdash B$ to signify that "A justifies B". A is an *assumption* and B is a *conclusion*. In the natural deduction system, there is only ever a single conclusion but there may be multiple assumptions. So $A_1, A_2, \ldots, A_n \vdash B$ should be read as $A_1 \wedge A_2 \wedge \ldots \wedge A_n \vdash B$. We use Γ to signify an arbitrary conjunction of assumptions.

Natural deduction rules either introduce or eliminate logical connectives on the right-hand side of the sequent. For example, implicative intuitionistic logic can be represented in a natural deduction framework with just three rules:

$$(Ax) : \frac{}{\Gamma \vdash A}\ (A \in \Gamma)$$

$$(\rightarrow I) : \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$$

$$(\rightarrow E) : \frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

In contrast, the rules of Gentzen's sequent calculus LK [14] only introduce connectives but can do so to both sides of the sequent. LK allows sequents with multiple alternative conclusions, such as $A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_n$, which should be read as $A_1 \wedge A_2 \wedge \ldots \wedge A_n \vdash B_1 \vee B_2 \vee \ldots \vee B_n$. We use Δ to signify an arbitrary disjunction of conclusions. The only way to eliminate a connective in LK is to eliminate the whole formula in which it appears by an application of the (cut)-rule. LK$(\rightarrow)$ is a variant of this calculus which can be used to represent implicative classical logic:

$$(Ax) : \frac{}{\Gamma, A \vdash A, \Delta} \qquad (cut) : \frac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}$$

$$(\Rightarrow R) : \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \qquad (\Rightarrow L) : \frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}$$

Gentzen defined a *cut-elimination procedure* for LK that removes all applications of the (cut)-rule from a proof. He proved [14] that for any LK proof that contains cuts there exists a normalised version of that proof in which all cuts have been removed. The procedure works via a series of local rewrites of the proof tree and we will see in §6 that reduction in $\mathcal{X}$ is based on cut elimination for LK.

5 The simply typed λ-calculus and Curry-Howard Isomorphism

I assume the reader is familiar with the λ-calculus [4] and will just briefly recall the definition of lambda terms and β-reduction.

Definition 5.1 (LAMBDA TERMS AND β-REDUCTION)
The set Λ of *lambda terms* is defined by this syntax:

$$M, N ::= x \mid \lambda x.M \mid MN$$

The reduction relation $\rightarrow_\beta$ is defined as the compatible closure of this rule:

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

This calculus has a notion of type assignment that corresponds to implicational intuitionistic logic in a natural deduction framework. This correspondence property is called the *Curry-Howard Isomorphism*. Informally, this is the "Terms as Proofs, Types as Propositions" idea.

Definition 5.2 (CURRY-HOWARD ISOMORPHISM)
Let M be a (closed) term, and A a type, then M is of type A if and only if A, read as a logical formula, is provable in the corresponding logic, using a proof whose structure corresponds to M.

This isomorphism expresses the fact that one can associate a term with a proof such that propositions become types and proof reductions become term reductions [6, 8]. Logical formulae can be seen as types and vice versa. The implication $A \Rightarrow B$ corresponds to the type $A \rightarrow B$. Further, the inference rules of implicative intuitionistic logic are isomorphic to the typing rules of the simply typed λ-calculus:

Definition 5.3 (TYPE ASSIGNMENT FOR λ-CALCULUS)
Every Λ term has a type, derived using the rules below.

$$(Ax) : \frac{}{\Gamma, x{:}A \vdash_\lambda x : A} \qquad (\rightarrow I) : \frac{\Gamma, x{:}A \vdash_\lambda M : B}{\Gamma \vdash_\lambda \lambda x.M : A \rightarrow B}$$

$$(\rightarrow E) : \frac{\Gamma \vdash_\lambda M : A \rightarrow B \qquad \Gamma \vdash_\lambda N : A}{\Gamma \vdash_\lambda MN : B}$$

5.1 Reduction Strategies

It is frequently the case that a term in a calculus could be reduced in more than one way. For example, the lambda term $(\lambda x.xx)((\lambda y.y)z)$ could be reduced in two different ways:

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

Lambda terms of the form x or $\lambda x.M$ are called *values*. If β-reduction only occurs when the argument is a value then the system obeys the *Call-by-Value* reduction strategy (CBV). In contrast, the second example above demonstrates a *Call-by-Name* reduction (CBN) in which substitution occurs before the argument has been reduced to a value. If a term in a calculus will always reach the same normal form regardless of which strategy is used then that calculus is *confluent*. It is notable that systems based on natural deduction are generally confluent whereas $\mathcal{X}$ is not. Cut-elimination in LK is not confluent: non-determinism is a key feature of classical logic.

6 $\mathcal{X}$

$\mathcal{X}$ is a language designed to exhibit the Curry-Howard Isomorphism with implicative classical logic. It is based on LK rather than natural deduction, which means that it does not have abstraction ($\rightarrow I$) and application ($\rightarrow E$) but instead has four syntactic constructs, corresponding to the (Ax), ($\Rightarrow R$), ($\Rightarrow L$) and (cut) rules of LK($\rightarrow$), and a system of reduction based on cut-elimination.

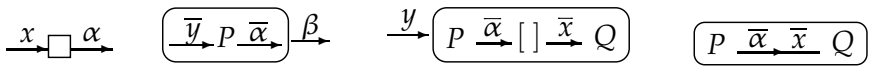
6.1 Syntax

The terms of $\mathcal{X}$ are called "circuits" and they are composed of named "plugs" and "sockets". If you have two circuits, designated P and Q, then you can cut them together, written $P\widehat{\alpha} \dagger \widehat{x}Q$, which can be thought of as wiring the plugs named α in P to the sockets named x in Q. Using the notation from the *Principia Mathematica* [17] we write $\widehat{x}$ to indicate that x is bound. So, in the previous example, α is bound in P and x is bound in Q.

Definition 6.1 (SYNTAX)
The circuits of $\mathcal{X}$ are defined by the following grammar, where $x, y, \ldots$ range over the infinite set of *sockets*, and $\alpha, \beta, \ldots$ over the infinite set of *plugs*.

$$\begin{array}{rcccccc} P, Q & ::= & \langle x.\alpha \rangle & \mid & \widehat{y}P\widehat{\alpha}\cdot\beta & \mid & P\widehat{\alpha}\,[y]\,\widehat{x}Q \mid P\widehat{\alpha} \dagger \widehat{x}Q \\ & & \textit{capsule} & & \textit{export} & & \textit{import} \qquad \textit{cut} \end{array}$$

Diagrammatically, we represent the basic circuits as:



Definition 6.2 The *free sockets* and *free plugs* in a circuit are:

$$\begin{array}{lll} fs(\langle x.\alpha \rangle) & = \{x\} & fp(\langle x.\alpha \rangle) = \{\alpha\} \\ fs(\widehat{x}P\widehat{\beta}\cdot\alpha) & = fs(P) \setminus \{x\} & fp(\widehat{x}P\widehat{\beta}\cdot\alpha) = (fp(P) \setminus \{\beta\}) \cup \{\alpha\} \\ fs(P\widehat{\alpha}\,[y]\,\widehat{x}Q) & = fs(P) \cup \{y\} \cup (fs(Q) \setminus \{x\}) & fp(P\widehat{\alpha}\,[y]\,\widehat{x}Q) = (fp(P) \setminus \{\alpha\}) \cup fp(Q) \\ fs(P\widehat{\alpha} \dagger \widehat{x}Q) & = fs(P) \cup (fs(Q) \setminus \{x\}) & fp(P\widehat{\alpha} \dagger \widehat{x}Q) = (fp(P) \setminus \{\alpha\}) \cup fp(Q) \end{array}$$

A socket x or plug α which is not free is called *bound*, written $x \in bs(P)$ and $\alpha \in bp(P)$. We will write $x \notin fs(P, Q)$ for $x \notin fs(P) \wedge x \notin fs(Q)$.

6.2 Reduction

Reduction takes place through the elimination of cuts. It is important to know when a socket or a plug is introduced, i.e. is connectable. Informally, a circuit P introduces a socket x if P is constructed from subcircuits which do not contain x as free socket: x only occurs at the "top level." This means that P is either an import with a middle connector $[x]$ or a capsule whose left part is x. Similarly, a circuit introduces a plug α if it is an export that "creates" α or a capsule whose right part is α.

Definition 6.3 (INTRODUCTION)

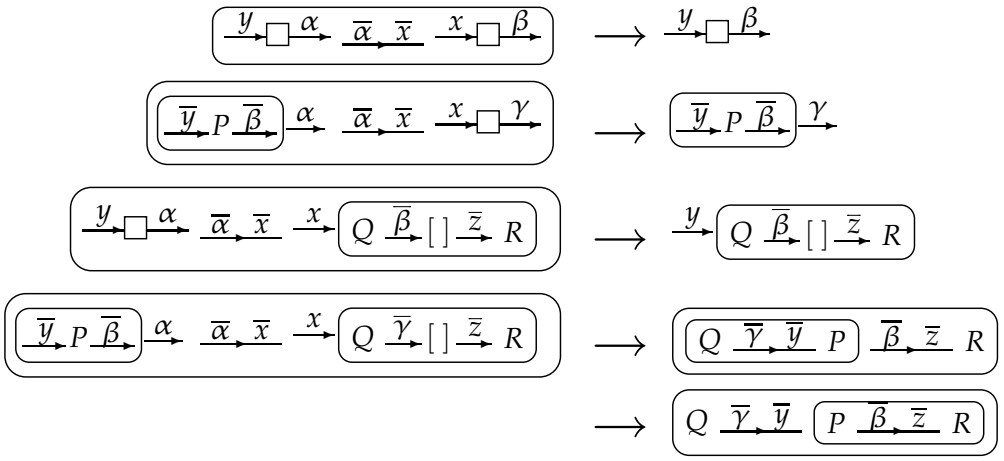
$$\begin{array}{ll} P \text{ introduces } x: & P = \langle x.\beta \rangle \text{ or } P = R\widehat{\alpha}\,[x]\,\widehat{y}Q, \text{ with } x \notin fs(R, Q). \\ P \text{ introduces } \alpha: & P = \langle y.\alpha \rangle \text{ or } P = \widehat{x}Q\widehat{\beta}\cdot\alpha, \text{ with } \alpha \notin fp(Q). \end{array}$$

If a circuit consists of two subcircuits cut together, both of which introduce connectors, then reduction is simple.

Definition 6.4 (LOGICAL REDUCTION)
Assume that the circuits on the left-hand side of the rules introduce the socket x and the plug α:

$$\begin{array}{rrcl}
(cap): & \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle & \rightarrow & \langle y.\beta\rangle \\
(exp): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x.\gamma\rangle & \rightarrow & \widehat{y}P\widehat{\beta}\cdot\gamma \\
(imp): & \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}(P\widehat{\beta}\,[x]\,\widehat{z}Q) & \rightarrow & P\widehat{\beta}\,[y]\,\widehat{z}Q \\
(exp\text{-}imp): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}\,[x]\,\widehat{z}R) & \rightarrow & (Q\widehat{\gamma} \dagger \widehat{y}P)\widehat{\beta} \dagger \widehat{z}R \\
 & & \text{or} & Q\widehat{\gamma} \dagger \widehat{y}(P\widehat{\beta} \dagger \widehat{z}R)
\end{array}$$

Diagrammatically, they look like this:



However, a cut circuit might not introduce its connector. In that case the cut must be propagated through the circuit so that every free instance of that connector is cut with the other circuit. In order to define the propagation rules, we must extend the syntax with two new operators called *activated* cuts:

$$P ::= \ldots \mid P\widehat{\alpha} \not\leftarrow \widehat{x}Q \mid P\widehat{\alpha} \not\rightarrow \widehat{x}Q$$

If two circuits are cut together and they do not both introduce their connectors then reduction according to the logical rules given above cannot occur. First, the cut must be activated.

Definition 6.5 (ACTIVATING THE CUTS)

$$\begin{array}{rcl}
(act\text{-}\mathrm{L}): & P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow P\widehat{\alpha} \not\leftarrow \widehat{x}Q, & \textit{if } P \textit{ does not introduce } \alpha \\
(act\text{-}\mathrm{R}): & P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow P\widehat{\alpha} \not\rightarrow \widehat{x}Q, & \textit{if } Q \textit{ does not introduce } x
\end{array}$$

Notice that both side-conditions might be true, i.e. it might be the case that neither subcircuit introduces its connector. If that is the case then either rule may be used, which constitutes a *critical pair* or *superposition* for reduction in $\mathcal{X}$ and causes the calculus to be non-confluent. This should not be surprising given that cut-elimination in LK has the same property. The LK proof

$$\dfrac{\dfrac{\Gamma, A \vdash_{\mathrm{LK}} B, \Delta}{\Gamma \vdash_{\mathrm{LK}} A \rightarrow B, \Delta}(\rightarrow R) \qquad \dfrac{\Gamma \vdash_{\mathrm{LK}} A, \Delta \quad \Gamma, B \vdash_{\mathrm{LK}} \Delta}{\Gamma, A \rightarrow B \vdash_{\mathrm{LK}} \Delta}(\rightarrow L)}{\Gamma \vdash_{\mathrm{LK}} \Delta}(cut)$$

reduces to both

$$\dfrac{\Gamma \vdash_{\mathrm{LK}} A, \Delta \qquad \dfrac{\Gamma, A \vdash_{\mathrm{LK}} B, \Delta \quad \dfrac{\Gamma, B \vdash_{\mathrm{LK}} \Delta}{\Gamma, A, B \vdash_{\mathrm{LK}} \Delta}}{\Gamma, A \vdash_{\mathrm{LK}} \Delta}}{\Gamma \vdash_{\mathrm{LK}} \Delta} \quad \text{and} \quad \dfrac{\dfrac{\Gamma \vdash_{\mathrm{LK}} A, \Delta}{\Gamma \vdash_{\mathrm{LK}} A, B, \Delta} \quad \Gamma, A \vdash_{\mathrm{LK}} B, \Delta}{\Gamma \vdash_{\mathrm{LK}} B, \Delta \qquad \Gamma, B \vdash_{\mathrm{LK}} \Delta}}{\Gamma \vdash_{\mathrm{LK}} \Delta}$$

Other calculi based on classical logic, such as Parigot's $\lambda\mu$, do not have this property. Even other non-confluent calculi based on classical logic, such as Herbelin's $\overline{\lambda}\mu\tilde{\mu}$-calculus, fail to represent both possible proof reductions given above.

Once activated, cuts must be propagated through the subcircuits until they either disappear or are eliminated by a logical rule.

Definition 6.6 (PROPAGATION REDUCTION)

Left propagation

$$
\begin{array}{rrcl}
(\not\leftarrow d): & \langle y.\alpha\rangle\widehat{\alpha} \not\leftarrow \widehat{x}P & \rightarrow & \langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P \\
(cap\not\leftarrow): & \langle y.\beta\rangle\widehat{\alpha} \not\leftarrow \widehat{x}P & \rightarrow & \langle y.\beta\rangle, \quad \beta \neq \alpha \\
(\textit{exp-outs}\not\leftarrow): & (\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha} \not\leftarrow \widehat{x}P & \rightarrow & (\widehat{y}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}P, \gamma \textit{ fresh} \\
(\textit{exp-ins}\not\leftarrow): & (\widehat{y}Q\widehat{\beta}\cdot\gamma)\widehat{\alpha} \not\leftarrow \widehat{x}P & \rightarrow & \widehat{y}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\cdot\gamma, \gamma \neq \alpha \\
(imp\not\leftarrow): & (Q\widehat{\beta}\,[z]\,\widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}P & \rightarrow & (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\,[z]\,\widehat{y}(R\widehat{\alpha} \not\leftarrow \widehat{x}P) \\
(cut\not\leftarrow): & (Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}P & \rightarrow & (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \not\leftarrow \widehat{x}P)
\end{array}
$$

Right propagation

$$
\begin{array}{rrcll}
(d\not\rightarrow): & P\widehat{\alpha} \not\rightarrow \widehat{x}\langle x.\beta\rangle & \rightarrow & P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle & \\
(\not\rightarrow cap): & P\widehat{\alpha} \not\rightarrow \widehat{x}\langle y.\beta\rangle & \rightarrow & \langle y.\beta\rangle, & y \neq x \\
(\not\rightarrow exp): & P\widehat{\alpha} \not\rightarrow \widehat{x}(\widehat{y}Q\widehat{\beta}\cdot\gamma) & \rightarrow & \widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta}\cdot\gamma & \\
(\not\rightarrow \textit{imp-outs}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(Q\widehat{\beta}\,[x]\,\widehat{y}R) & \rightarrow & P\widehat{\alpha} \dagger \widehat{z}((P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}R)), & z \textit{ fresh} \\
(\not\rightarrow \textit{imp-ins}): & P\widehat{\alpha} \not\rightarrow \widehat{x}(Q\widehat{\beta}\,[z]\,\widehat{y}R) & \rightarrow & (P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}R), & z \neq x \\
(\not\rightarrow cut): & P\widehat{\alpha} \not\rightarrow \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) & \rightarrow & (P\widehat{\alpha} \not\rightarrow \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \not\rightarrow \widehat{x}R) &
\end{array}
$$

The rules $(\textit{exp-outs}\not\leftarrow)$ and $(\not\rightarrow \textit{imp-outs})$ deserve some attention. In the left-hand side of $(\textit{exp-outs}\not\leftarrow)$ α may not be introduced, which means that α could appear free inside Q. The appearance outside Q (i.e. after the dot) is dealt with separately by creating a new name γ. Note that the cut associated with that γ is then unactivated. This is because although we know that γ is introduced, we do not know if x is introduced or not so the cut may need to be right-activated to continue. A similar reasoning holds for x in $(\not\rightarrow \textit{imp-outs})$ so a new name z is created and the external cut is not active.

6.3 Reduction Strategies

As mentioned above, it is sometimes the case that both activation rules are valid at the same time. This is similar to the case described in §5.1 in which a lambda term might be reduced in more than one way (although in the λ-calculus one always ends up at the same normal form regardless of which choice one makes because β-reduction is confluent, whereas this is not the case in $\mathcal{X}$.) So, analogously, there are two different reduction strategies in $\mathcal{X}$. Consider a term $P\widehat{\alpha} \dagger \widehat{x}Q$ where P does not introduce α and Q does not introduce x: intuitively, CBV tends to push Q through P and CBN tends to do the opposite.

Definition 6.7 (CALL-BY-VALUE AND CALL-BY-NAME)

- If a cut can be activated in two ways, the CBV strategy only allows it to activated via $(\textit{act}\text{-L})$; we write $P \rightarrow_{\text{V}} Q$ in that case. We can formulate this by replacing the rule $(\textit{act}\text{-R})$ by:

$$(\textit{act}\text{-R}): P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow P\widehat{\alpha} \not\rightarrow \widehat{x}Q, \textit{ if } P \textit{ introduces } \alpha \textit{ and } Q \textit{ does not introduce } x.$$

- If a cut can be activated in two ways, the CBN strategy only allows it to activated via $(\textit{act}\text{-R})$; we write $P \rightarrow_{\text{N}} Q$ in that case. We can formulate this by replacing the rule $(\textit{act}\text{-L})$ by:

$$(\textit{act}\text{-L}): P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow P\widehat{\alpha} \not\leftarrow \widehat{x}Q, \textit{ if } P \textit{ does not introduce } \alpha \textit{ and } Q \textit{ introduces } x.$$

6.4 Typing for $\mathcal{X}$

The typing for $\mathcal{X}$ exhibits a Curry-Howard Isomorphism with implicative classical logic in the framework of LK$(\rightarrow)$ in the same way that the simple typing system for the λ-calculus is isomorphic to implicative intuitionistic logic in a natural deduction framework.

Definition 6.8 (TYPING FOR $\mathcal{X}$)
A *context of sockets* Γ is a mapping from sockets to types, denoted as a finite set of statements of the form $x{:}A$, such that the subjects of the statements (e.g. x) are distinct. We write $\Gamma, x{:}A$ for the context defined by:

$$\begin{aligned} \Gamma, x{:}A &= \Gamma \cup \{x{:}A\}, \text{ if } \Gamma \text{ is not defined on } x \\ &= \Gamma, \qquad\qquad \text{otherwise} \end{aligned}$$

(Notice that the second case implies that $x{:}A \in \Gamma$.) So, when writing a context as $\Gamma, x{:}A$, this implies that $x{:}A \in \Gamma$, or Γ is not defined on x. When we write Γ_1, Γ_2 we mean the union of Γ_1 and Γ_2 when Γ_1 and Γ_2 are coherent (if Γ_1 contains $x{:}A_1$ and Γ_2 contains $x{:}A_2$ then $A_1 = A_2$).

Contexts of *plugs* Δ are defined in a similar way.

Type judgements are expressed via a ternary relation $P \vdots \Gamma \vdash \Delta$, where P is a circuit whose free connectors can be found in Γ and Δ with their types. We say that P is the *witness* of this judgement.

Type assignment for $\mathcal{X}$ is defined by the following sequent calculus:

$$(cap): \frac{}{\langle x.\alpha \rangle \vdots \Gamma, x{:}A \vdash \alpha{:}A, \Delta}$$

$$(imp): \frac{P \vdots \Gamma \vdash \alpha{:}A, \Delta \quad Q \vdots \Gamma, x{:}B \vdash \Delta}{P\widehat{\alpha}\,[y]\,\widehat{x}Q \vdots \Gamma, y{:}A{\rightarrow}B \vdash \Delta}$$

$$(exp): \frac{P \vdots \Gamma, x{:}A \vdash \alpha{:}B, \Delta}{\widehat{x}P\widehat{\alpha}{\cdot}\beta \vdots \Gamma \vdash \beta{:}A{\rightarrow}B, \Delta}$$

$$(cut): \frac{P \vdots \Gamma \vdash \alpha{:}A, \Delta \quad Q \vdots \Gamma, x{:}A \vdash \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q \vdots \Gamma \vdash \Delta}$$

We write $P \vdots \Gamma \vdash \Delta$ if there exists a derivation that has this judgement in the bottom line. There is no notion of type for P itself, instead the derivable statement shows how P is connectable.

7 Parigot's $\lambda\mu$-calculus

Parigot [13] extended the λ-calculus in such a way that its typing rules would correspond to classical logic. However, he retained the natural deduction framework. In order to cope with the possibility of multiple alternative conclusions, he introduced the notion of *activation*. This allows only one conclusion to be the *active* conclusion at any one time, as in normal natural deduction, but allows multiple alternative *passive* conclusions to exist as well. The active conclusion is separated from the other conclusions in Parigot's sequents by '$|$' which is to be understood as the logical connective $\vee$ (just as commas on the right-hand side of a sequent are usually understood.) The rationale for this system becomes clear when we extend the rules of natural deduction to deal with classical logic.

As mentioned in §3, classical logic can be thought of as intuitionistic logic with the addition of a rule expressing the law of excluded middle. However, rather than the law of excluded middle, it is possible to add a rule for double negation elimination or proof by contradiction instead and get the same result. Given one of those three rules, the other two become admissible. (This is not quite true, Ariola and Herbelin [1] distinguish between "weak classical", "minimal classical" and "full classical" axioms and the logics that result from their addition to intuitionistic logic. However, for our purposes here we do not need to make such distinctions.)

Proof by contradiction is a rule which allows you to state A if you can prove that the negation of A ($\neg A$) implies contradiction ($\bot$). In intuitionistic logic, if you can prove that a proposition implies contradiction that only allows you to state the negation of the proposition (in fact, this is the definition of negation: $\neg A = A \rightarrow \bot$.) Thus if $\neg A \rightarrow \bot$ you may state $\neg\neg A$ but you may not immediately state A. Double negation elimination is the rule that allows you to state A if you can prove $\neg\neg A$.

Parigot chose to add a rule for proof by contradiction:

$$(PC) : \frac{\Gamma, A \Rightarrow \bot \vdash \bot}{\Gamma \vdash A}$$

Assumptions can now be discharged either through the $(\Rightarrow I)$-rule or, if they are of the form $(A \Rightarrow \bot)$, through the above (PC)-rule. However, in classical logic, $(A \Rightarrow \bot) \Rightarrow B$ is logically equivalent to $A \vee B$. So a single conclusion sequent such as

$$\Gamma, A_1 \Rightarrow \bot, A_2 \Rightarrow \bot, \ldots, A_n \Rightarrow \bot \vdash B$$

is logically equivalent to the multiple conclusion sequent

$$\Gamma \vdash B \mid A_1, A_2, \ldots, A_n$$

Thus proof by contradiction becomes

$$(PC) : \frac{\Gamma \vdash \bot \mid A, \Delta}{\Gamma \vdash A \mid \Delta}$$

which exhibits the neutrality of $\bot$ for disjunction.

However, when we assumed $A \Rightarrow \bot$, we may have wanted to use it not for proof by contradiction but actually as an implication as such:

$$\frac{\dfrac{}{\Gamma, A \Rightarrow \bot \vdash A \Rightarrow \bot} \qquad \Gamma, A \Rightarrow \bot \vdash A}{\Gamma, A \Rightarrow \bot \vdash \bot}$$

And so we also need this rule:

$$(\bot I): \frac{\Gamma \vdash A \mid A, \Delta}{\Gamma \vdash \bot \mid A, \Delta}$$

where again the neutrality of $\bot$ for disjunction is exhibited. Alternatively, it is clear that if we have a rule for *PC*, which requires the existence of $\bot$, then we also need a rule for $\bot$ introduction. Bearing in mind that the sequent $\Gamma \vdash A \mid A, \Delta$ is equivalent to the single-conclusion sequent $\Gamma, \neg\Delta, \neg A \vdash A$, the above rule is the obvious choice.

Parigot extended the typing system of λ-calculus to include the pseudo-type $\bot$ and added syntactic constructs to act as witnesses to the two new rules: (PC) and $(\bot I)$. He called the result the $\lambda\mu$-calculus as it uses two disjoint sets of variables: as before, Roman letters designate λ variables, but now there are also Greek letters designating μ variables.

Definition 7.1 (TERMS OF $\lambda\mu$)

$$M, N ::= x \mid \lambda x.M \mid MN \mid [\alpha]M \mid \mu\alpha.M$$

Definition 7.2 (TYPING RULES FOR $\lambda\mu$)
Type assignment for $\lambda\mu$ is defined by the following natural deduction system. There is an *active* conclusion, labelled by a term of his calculus, and the alternative conclusions are labelled by the set of Greek variables $\alpha, \beta, \gamma, \ldots$

$$(Ax): \frac{}{\Gamma, x{:}A \vdash_{\lambda\mu} x{:}A \mid \Delta} \qquad (\rightarrow I): \frac{\Gamma, x{:}A \vdash_{\lambda\mu} M{:}B \mid \Delta}{\Gamma \vdash_{\lambda\mu} \lambda x.M{:}A\rightarrow B \mid \Delta}$$

$$(\rightarrow E): \frac{\Gamma \vdash_{\lambda\mu} M{:}A\rightarrow B \mid \Delta \quad \Gamma \vdash_{\lambda\mu} N{:}A \mid \Delta}{\Gamma \vdash_{\lambda\mu} MN{:}B \mid \Delta}$$

$$(\bot I): \frac{\Gamma \vdash_{\lambda\mu} M{:}A \mid \Delta}{\Gamma \vdash_{\lambda\mu} [\alpha]M{:}\bot \mid \alpha{:}A, \Delta} \qquad (PC): \frac{\Gamma \vdash_{\lambda\mu} M{:}\bot \mid \alpha{:}A, \Delta}{\Gamma \vdash_{\lambda\mu} \mu\alpha.M{:}A \mid \Delta}$$

We can think of $[\alpha]M$ as storing the type of M among the alternative conclusions by giving it a name: the set of Greek variables is called the set of *name* variables and the rule corresponding to $\bot I$ is called *naming*. Also, $\mu\alpha.M$ binds α in M and the rule corresponding to *PC* is called *μ-abstraction*.

Note that $\bot$ is not a real type: no term may have a type of $\bot$ except as a result of the $(\bot I)$ rule above. Some authors emphasise this point by not including $[\alpha]M$ in the category of terms but instead call such expressions "commands" or just "named terms".

Definition 7.3 (REDUCTION IN $\lambda\mu$)
In addition to the logical β-reduction from the λ-calculus, $\lambda\mu$ also has structural μ-reduction and two simplification rules.

$$\begin{aligned}
\textit{logical } (\beta): &\quad (\lambda x.M)N &&\rightarrow M[N/x] \\
\textit{structural } (\mu): &\quad (\mu\alpha.M)N &&\rightarrow \mu\gamma.M[N{\cdot}\gamma/\alpha] \\
\textit{renaming}: &\quad \mu\alpha.[\beta](\mu\gamma.[\delta]M) &&\rightarrow \mu\alpha.[\delta]M[\beta/\gamma] \\
\textit{erasing}: &\quad \mu\alpha.[\alpha]M &&\rightarrow M \textit{ if } \alpha \textit{ does not occur in } M.
\end{aligned}$$

The notation $M[N{\cdot}\gamma/\alpha]$ denotes the *recursive* substitution of every named term $[\alpha]M'$ found within M by $[\gamma](M'N)$ (γ is a fresh variable.) This is an unusual reduction step that substitutes terms for terms rather than the more usual terms for variables. In β-reduction we search through the body of the abstraction (M) for free occurrences of the bound variable (x) and each time we find one we replace

it with the argument (N). In μ-reduction we recursively search through the body of the abstraction (M) and each time we find a term named by the bound name variable ($[\alpha]M'$) we replace it with the application of that term to the argument ($M'N$) named with a fresh name variable, which we bind at the outermost level.

It is not obvious that such a strange rule should do what we want, so I will provide a somewhat contrived example to show how it works. Consider the following natural deduction proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash A{\to}B \mid A{\to}B, \Delta}}{\Gamma \vdash \bot \mid A{\to}B, \Delta}}{\Gamma \vdash A{\to}B \mid A{\to}B, \Delta}}{\Gamma \vdash \bot \mid A{\to}B, \Delta}}{\Gamma \vdash A{\to}B \mid \Delta}}{\Gamma, A \vdash A{\to}B \mid \Delta}}{\Gamma, A \vdash B \mid \Delta} \qquad \dfrac{}{\Gamma, A \vdash A \mid \Delta}$$

Clearly there is plenty of opportunity for reduction. Here is the proof inhabited with $\lambda\mu$ terms:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\lambda\mu} x{:}A{\to}B \mid \alpha{:}A{\to}B, \beta{:}A{\to}B, \Delta}}{\Gamma \vdash_{\lambda\mu} [\alpha]x{:}\bot \mid \alpha{:}A{\to}B, \beta{:}A{\to}B, \Delta}}{\Gamma \vdash_{\lambda\mu} \mu\beta.[\alpha]x{:}A{\to}B \mid \alpha{:}A{\to}B, \Delta}}{\Gamma \vdash_{\lambda\mu} [\alpha]\mu\beta.[\alpha]x{:}\bot \mid \alpha{:}A{\to}B, \Delta}}{\Gamma \vdash_{\lambda\mu} \mu\alpha.[\alpha]\mu\beta.[\alpha]x{:}A{\to}B \mid \Delta}}{\Gamma, y{:}A \vdash_{\lambda\mu} \mu\alpha.[\alpha]\mu\beta.[\alpha]x{:}A{\to}B \mid \Delta} \qquad \dfrac{}{\Gamma, y{:}\, A \vdash_{\lambda\mu} y{:}A \mid \Delta}}{\Gamma, y{:}A \vdash_{\lambda\mu} (\mu\alpha.[\alpha]\mu\beta.[\alpha]x)y{:}B \mid \Delta}$$

Now we can demonstrate the recursive nature of the structural μ-reduction:

$$\begin{array}{rll} (\mu\alpha.[\alpha]\mu\beta.[\alpha]x)y & \to \mu\gamma.[\gamma]((\mu\beta.[\gamma]xy)y) & \textit{(structural)} \\ & \to \mu\gamma.[\gamma](\mu\delta.[\gamma]xy) & \textit{(structural)} \\ & \to \mu\gamma.[\gamma]xy & \textit{(renaming)} \\ & \to xy & \textit{(erasing)} \end{array}$$

Which corresponds to the much more sensible proof:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash A{\to}B \mid \Delta}}{\Gamma, A \vdash A{\to}B \mid \Delta} \qquad \dfrac{}{\Gamma, A \vdash A \mid \Delta}}{\Gamma, A \vdash B \mid \Delta}$$

8 Bierman's Abstract Machine

Bierman has shown [5] that $\lambda\mu$ can be interpreted as "a typed λ-calculus which is able to save and restore the runtime environment." In order to demonstrate this we must formally introduce the idea of an *evaluation context* that was mentioned in §2.

8.1 Evaluation Contexts

We write $E[\bullet]$ to signify an evaluation context in which $\bullet$ is the 'hole' - the part of the calculation for which we are waiting. The fundamental property of evaluation contexts is this:

Lemma 8.1 Every closed term, M, is either a value, V, or is uniquely of the form $E[R]$ where $E[\bullet]$ is an evaluation context and R is a redex.

How we constitute a system of evaluation contexts depends on the reduction strategy that we want to use. Later, when we translate between $\mathcal{X}$ and $\lambda\mu$, it will become necessary to pick a single reduction strategy. This is because $\mathcal{X}$ is symmetric and not confluent whereas $\lambda\mu$ is asymmetric and confluent, so $\lambda\mu$ cannot completely represent reduction in $\mathcal{X}$. However, if we restrict ourselves to a single reduction strategy, either CBV or CBN, then reduction in $\mathcal{X}$ will be confluent and we can construct consistent and complete translations. Which system we choose is largely arbitrary but I have chosen to use CBN. The system of evaluation contexts that Bierman presents in his paper is for a CBV system so choosing CBN allows me to do some more original work.

Definition 8.2 (SYNTACTIC CLASSES FOR CBN)

$$\begin{array}{rrcl} \text{Terms:} & M, N & ::= & x \mid \lambda x.M \mid M\,N \mid [\alpha]M \mid \mu\alpha.M \\ \text{Values:} & V & ::= & x \mid \lambda x.M \\ \text{Evaluation Contexts:} & E & ::= & \bullet \mid EM \\ \text{Redexes:} & R & ::= & (\lambda x.M)N \mid [\alpha]M \mid \mu\alpha.M \end{array}$$

Contrast this with CBV, in which an evaluation context may also take the form VE but a redex must take the form VV (or a μ-abstraction or naming.) As expected, in the CBV variant a reduction can only occur when the argument is a value but this restriction does not apply in CBN. Equally expectedly, in the CBN variant the redex always appears on the left, whereas in CBV it can appear to the right of a value. For example, $[\lambda x.M\,\bullet]$ is a valid context in CBV but not in CBN. In CBN the leftmost redex is always reduced first.

8.2 The Machine

We represent the state of the abstract machine by a pair: the current evaluation context and a function. The function maps name variables to evaluation contexts. If Σ is such a function, then $\Sigma \uplus \{\alpha \mapsto E[\bullet]\}$ denotes that function extended with the mapping $\alpha \mapsto E[\bullet]$.

The single-step reduction rules for our machine are as follows:

$$\begin{array}{rcl} \langle E[(\lambda x.M)N],\, \Sigma\rangle & \Rightarrow & \langle E[M[N/x]],\, \Sigma\rangle \\ \langle E[\mu\alpha.M],\, \Sigma\rangle & \Rightarrow & \langle M,\, \Sigma \uplus \{\alpha \mapsto E[\bullet]\}\rangle \\ \langle E[[\alpha]M],\, \Sigma \uplus \{\alpha \mapsto E'[\bullet]\}\rangle & \Rightarrow & \langle E'[M],\, \Sigma \uplus \{\alpha \mapsto E'[\bullet]\}\rangle \end{array}$$

The first rule is just standard logical β-reduction from the λ-calculus. In the second rule, we see that μ-abstraction causes a pointer to the current evaluation context to be saved; then evaluation continues inside the body of the μ-abstraction. In the third rule, naming causes the current evaluation context to be discarded and replaced by a previously referenced context. Intuitively, we can see that this captures the recursive nature of structural μ-reduction in the $\lambda\mu$ calculus.

Here is the $\lambda\mu$ term that was used as an example in §7 being run on the machine:

$$\begin{aligned}
& \langle (\mu\alpha.[\alpha]\mu\beta.[\alpha]x)y,\ \Sigma\rangle \\
\Rightarrow\ & \langle [\alpha]\mu\beta.[\alpha]x,\ \Sigma \uplus \{\alpha \mapsto [\bullet y]\}\rangle \\
\Rightarrow\ & \langle (\mu\beta.[\alpha]x)y,\ \Sigma \uplus \{\alpha \mapsto [\bullet y]\}\rangle \\
\Rightarrow\ & \langle [\alpha]x,\ \Sigma \uplus \{\alpha \mapsto [\bullet y],\ \beta \mapsto [\bullet y]\}\rangle \\
\Rightarrow\ & \langle xy,\ \Sigma \uplus \{\alpha \mapsto [\bullet y],\ \beta \mapsto [\bullet y]\}\rangle
\end{aligned}$$

If $[\alpha]N$ occurs within $\mu\alpha.M$, then the μ-abstraction looks like a *try/catch* block, as described in §2, and the naming looks like throwing an exception.

In order to investigate continuations in $\mathcal{X}$ then, it makes sense to look at the relationship between $\mathcal{X}$ and $\lambda\mu$.

9 From $\lambda\mu$ to $\mathcal{X}$

We have seen that μ-abstraction and naming in $\lambda\mu$ correspond to saving and restoring continuations. So now we would like to know how the equivalent operations are represented in $\mathcal{X}$.

We do not have the pseudo-type $\perp$ in our typing system for $\mathcal{X}$ so we cannot directly represent a single μ-abstraction or naming operation. However, if we extend the syntax of $\mathcal{X}$ to represent negation, we can use a proof containing double negation elimination to represent the process of naming immediately followed by μ-abstraction. The rationale for this will become clear when we construct the sequent proof below.

First, we note that the negation of an assumption on the left-hand side of a sequent moves it to the right-hand side to be a possible conclusion. Similarly, the negation of a conclusion moves it to the left-hand side:

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$$

Next, we note that in our typing for $\mathcal{X}$, the introduction of a socket witnesses the creation of an assumption and the introduction of a plug witnesses the creation of a conclusion. Similarly, the elimination of assumptions and conclusions corresponds to the binding of sockets and plugs. So, the natural representation of negation in $\mathcal{X}$ would involve the binding of a plug and the introduction of a socket or vice versa.

Thus we extend the syntax of $\mathcal{X}$ to include the following two constructs:

$$P \ ::= \ \ldots \mid x\cdot P\widehat{\alpha} \mid \widehat{x}P\cdot\alpha$$

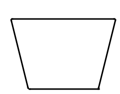
Which we type like this:

$$\frac{P \ :\cdot\ \Gamma \vdash \alpha{:}A, \Delta}{x\cdot P\widehat{\alpha} \ :\cdot\ \Gamma, x{:}\neg A \vdash \Delta} \qquad \frac{P \ :\cdot\ \Gamma, x{:}A \vdash \Delta}{\widehat{x}P\cdot\alpha \ :\cdot\ \Gamma \vdash \alpha{:}\neg A, \Delta}$$

And reduce like this:

$$(\widehat{y}P\cdot\beta)\widehat{\beta} \dagger \widehat{x}(x\cdot Q\widehat{\alpha}) \ \rightarrow \ Q\widehat{\alpha} \dagger \widehat{y}P$$

We can now interpret naming and μ-abstraction by constructing a witness for double negation elimination. We take an assumption A and negate it twice to produce $\neg\neg A$ then we eliminate the double negation. This is expressed in LK sequents like so:



$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}}{\Gamma \vdash \neg\neg A, \Delta} \qquad \dfrac{\dfrac{\Gamma, A \vdash A, \Delta}{\Gamma \vdash \neg A, A, \Delta}}{\Gamma, \neg\neg A \vdash A, \Delta}}{\Gamma \vdash A, \Delta}$$

Compare the above with the sequents for (PC) and $(\perp I)$ given in §7.

This gives us the following witnesses:

$$\dfrac{\dfrac{\dfrac{\boxed{}}{P \mathrel{:}\cdot \Gamma \vdash \alpha{:}A, \Delta}}{x \cdot P\widehat{\alpha} \mathrel{:}\cdot \Gamma, x{:}\neg A \vdash \Delta}}{\widehat{x}(x \cdot P\widehat{\alpha}) \cdot \delta \mathrel{:}\cdot \Gamma \vdash \delta{:}\neg\neg A, \Delta} \qquad \dfrac{\dfrac{\langle y.\beta\rangle \mathrel{:}\cdot \Gamma, y{:}A \vdash \beta{:}A, \Delta}{\widehat{y}\langle y.\beta\rangle \cdot \gamma \mathrel{:}\cdot \Gamma \vdash \gamma{:}\neg A, \beta{:}A, \Delta}}{z \cdot (\widehat{y}\langle y.\beta\rangle \cdot \gamma)\widehat{\gamma} \mathrel{:}\cdot \Gamma, z{:}\neg\neg A \vdash \beta{:}A, \Delta}$$

$$\begin{aligned} & (\widehat{x}(x \cdot P\widehat{\alpha}) \cdot \delta)\widehat{\delta} \dagger \widehat{z}(z \cdot (\widehat{y}\langle y.\beta\rangle \cdot \gamma)\widehat{\gamma}) \mathrel{:}\cdot \Gamma \vdash \beta{:}A, \Delta \\ \to\ & (\widehat{y}\langle y.\beta\rangle \cdot \gamma)\widehat{\gamma} \dagger \widehat{x}(x \cdot P\widehat{\alpha}) \mathrel{:}\cdot \Gamma \vdash \beta{:}A, \Delta \\ \to\ & P\widehat{\alpha} \dagger \widehat{y}\langle y.\beta\rangle \mathrel{:}\cdot \Gamma \vdash \beta{:}A, \Delta \end{aligned}$$

So we see that naming then μ-abstraction in $\lambda\mu$, which causes a pointer to the current continuation to be saved on Bierman's machine, corresponds to a renaming in $\mathcal{X}$. (Note that when the term is run on Bierman's machine the μ-abstraction will be reduced before the naming: when I say "naming followed by μ-abstraction" I am referring to the order of their appearance in the construction of the term.) Later (in §11) we will see that a reference to the current continuation is saved when a plug is bound, so it makes sense to see here that naming then μ-abstraction corresponds to the binding of a plug and its replacement by a free plug of a different name. We could in fact make this more explicit by introducing a new syntactic construct:

$$P ::= \ldots \mid P\widehat{\alpha} \cdot \beta$$

But since a renaming does the same thing (and avoids the need to create new propagation rules to go with the new construct) there is little point in doing so. Note that we do not need to retain our extension of $\mathcal{X}$ that represents negation, that merely provides a rationale for the above representation of saving a continuation.

10 From $\mathcal{X}$ to $\lambda\mu$

We have a translation from $\mathcal{X}$ into $\lambda\mu$ [15] but it uses the version of $\lambda\mu$ in which *naming* and *μ-abstraction* are combined in a single operation. As Bierman's abstract machine for $\lambda\mu$ [5] treats those operations separately, we must use a translation from $\mathcal{X}$ into a version of $\lambda\mu$ that keeps them separate. The translation of terms is the same but the proofs of its consistency are no longer valid as we are translating into a version of $\lambda\mu$ with a different type system (a system containing the pseudo-type $\perp$.) The proof that reduction in $\mathcal{X}$ is preserved in the target calculus is the same in both cases (since we are using the same translation and both versions of $\lambda\mu$ reduce in the same way) but a new proof that types are preserved is needed (since the different versions of $\lambda\mu$ have different typing systems.) As before, we restrict ourselves to the CBN subsystem.

10.1 The Translation

The translation consists of two stages. First we use a double negation translation (see §3) then we recover the types. I know of no theoretical reason why such a two-stage translation should be necessary (since we are translating from a classical logic system to another classical logic system) but in practice it seems to be. The fact that $\lambda\mu$ is based on natural deduction seems to require a translation to intuitionistic logic then a recovery back to classical logic. As we are using a double negation translation, we need some way to represent negation. We cannot use $\perp$ in an implication so we extend the typing system to include a constant Ω.

Definition 10.1 (NEGATION)
Let Ω be a type constant. Then if T is a type

$$\neg T \stackrel{\Delta}{=} T{\rightarrow}\Omega$$

Definition 10.2 (TRANSLATION OF TERMS)
The notation $\mu!.M$ is shorthand for $\mu\eta.M$ where η is a fresh name variable wrt M of type Ω.

$$\begin{aligned}
\llbracket\langle x.\alpha\rangle\rrbracket^{\mu'}_{\mathrm{N}} &\stackrel{\Delta}{=} \lambda v.(\mu!.([\alpha](\lambda f.fv)))\\
\llbracket P\widehat{\beta}\,[x]\,\widehat{y}Q\rrbracket^{\mu'}_{\mathrm{N}} &\stackrel{\Delta}{=} \lambda v.\mu!.[\omega]\lambda y.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}\, v\,\mu\beta.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}
\end{aligned}$$

$$\begin{aligned}
\llbracket\langle x.\alpha\rangle\rrbracket^{\mu}_{\mathrm{N}} &\stackrel{\Delta}{=} [\omega]x\,\llbracket\langle x.\alpha\rangle\rrbracket^{\mu'}_{\mathrm{N}}\\
\llbracket\widehat{y}P\widehat{\beta}\cdot\alpha\rrbracket^{\mu}_{\mathrm{N}} &\stackrel{\Delta}{=} [\alpha]\lambda f.f(\lambda y.\mu\beta.\llbracket P\rrbracket^{\mu}_{\mathrm{N}})\\
\llbracket P\widehat{\beta}\,[x]\,\widehat{y}Q\rrbracket^{\mu}_{\mathrm{N}} &\stackrel{\Delta}{=} [\omega]x\,\llbracket P\widehat{\beta}\,[x]\,\widehat{y}Q\rrbracket^{\mu'}_{\mathrm{N}}\\
\llbracket P\widehat{\alpha}\dagger\widehat{x}Q\rrbracket^{\mu}_{\mathrm{N}} \stackrel{\Delta}{=} \llbracket P\widehat{\alpha}\nwarrow\widehat{x}Q\rrbracket^{\mu}_{\mathrm{N}} &\stackrel{\Delta}{=} [\omega]\lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}\,\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}\\
\llbracket P\widehat{\alpha}\swarrow\widehat{x}Q\rrbracket^{\mu}_{\mathrm{N}} &\stackrel{\Delta}{=} [\omega]\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}\,\llbracket Q\rrbracket^{\mu'}_{\mathrm{N}}
\end{aligned}$$

Definition 10.3 (TRANSLATION OF TYPES)
The CBN interpretation of a type T is defined by

$$\llbracket T\rrbracket^{\mu}_{\mathrm{N}} \stackrel{\Delta}{=} \neg\neg\llbracket T\rrbracket^{\mu'}_{\mathrm{N}}$$

If X is a type variable, $\llbracket\cdot\rrbracket^{\mu'}_{\mathrm{N}}$ is defined inductively by

$$\begin{aligned}
\llbracket X\rrbracket^{\mu'}_{\mathrm{N}} &\stackrel{\Delta}{=} X\\
\llbracket A{\rightarrow}B\rrbracket^{\mu'}_{\mathrm{N}} &\stackrel{\Delta}{=} \llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\rightarrow}\llbracket B\rrbracket^{\mu}_{\mathrm{N}}
\end{aligned}$$

Note that we duplicate notation: we have two translation functions, one for terms and one for types, both denoted by $\llbracket\cdot\rrbracket^{\mu}_{\mathrm{N}}$. Similarly, we have two functions both denoted by $\llbracket\cdot\rrbracket^{\mu'}_{\mathrm{N}}$. This cannot lead to confusion as it is clear from the context which function is intended and this helps to link the two stages of the translation of terms with the two stages of the translation of types.

10.2 Proof of Type Preservation

Lemma 10.4 (WEAKENING)
The following rule is admissible:

$$(W) : \frac{\Gamma \vdash_{\lambda\mu} M{:}A \mid \Delta}{\Gamma' \vdash_{\lambda\mu} M{:}A \mid \Delta'}$$

for any $\Gamma' \supseteq \Gamma$ *and* $\Delta' \supseteq \Delta$.

Weakening is used frequently in the following proof.

Recall that $\mu!.M$ is shorthand for $\mu\eta.M$ where η is a fresh name variable wrt M of type Ω. Thus some name variables in the proof below may appear to have duplicate names if the shorthand ! is used more than once in the same derivation. However, this is not a problem as in each instance it represents the binding of a fresh variable, so we know that the variable does not appear anywhere else in the term and α-conversion can be used if necessary.

Thus the following rule is admissible:

$$(!) : \frac{\Gamma \vdash_{\lambda\mu} M{:}\bot \mid \Delta}{\Gamma \vdash_{\lambda\mu} \mu!.M{:}\Omega \mid \Delta}$$

Lemma 10.5 *If* $M{:}A$ *then* $\lambda f.fM{:}\neg\neg A$
Proof:

$$\dfrac{\dfrac{}{\Gamma, f{:}A{\to}\Omega \vdash_{\lambda\mu} f{:}A{\to}\Omega \mid \Delta} \qquad \dfrac{\Gamma \vdash_{\lambda\mu} M{:}A \mid \Delta}{\Gamma, f{:}A{\to}\Omega \vdash_{\lambda\mu} M{:}A \mid \Delta}\,(W)}{\dfrac{\Gamma, f{:}A{\to}\Omega \vdash_{\lambda\mu} fM{:}\Omega \mid \Delta}{\Gamma \vdash_{\lambda\mu} \lambda f.fM{:}(A{\to}\Omega){\to}\Omega \mid \Delta}}$$

Thus the following rule is admissible:

$$(*) : \frac{\Gamma \vdash_{\lambda\mu} M{:}A \mid \Delta}{\Gamma \vdash_{\lambda\mu} \lambda f.fM{:}\neg\neg A \mid \Delta}$$

Theorem 10.6 *If* $P :\cdot \Gamma \vdash \Delta$, *then* $[\![\Gamma]\!]^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} [\![P]\!]^{\mu}_{\mathrm{N}}{:}\bot \mid \omega{:}\Omega, [\![\Delta]\!]^{\mu}_{\mathrm{N}}$.
Proof: By induction on the structure of derivations.
Recall that $[\![T]\!]^{\mu}_{\mathrm{N}} \triangleq ([\![T]\!]^{\mu'}_{\mathrm{N}}{\to}\Omega){\to}\Omega$.

(cap) : Then $\langle x.\alpha\rangle :\cdot \Gamma, x{:}A \vdash \alpha{:}A, \Delta$.

$$\dfrac{\dfrac{}{[\![\Gamma]\!]^{\mu}_{\mathrm{N}}, x{:}[\![A]\!]^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} x{:}[\![A]\!]^{\mu}_{\mathrm{N}} \mid [\![\Delta]\!]^{\mu}_{\mathrm{N}}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{[\![\Gamma]\!]^{\mu}_{\mathrm{N}}, v{:}[\![A]\!]^{\mu'}_{\mathrm{N}} \vdash_{\lambda\mu} v{:}[\![A]\!]^{\mu'}_{\mathrm{N}} \mid \omega{:}\Omega, [\![\Delta]\!]^{\mu}_{\mathrm{N}}}}{[\![\Gamma]\!]^{\mu}_{\mathrm{N}}, v{:}[\![A]\!]^{\mu'}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda f.fv{:}[\![A]\!]^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, [\![\Delta]\!]^{\mu}_{\mathrm{N}}}\,(*)}{[\![\Gamma]\!]^{\mu}_{\mathrm{N}}, v{:}[\![A]\!]^{\mu'}_{\mathrm{N}} \vdash_{\lambda\mu} [\alpha](\lambda f.fv){:}\bot \mid \alpha{:}[\![A]\!]^{\mu}_{\mathrm{N}}, \omega{:}\Omega, [\![\Delta]\!]^{\mu}_{\mathrm{N}}}}{[\![\Gamma]\!]^{\mu}_{\mathrm{N}}, v{:}[\![A]\!]^{\mu'}_{\mathrm{N}} \vdash_{\lambda\mu} \mu!.[\alpha](\lambda f.fv){:}\Omega \mid \alpha{:}[\![A]\!]^{\mu}_{\mathrm{N}}, \omega{:}\Omega, [\![\Delta]\!]^{\mu}_{\mathrm{N}}}\,(!)}{[\![\Gamma]\!]^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda v.\mu!.[\alpha](\lambda f.fv){:}([\![A]\!]^{\mu'}_{\mathrm{N}}{\to}\Omega) \mid \alpha{:}[\![A]\!]^{\mu}_{\mathrm{N}}, \omega{:}\Omega, [\![\Delta]\!]^{\mu}_{\mathrm{N}}}}{[\![\Gamma]\!]^{\mu}_{\mathrm{N}}, x{:}[\![A]\!]^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} x\lambda v.\mu!.[\alpha](\lambda f.fv){:}\Omega \mid \alpha{:}[\![A]\!]^{\mu}_{\mathrm{N}}, \omega{:}\Omega, [\![\Delta]\!]^{\mu}_{\mathrm{N}}}}{[\![\Gamma]\!]^{\mu}_{\mathrm{N}}, x{:}[\![A]\!]^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} [\omega]x\lambda v.\mu!.[\alpha](\lambda f.fv){:}\bot \mid \alpha{:}[\![A]\!]^{\mu}_{\mathrm{N}}, \omega{:}\Omega, [\![\Delta]\!]^{\mu}_{\mathrm{N}}}$$

(exp) : Then $\widehat{x}P\widehat{\alpha}\cdot\beta :\cdot \Gamma \vdash \beta{:}A{\to}B, \Delta$, with a subderivation for $P :\cdot \Gamma, x{:}A \vdash \alpha{:}B, \Delta$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{By Induction Hypothesis}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, x{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \alpha{:}\llbracket B\rrbracket^{\mu}_{\mathrm{N}}, \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, x{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket B\rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda x.\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\to}\llbracket B\rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda f.f(\lambda x.\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}){:}\llbracket A{\to}B\rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}\,(*)}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} [\beta]\lambda f.f(\lambda x.\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}){:}\bot \mid \beta{:}\llbracket A{\to}B\rrbracket^{\mu}_{\mathrm{N}}, \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}$$

(imp) : Then $P\widehat{\alpha}\,[y]\,\widehat{x}Q :\cdot \Gamma, y{:}A{\to}B \vdash \Delta$, with subderivations for both $P :\cdot \Gamma \vdash \alpha{:}A, \Delta$ and $Q :\cdot \Gamma, x{:}B \vdash \Delta$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{By Induction Hypothesis}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \alpha{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}, \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\to}\llbracket B\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} v\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket B\rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\vdots}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\to}\llbracket B\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket B\rrbracket^{\mu}_{\mathrm{N}}{\to}\Omega \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\to}\llbracket B\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}v\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\to}\llbracket B\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}v\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}\,(!)}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\to}\llbracket B\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu!.[\omega]\lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}v\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}\,(!)}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda v.\mu!.[\omega]\lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}v\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}(\llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\to}\llbracket B\rrbracket^{\mu}_{\mathrm{N}}){\to}\Omega \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\vdots}$$

$$\dfrac{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, y{:}\llbracket A{\to}B\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} y{:}\llbracket A{\to}B\rrbracket^{\mu}_{\mathrm{N}} \mid \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}} \qquad \vdots}{\dfrac{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, y{:}\llbracket A{\to}B\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} y\lambda v.\mu!.[\omega]\lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}v\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, y{:}\llbracket A{\to}B\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} [\omega]y\lambda v.\mu!.[\omega]\lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}v\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}$$

(cut) : Then $P\widehat{\alpha}\,\dagger\,\widehat{x}Q :\cdot \Gamma \vdash \Delta$, with subderivations for both $P :\cdot \Gamma \vdash \alpha{:}A, \Delta$ and $Q :\cdot \Gamma, x{:}A \vdash \Delta$.
There are two cases:

$(inactive\ or\ right\ cut)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{By Induction Hypothesis}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, x{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket Q\rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}}, x{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}\,(!)}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}{\to}\Omega \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}} \qquad \dfrac{\dfrac{\text{By Induction Hypothesis}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \alpha{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}}, \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket A\rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket\Gamma\rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} [\omega]\lambda x.\mu!.\llbracket Q\rrbracket^{\mu}_{\mathrm{N}}\mu\alpha.\llbracket P\rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \omega{:}\Omega, \llbracket\Delta\rrbracket^{\mu}_{\mathrm{N}}}$$

(*left cut*) :

Lemma 10.7 *If $Q \;\because\; \Gamma, x{:}A \vdash \Delta$ and Q introduces x then $\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket Q \rrbracket^{\mu'}_{\mathrm{N}} {:} \neg \llbracket A \rrbracket^{\mu'}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}$.*

Proof: If Q introduces x then Q is either a *capsule* or an *import*.

$\langle x.\beta \rangle$ So $Q \;\because\; \Gamma, x{:}A \vdash \beta{:}A, \Delta$ and $\llbracket Q \rrbracket^{\mu'}_{\mathrm{N}} \triangleq \lambda v.\mu!.[\beta](\lambda f.fv)$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\overline{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A \rrbracket^{\mu'}_{\mathrm{N}} \vdash_{\lambda\mu} v{:}\llbracket A \rrbracket^{\mu'}_{\mathrm{N}} \mid \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A \rrbracket^{\mu'}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda f.fv{:}\llbracket A \rrbracket^{\mu}_{\mathrm{N}} \mid \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}\,(*)}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A \rrbracket^{\mu'}_{\mathrm{N}} \vdash_{\lambda\mu} [\beta]\lambda f.fv{:}\bot \mid \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket A \rrbracket^{\mu'}_{\mathrm{N}} \vdash_{\lambda\mu} \mu!.[\beta]\lambda f.fv{:}\Omega \mid \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}\,(!)}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda v.\mu!.[\beta]\lambda f.fv{:}\llbracket A \rrbracket^{\mu'}_{\mathrm{N}} \to \Omega \mid \beta{:}\llbracket A \rrbracket^{\mu}_{\mathrm{N}}, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}
$$

$R\widehat{\beta}\,[x]\,\widehat{y}S$ So $Q \;\because\; \Gamma, x{:}B \to C \vdash \Delta$ and $\llbracket Q \rrbracket^{\mu'}_{\mathrm{N}} \triangleq \lambda v.\mu!.[\omega]\lambda y.\mu!.\llbracket S \rrbracket^{\mu}_{\mathrm{N}} v\mu\beta.\llbracket R \rrbracket^{\mu}_{\mathrm{N}}$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{By Induction Hypothesis}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, y{:}\llbracket C \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket S \rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, y{:}\llbracket C \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu!.\llbracket S \rrbracket^{\mu}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}\,(!)}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda y.\mu!.\llbracket S \rrbracket^{\mu}_{\mathrm{N}}{:}(\llbracket C \rrbracket^{\mu}_{\mathrm{N}} \to \Omega) \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}} \quad \dfrac{\dfrac{\dfrac{\text{By Induction Hypothesis}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket R \rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \beta{:}\llbracket B \rrbracket^{\mu}_{\mathrm{N}}, \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu\beta.\llbracket R \rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket B \rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket B \rrbracket^{\mu}_{\mathrm{N}} \to \llbracket C \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} v\mu\beta.\llbracket R \rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket C \rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket B \rrbracket^{\mu}_{\mathrm{N}} \to \llbracket C \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda y.\mu!.\llbracket S \rrbracket^{\mu}_{\mathrm{N}} v\mu\beta.\llbracket R \rrbracket^{\mu}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket B \rrbracket^{\mu}_{\mathrm{N}} \to \llbracket C \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} [\omega]\lambda y.\mu!.\llbracket S \rrbracket^{\mu}_{\mathrm{N}} v\mu\beta.\llbracket R \rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}}, v{:}\llbracket B \rrbracket^{\mu}_{\mathrm{N}} \to \llbracket C \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu!.[\omega]\lambda y.\mu!.\llbracket S \rrbracket^{\mu}_{\mathrm{N}} v\mu\beta.\llbracket R \rrbracket^{\mu}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}\,(!)}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \lambda v.\mu!.[\omega]\lambda y.\mu!.\llbracket S \rrbracket^{\mu}_{\mathrm{N}} v\mu\beta.\llbracket R \rrbracket^{\mu}_{\mathrm{N}}{:}(\llbracket B \rrbracket^{\mu}_{\mathrm{N}} \to \llbracket C \rrbracket^{\mu}_{\mathrm{N}}) \to \Omega \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\text{By Induction Hypothesis}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket P \rrbracket^{\mu}_{\mathrm{N}}{:}\bot \mid \alpha{:}\llbracket A \rrbracket^{\mu}_{\mathrm{N}}, \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu\alpha.\llbracket P \rrbracket^{\mu}_{\mathrm{N}}{:}\llbracket A \rrbracket^{\mu}_{\mathrm{N}} \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}} \quad \dfrac{\text{By Lemma 10.7}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \llbracket Q \rrbracket^{\mu'}_{\mathrm{N}}{:}(\llbracket A \rrbracket^{\mu'}_{\mathrm{N}} \to \Omega) \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} \mu\alpha.\llbracket P \rrbracket^{\mu}_{\mathrm{N}} \llbracket Q \rrbracket^{\mu'}_{\mathrm{N}}{:}\Omega \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}}{\llbracket \Gamma \rrbracket^{\mu}_{\mathrm{N}} \vdash_{\lambda\mu} [\omega]\mu\alpha.\llbracket P \rrbracket^{\mu}_{\mathrm{N}} \llbracket Q \rrbracket^{\mu'}_{\mathrm{N}}{:}\bot \mid \omega{:}\Omega, \llbracket \Delta \rrbracket^{\mu}_{\mathrm{N}}}
$$

□

11 From $\mathcal{X}$ to the Machine

Now that we have a consistent and complete translation from $\mathcal{X}$ to $\lambda\mu$, we can define a translation from $\mathcal{X}$ to Bierman's abstract machine. However, first we must elaborate on the reduction rules for the machine. In Bierman's system, naming discards the current continuation and restores a previously saved continuation. But this relies on the fact that the named variable is bound and therefore already points to some previously saved continuation. This need not be the case: in fact, the translation from $\mathcal{X}$ to $\lambda\mu$ creates terms containing free name variables (signified by ω.) What happens when a term named by a free variable is reduced?

To see how this should work, refer to the idea of a control stack mentioned in §2. We have seen that a μ-abstraction pushes the current context onto the control stack and a naming pops it. When you name a term with a free variable, such as ω in our translation, you are popping the outermost context, i.e. returning control to the top of the stack. Hence a term such as $[\omega]M$ should be understood to mean "run M now".

The translation from $\mathcal{X}$ to the machine is then as follows:

Definition 11.1 (FROM $\mathcal{X}$ TO THE ABSTRACT MACHINE)

$$\llbracket \langle x.\alpha \rangle \rrbracket_{\mathrm{N}}^{\mathrm{BM}} \triangleq \langle x(\lambda v.\mu!.[\alpha](\lambda f.fv)),\ \Sigma \rangle$$

$$\llbracket \widehat{y}P\widehat{\beta}\cdot\alpha \rrbracket_{\mathrm{N}}^{\mathrm{BM}} \triangleq \langle [\alpha]\lambda f.f(\lambda y.\mu\beta.\llbracket P \rrbracket_{\mathrm{N}}^{\mu}),\ \Sigma \rangle$$

$$\llbracket P\widehat{\beta}\,[x]\,\widehat{y}Q \rrbracket_{\mathrm{N}}^{\mathrm{BM}} \triangleq \langle x(\lambda v.\mu!.[\omega]((\lambda y.\mu!.\llbracket Q \rrbracket_{\mathrm{N}}^{\mu})(v\mu\beta.\llbracket P \rrbracket_{\mathrm{N}}^{\mu}))),\ \Sigma \rangle$$

$$\llbracket P\widehat{\alpha} \dagger \widehat{x}Q \rrbracket_{\mathrm{N}}^{\mathrm{BM}} \stackrel{\Delta}{=} \llbracket P\widehat{\alpha} \nwarrow \widehat{x}Q \rrbracket_{\mathrm{N}}^{\mathrm{BM}} \triangleq \langle (\lambda x.\mu!.\llbracket Q \rrbracket_{\mathrm{N}}^{\mu})(\mu\alpha.\llbracket P \rrbracket_{\mathrm{N}}^{\mu}),\ \Sigma \rangle$$

$$\llbracket P\widehat{\alpha} \nearrow \widehat{x}Q \rrbracket_{\mathrm{N}}^{\mathrm{BM}} \triangleq \langle (\mu\alpha.\llbracket P \rrbracket_{\mathrm{N}}^{\mu})\llbracket Q \rrbracket_{\mathrm{N}}^{\mu'},\ \Sigma \rangle$$

We can see from these translations that when a circuit P is cut with a circuit Q, the context surrounding P is saved and referred to by the name of the plug bound in the cut. If that plug appears free in any of the subcircuits of P then that context will be restored whenever the plug is encountered. In effect, the introduction of a plug equates to the storing of a reference to a continuation. While the plug remains free, it will always refer to the outermost continuation (the top of the control stack.) However, when it is bound the plug will then refer to the continuation at the point at which it was bound.

We can illustrate this representation of continuations by comparing reduction in $\mathcal{X}$ with reduction of the translation of $\mathcal{X}$ circuits on the machine.

(cap) :

$$\langle y.\alpha \rangle\widehat{\alpha} \dagger \widehat{x}\langle x.\beta \rangle \rightarrow_{\mathrm{N}} \langle y.\beta \rangle$$

$$\begin{aligned}
\llbracket \langle y.\alpha \rangle\widehat{\alpha} \dagger \widehat{x}\langle x.\beta \rangle \rrbracket_{\mathrm{N}}^{\mathrm{BM}} &\triangleq \langle (\lambda x.\mu!.[\omega]x(\lambda v.\mu!.[\beta]\lambda f.fv))(\mu\alpha.[\omega]y(\lambda w.\mu!.[\alpha]\lambda f.fw)),\ \Sigma \rangle \\
&\Rightarrow \langle \mu!.[\omega](\mu\alpha.[\omega]y(\lambda w.\mu!.[\alpha]\lambda f.fw))(\lambda v.\mu!.[\beta]\lambda f.fv),\ \Sigma \rangle \\
&\Rightarrow \langle [\omega](\mu\alpha.[\omega]y(\lambda w.\mu!.[\alpha]\lambda f.fw))(\lambda v.\mu!.[\beta]\lambda f.fv),\ \Sigma \rangle \\
&\Rightarrow \langle (\mu\alpha.[\omega]y(\lambda w.\mu!.[\alpha]\lambda f.fw))(\lambda v.\mu!.[\beta]\lambda f.fv),\ \Sigma \rangle \\
&\Rightarrow \langle [\omega]y(\lambda w.\mu!.[\alpha]\lambda f.fw),\ \Sigma \uplus \{\alpha \mapsto [\bullet(\lambda v.\mu!.[\beta]\lambda f.fv)]\} \rangle \\
&\Rightarrow \langle y(\lambda w.\mu!.[\alpha]\lambda f.fw),\ \Sigma \uplus \{\alpha \mapsto [\bullet(\lambda v.\mu!.[\beta]\lambda f.fv)]\} \rangle
\end{aligned}$$

So $\llbracket \langle y.\alpha \rangle\widehat{\alpha} \dagger \widehat{x}\langle x.\beta \rangle \rrbracket_{\mathrm{N}}^{\mathrm{BM}}$ runs to exactly the same as $\llbracket \langle y.\alpha \rangle \rrbracket_{\mathrm{N}}^{\mathrm{BM}}$ except that α now points to a continuation in which the result is applied to $\llbracket \langle x.\beta \rangle \rrbracket_{\mathrm{N}}^{\mu'}$. This fits with the idea that α refers to the continuation at the point where it was bound.

(exp) :

$$(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x.\gamma\rangle \to_{\mathrm{N}} \widehat{y}P\widehat{\beta}\cdot\gamma$$

$$\begin{aligned}
[\![(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x.\gamma\rangle]\!]^{\mathrm{BM}}_{\mathrm{N}} &\triangleq \langle(\lambda x.\mu!.[\omega]x(\lambda v.\mu!.[\gamma]\lambda f.fv))(\mu\alpha.[\alpha]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}})),\ \Sigma\rangle \\
&\Rightarrow \langle\mu!.[\omega](\mu\alpha.[\alpha]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}))(\lambda v.\mu!.[\gamma]\lambda f.fv),\ \Sigma\rangle \\
&\Rightarrow \langle[\omega](\mu\alpha.[\alpha]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}))(\lambda v.\mu!.[\gamma]\lambda f.fv),\ \Sigma\rangle \\
&\Rightarrow \langle(\mu\alpha.[\alpha]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}))(\lambda v.\mu!.[\gamma]\lambda f.fv),\ \Sigma\rangle \\
&\Rightarrow \langle[\alpha]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}),\ \Sigma \uplus \{\alpha \mapsto [\bullet(\lambda v.\mu!.[\gamma]\lambda f.fv)]\}\rangle \\
&\Rightarrow \langle(\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}))(\lambda v.\mu!.[\gamma]\lambda f.fv),\ \Sigma \uplus \{\alpha \mapsto [\bullet(\lambda v.\mu!.[\gamma]\lambda f.fv)]\}\rangle \\
&\Rightarrow \langle(\lambda v.\mu!.[\gamma]\lambda f.fv)(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}),\ \Sigma \uplus \{\alpha \mapsto [\bullet(\lambda v.\mu!.[\gamma]\lambda f.fv)]\}\rangle \\
&\Rightarrow \langle\mu!.[\gamma]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}),\ \Sigma \uplus \{\alpha \mapsto [\bullet(\lambda v.\mu!.[\gamma]\lambda f.fv)]\}\rangle \\
&\Rightarrow \langle[\gamma]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}),\ \Sigma \uplus \{\alpha \mapsto [\bullet(\lambda v.\mu!.[\gamma]\lambda f.fv)]\}\rangle
\end{aligned}$$

Here the translation on the machine produces the same result as the $\mathcal{X}$ reduction but we can see that α again refers to the continuation at the point where it was bound. If $\widehat{y}P\widehat{\beta}\cdot\alpha$ does not introduce α then it must occur free inside P. When a free α inside P is encountered it will restore the continuation saved at the point where it was bound (in this case, the cut with $\langle x.\gamma\rangle$ represented by a context in which the result is applied to $[\![\langle x.\gamma\rangle]\!]^{\mu'}_{\mathrm{N}}$.)

$(exp\text{-}outs\not\prec)$:

$$(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \not\prec \widehat{x}Q \to_{\mathrm{N}} (\widehat{y}(P\widehat{\alpha} \not\prec \widehat{x}Q)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}Q$$

$$\begin{aligned}
[\![(\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \not\prec \widehat{x}Q]\!]^{\mathrm{BM}}_{\mathrm{N}} &\triangleq \langle(\mu\alpha.[\alpha]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}))[\![Q]\!]^{\mu'}_{\mathrm{N}},\ \Sigma\rangle \\
&\Rightarrow \langle[\alpha]\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}),\ \Sigma \uplus \{\alpha \mapsto [\bullet[\![Q]\!]^{\mu'}_{\mathrm{N}}]\}\rangle \\
&\Rightarrow \langle(\lambda f.f(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}))[\![Q]\!]^{\mu'}_{\mathrm{N}},\ \Sigma \uplus \{\alpha \mapsto [\bullet[\![Q]\!]^{\mu'}_{\mathrm{N}}]\}\rangle \\
&\Rightarrow \langle[\![Q]\!]^{\mu'}_{\mathrm{N}}(\lambda y.\mu\beta.[\![P]\!]^{\mu}_{\mathrm{N}}),\ \Sigma \uplus \{\alpha \mapsto [\bullet[\![Q]\!]^{\mu'}_{\mathrm{N}}]\}\rangle
\end{aligned}$$

In this case we know that α appears free inside P. It is less obvious that the abstract machine is reducing towards the same result as standard CBN reduction in $\mathcal{X}$ but it is clear that, once again, α refers to the continuation at the point at which it was cut, which will be restored when reduction reaches a free α in P.

However, what is particularly notable is that all free name variables, e.g. ω in these examples, refer to the outermost continuation. They point to the top of the control stack. All other name variables correspond to a plug that is bound in a cut. The plugs are always bound in a cut "above" the point at which they are encountered (i.e. further up the control stack.) Thus the continuations that are restored will always cause control to move up the stack, never sideways. So we do not have first-class continuations in $\mathcal{X}$ at all: we have something much more like exceptions.

12 Conclusion

I have shown that $\mathcal{X}$, like other calculi based on classical logic, has the ability to represent the saving and invoking of continuations. Unlike other calculi, such as $\lambda\mu$ and λ_C, in which such representations are syntactic extensions to a calculus without this ability, the modelling of control flow is an inherent part of reduction in $\mathcal{X}$. Every cut involves the binding of a plug and every binding of a plug turns that plug into a reference to the continuation at the point where it was bound. So our intuition that $\mathcal{X}$ inherently works in a "continuation-passing style" has proved to be correct.

However, $\mathcal{X}$ does not have first-class continuations. It is not possible to model in $\mathcal{X}$ the kind of behaviour that I demonstrated with Scheme in §2. Instead, the continuations in $\mathcal{X}$ are more structured: they are restricted to moving up the control stack like an exception. Although this might be considered a good thing, as it means that $\mathcal{X}$ is in some sense "well-behaved", I would like to see further investigation of the possibility of representing first-class continuations in $\mathcal{X}$. Intriguingly, Griffin suggests [7] that a calculus needs $\bot$ or at least some representation of negation in its typing system in order to represent first-class continuations. It would be interesting to study the extension of $\mathcal{X}$ that represents negation with that in mind.

If I had more time, I would like to have been able to investigate a calculus with first-class continuations, such as Griffin's *Idealised Scheme*, in order to better understand the role of negation and contradiction in the representation of continuations. I would also like to have been able to construct an abstract machine for $\mathcal{X}$, rather than just a translation from $\mathcal{X}$ into an abstract machine designed for a different calculus. Similarly, I would like to study some actual implementations of $\mathcal{X}$. All of the above would have been useful extensions of my project.

References

[1] Z. Ariola and H. Herbelin. Minimal classical logic and control operators. In J. P. J. C. M. Baeten, J. K. Lenstra and G. J. Woeginger, editors, *Lecture Notes in Computer Science*, volume 2719 of *ICALP*, pages 871–885, 2003. Available from: http://citeseer.ist.psu.edu/ariola03minimal.html.

[2] Z. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of continuations and prompts. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 40–53, New York, NY, USA, 2004. ACM Press.

[3] J. Avigad. Classical and constructive logic. Lecture notes, September 2000. Available from: http://www.cs.cmu.edu/∼fp/courses/logic/lectures/lecture08.html.

[4] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, Netherlands, revised edition, 1984.

[5] G. M. Bierman. A computational interpretation of the $\lambda\mu$-calculus. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Proceedings 23rd Int. Symp. on Math. Found. of Comp. Science, MFCS'98, Brno, Czech Rep., 24–28 Aug. 1998*, volume 1450, pages 336–345. Springer-Verlag, Berlin, 1998.

[6] N. G. de Bruijn. The mathematical language automath, its usage and some of its extensions. In *Symposium on automatic demonstration (IRIA, Versailles 1968), Lecture Notes in Mathematics 125*, pages 29–61. Springer, 1970.

[7] T. G. Griffin. The formulae-as-types notion of control. In *Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan 1990*, pages 47–58. ACM Press, New York, 1990. Available from: http://citeseer.ist.psu.edu/griffin90formulaeastypes.html.

[8] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, UK, 1980.

[9] J.-L. Krivine. Classical logic, storage operators and second-order lambda-calculus. In *Ann. Pure App. Logic*, volume 68, pages 63–78. Elsevier, 1994.

[10] D. Madore. A page about call/cc [online]. Available from: http://www.madore.org/∼david/computers/callcc.html.

[11] J. Moschovakis. Intuitionistic logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Spring 2007. Available from: http://plato.stanford.edu/archives/spr2007/entries/logic-intuitionistic/.

[12] C.-H. L. Ong and C. A. Stewart. A Curry-Howard foundation for functional computation with control. In *Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan. 1997*, pages 215–227. ACM Press, New York, 1997. Available from: http://citeseer.ist.psu.edu/ong97curryhoward.html.

[13] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR '92: Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 190–201, London, UK, 1992. Springer-Verlag.

[14] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, Netherlands, 1969.

[15] S. van Bakel and P. Audebaud. Understanding $\mathcal{X}$ with $\lambda\mu$. INRIA No. 00097235, September 2006.

[16] S. van Bakel and P. Lescanne. Computation with classical sequents. Unpublished.

[17] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 2nd edition, 1925.

[18] Wikipedia. Continuation [online]. Available from: http://en.wikipedia.org/wiki/Continuation.