

# On the Simply-Typed Functional Machine Calculus: Categorical Semantics and Strong Normalisation

Chris Barrett

A thesis submitted for the degree of  
Doctor of Philosophy

University of Bath  
Department of Computer Science

October 2022

## **Copyright**

Attention is drawn to the fact that copyright of this thesis rests with the author and copyright of any previously published materials included may rest with third parties. A copy of this thesis has been supplied on condition that anyone who consults it understands that they must not copy it or use material from it except as licenced, permitted by law or with the consent of the author or other copyright owners, as applicable.

## **Declarations**

The material presented here for examination for the award of a higher degree by research has not been incorporated into a submission for another degree.

*Chris Barrett*

I am the author of this thesis, and the work described therein was carried out by myself personally.

*Chris Barrett*

## Abstract

The Functional Machine Calculus (FMC) was recently introduced as a generalization of the lambda-calculus to include higher-order global state, probabilistic and non-deterministic choice, and input and output, while retaining confluence. The calculus can encode both the call-by-name and call-by-value semantics of these effects. This is enabled by two independent generalizations, both natural from the perspective of the FMC's operational semantics, which is given by a simple (multi-)stack machine.

The first generalization decomposes the syntax of the lambda-calculus in a way that allows for sequential composition of terms and the encoding of reduction strategies. Specifically, there exist translations of the call-by-name and call-by-value lambda-calculus which preserve operational semantics. The second parameterizes application and abstraction in terms of 'locations' (corresponding to the multiple stacks of the machine), which gives a unification of the operational semantics, syntax, and reduction rules of the given effects with those of the lambda-calculus. The FMC further comes equipped with a simple type system which restricts and captures the behaviour of effects.

This thesis makes two main contributions, showing that **two fundamental properties of the lambda-calculus are preserved by the FMC**. The first is to show that **the categorical semantics of the FMC, modulo an appropriate equational theory, is given by the free Cartesian closed category**. The equational theory is validated by a notion of observational equivalence. The second contribution is a proof that **typed FMC-terms are strongly normalising**. This is an extension (and small simplification) of Gandy's proof for the lambda-calculus, which additionally emphasizes its latent operational intuition.

## Acknowledgements

First and absolutely foremost, I'd like to thank my supervisor, Willem Heijltjes. It is perhaps a triviality to say that without the supervisor, the thesis would not exist. But this is doubly true, because I was lucky enough to study under Willem when he had the brilliant idea that is the Functional Machine Calculus, the topic of this thesis. Much more than that, I am grateful for his seemingly *endless* patience throughout my PhD, and for his advice and guidance, which was *always* insightful and has highly coloured my thinking. Finally, for giving me the freedom to follow my interests, take tangents, and develop as an independent mathematician.

I'd also to especially thank Alessio Guglielmi, who was my second supervisor during the first year of my PhD, and with whom I co-authored my first research paper. His unusual webpage brought me to Bath, and to a computer science, rather than mathematics, department: a choice that I have never since questioned. He founded Deep Inference, which was, in fact, the first proof formalism I understood – before even the sequent calculus – and has been formative in my thinking about proof theory. Guy McCusker also deserves a special mention, with whom Willem and I co-authored two papers, for his support. I'd like also to thank my examiners, Jim Laird and Ugo Dal Lago, for their careful attention and helpful suggestions.

I'd like to additionally thank my colleagues – Ben, Andrea, Alessio (Jr.), David, Georgi, Torie, Giulio, Tom, Vincent, Hollie, Cameron, Dan, and Keji for many interesting discussions, encouragement, and for putting up with more than a couple of rambles on whatever topic in proof theory I was most excited about that week. I have very much enjoyed the company of all of the above – mentors and colleagues – over the years.

The wider community also deserves mention, and I will single out Anupam Das for his encouragement. I'd also like to thank my master's supervisor, Evgenios, who was the first person I had whiteboard discussions on mathematics with, and who helped me towards my current position in Bath.

I also have to thank my family: my mother, Helen, and sister, Emma, and my friends, in particular, Edd and Elisa, who have supported me in every way I could have asked for when times were difficult, and for always believing that I would finish, even when I doubted myself.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	The $\lambda$ -calculus and Computational Effects . . . . .	4
1.1.1	The Problem . . . . .	4
1.1.2	The Solution . . . . .	6
1.2	The $\lambda$ -calculus and the Stack Machine . . . . .	10
1.3	The Sequential $\lambda$ -Calculus . . . . .	11
1.3.1	Syntax . . . . .	12
1.3.2	The Sequential $\lambda$ -Calculus as a Stack Transformer Language . . . . .	13
1.3.3	Simple Types for the Sequential $\lambda$ -Calculus . . . . .	14
1.3.4	Translations of the CBN and CBV $\lambda$ -calculus . . . . .	16
1.4	The Functional Machine Calculus . . . . .	18
1.4.1	Syntax . . . . .	19
1.4.2	The FMC as a Multi-Stack Transformer Language . . . . .	20
1.4.3	Simple Types for the FMC . . . . .	22
1.4.4	Translations of CBN and CBV Effectful $\lambda$ -Calculi . . . . .	23
1.5	Summary and Outline of Thesis . . . . .	26
1.5.1	Summary . . . . .	27
1.5.2	Outline . . . . .	27
<b>2</b>	<b>Preliminaries: Categorical Semantics of the Lambda Calculus</b>	<b>29</b>
2.1	Adjunctions . . . . .	30
2.2	Cartesian Closed Categories . . . . .	30
2.3	Signatures and The Free Cartesian Closed Category . . . . .	33
2.4	The Simply-Typed $\lambda$ -Calculus with Patterns . . . . .	35
2.5	The Category of Simply-Typed $\lambda$ -Terms . . . . .	37
<b>3</b>	<b>The Functional Machine Calculus</b>	<b>41</b>
3.1	The FMC with Values: !FMC . . . . .	42
3.2	Simple Types for the Functional Machine Calculus . . . . .	45
3.3	Variants . . . . .	48
3.4	Machine Termination . . . . .	49

<b>4</b>	<b>The Functional Machine Category</b>	<b>52</b>
4.1	String Diagrams . . . . .	54
4.1.1	The First-Order Fragment . . . . .	54
4.1.2	The Locational Fragment . . . . .	56
4.2	The Equational Theory . . . . .	58
4.3	The Functional Machine Category is Cartesian Closed . . . . .	63
4.4	Machine Equivalence . . . . .	66
<b>5</b>	<b>Categorical Semantics of the Functional Machine Calculus</b>	<b>73</b>
5.1	The Sequential $\lambda$ -Calculus is Equivalent to the STLC . . . . .	74
5.1.1	The Free Functor: STLC to !SLC . . . . .	74
5.1.2	The Interpretation: !SLC to STLC . . . . .	76
5.1.3	Soundness of the Interpretation . . . . .	78
5.1.4	Equivalence . . . . .	81
5.2	The !FMC is Equivalent to the !SLC . . . . .	86
5.2.1	Embedding: !SLC to !FMC . . . . .	87
5.2.2	Collapsing the Memory: !FMC to !SLC . . . . .	90
5.2.3	Soundness of the Collapse of the Memory . . . . .	92
5.2.4	Equivalence . . . . .	98
<b>6</b>	<b>Strong Normalisation</b>	<b>107</b>
6.1	A Quantitative Interpretation . . . . .	110
6.2	The Substitution Lemma . . . . .	115
6.3	Beta Reduction is Monotonic . . . . .	119
6.4	The Measure for Strong Normalisation . . . . .	122
6.5	Permutation Reduction . . . . .	124
6.6	The Weak Interpretation . . . . .	125
<b>7</b>	<b>Related Literature</b>	<b>128</b>
7.1	Effect-Passing Style . . . . .	128
7.2	Monads and Moggi's Computational Metalanguage . . . . .	130
7.3	Pre-monoidal and Freyd Categories . . . . .	132
7.4	Call-by-Push-Value . . . . .	133
7.5	Linear Logic and the Bang Calculus . . . . .	135
7.6	Universal Algebra and Algebraic Effects . . . . .	137
7.7	Concatenative Programming and $\kappa$ -calculus . . . . .	144
<b>8</b>	<b>Conclusion</b>	<b>145</b>
8.1	Further Research . . . . .	145
8.2	Summary . . . . .	147
	<b>Bibliography</b>	<b>148</b>

# List of Figures

1.1	String diagrams for effectful FMC-terms . . . . .	7
1.2	Typing rules for the Sequential $\lambda$ -Calculus . . . . .	15
1.3	Admissible rules for the Sequential $\lambda$ -calculus . . . . .	15
1.4	String diagrams for Sequential $\lambda$ -calculus . . . . .	17
1.5	Typing rules for the Functional Machine Calculus . . . . .	23
2.1	Typing rules for the $\lambda$ -calculus with pattern matching . . . . .	36
2.2	Admissible rules for the the $\lambda$ -calculus with pattern matching . . . .	37
3.1	Typing rules for the Functional Machine Calculus with Values . . . .	47
3.2	Admissible rules for the Functional Machine Calculus with Values . .	47
5.1	Translations between the Sequential $\lambda$ -calculus and the $\lambda$ -calculus . .	82
7.1	Typing rules for other approaches to sequencing computation . . . .	129

# Chapter 1

## Introduction

Since the 60's, the  $\lambda$ -calculus has been regarded as the definitive model of higher-order functional programming [6, 18, 62, 63]. It has a remarkably compact syntax, a natural denotational semantics, and satisfies confluence. That is, the order in which one chooses to evaluate sub-expressions makes no difference to the eventual result of the computation. This means that each expression has one canonical ‘meaning’, namely its result. Further, it comes equipped with a powerful type system that guarantees strong normalisation.

### 1.1 The $\lambda$ -calculus and Computational Effects

Serious programming languages seem to demand *effectful* primitives be added to the  $\lambda$ -calculus: for example, to deal with input and output, mutable store, probabilistic choice, exceptions, continuations, or anything that isn't *pure* computation. However, confluence is lost when one tries to extend the  $\lambda$ -calculus naively. In an upcoming example, we will see how loss of confluence can mean for the programmer a confusing loss of the soundness of equational reasoning (one cannot always soundly substitute a function for its definition), and for the theoretician a concerning challenge to its canonical denotational semantics (i.e., one independent of order of evaluation).

Thus, there is an important research program to be undertaken: how can we extend the  $\lambda$ -calculus to incorporate computational effects while maintaining the good properties listed above, which made it a centre of programming language research in the first place?

#### 1.1.1 The Problem

Let us first explore the problem with the  $\lambda$ -calculus and effects. Take, as an example, a standard  $\lambda$ -calculus incorporating *probabilistic choice* and *state*.

$$M ::= x \mid MN \mid \lambda x.M \mid M \oplus N \mid c := N; M \mid !c$$



The syntax of the  $\lambda$ -calculus is extended with new primitives: a term  $M \oplus N$ , which tosses a coin and evaluates as  $M$  or  $N$  based on the result; a term  $c := N; M$ , which assigns the value  $N$  to the memory cell  $c$ , before continuing as  $M$ , and a term  $!c$  which dereferences the cell  $c$ . We give two example  $\lambda$ -terms below, one for probabilistic choice, and one for state, demonstrating the failure of confluence. That is, each term gives different results depending on whether it is reduced by a call-by-name (CBN) or call-by-value (CBV) strategy [87]. In the first term,  $=$  is an equality-checking function, and  $\top$  and  $\perp$  are Booleans.

$$\begin{array}{ccc} \top \oplus \perp & \xrightarrow{\text{cbn}} & (\lambda x. x = x)(\top \oplus \perp) \quad \xrightarrow{\text{cbv}} \quad \top \\ 5 & \xrightarrow{\text{cbn}} & c := 3; (\lambda x. !c)(c := 5; M) \quad \xrightarrow{\text{cbv}} \quad 3 \end{array}$$

According to CBN and CBV, respectively, the case of probabilistic choice either *duplicates first* its argument, and *then tosses two distinct coins*, one for each resulting choice to be made, vs. *tossing a coin first* to decide the result of its argument, and *then duplicating* that result. For the case of state, the sub-term  $c := 5; M$  is either *discarded before it has chance to run*, thus making no change to state, vs. *being run before it is discarded*, thus updating the state.

The common approach of insisting on a fixed order of evaluation only vacuously preserves confluence. Nevertheless, one would be forgiven for thinking that *another*  $\lambda$ -calculus variant is not the way forward. Indeed, there is perhaps a problem of canonicity in the field of programming languages: to quote Abramsky, “it is too easy to cook up yet another variant lambda calculus; there are too few constraints” [1].

We proceed in this vein of inquiry anyway. The remarkable work of developing a variant which is confluent in the presence of (an important subset of) effects, the *Functional Machine Calculus (FMC)*, was recently carried out by Heijltjes [9], building on [20]. However, to defend the calculus, we wish to show it preserves *all* of the good properties of the  $\lambda$ -calculus. The aforementioned work already equips the FMC with a system of simple types which restricts and captures the behaviour of effects. The main body of the thesis contributes by showing important remaining properties of the  $\lambda$ -calculus are preserved: namely, a natural denotational (in this case, categorical) semantics and the strong normalisation of typed terms. The former involves developing and justifying an appropriate equational theory on FMC-terms. Overall, the long-term aim of this program of research is to bring the powerful reasoning tools developed for the  $\lambda$ -calculus to bear on real-world, effectful programming languages.

Introduction and discussion of the Functional Machine Calculus is the main focus of the rest of this chapter, which then concludes with a summary of the remaining thesis. Before introducing the FMC from first principles, we begin, in the next section, with a motivating example.

### 1.1.2 The Solution

Realizing that a fully general solution to confluence for effects is out of reach, at least for the moment, will restrict our focus to a subset of computational effects, which we call *reader/writer effects*. Namely, they are input and output, global higher-order state, and probabilistic and non-deterministic choice. This is an important subset, but we will also comment, briefly, in the final chapter on how the insights offered by the Functional Machine Calculus are suggestive of further extensions, *e.g.* to local state, and even concurrency.

The *unique* insight of the Functional Machine Calculus is that the operational semantics of higher-order computation and reader/writer effects can be unified. This leads further to a unification at the syntactic and algebraic level – that is to say, both higher-order computation and effects are encoded by the same syntax, with their algebraic theory given by the same, familiar, *beta reduction*. In other words, instead of adding *new* primitives for computational effects, the FMC *decomposes* them into its existing syntax, which models higher-order computation in a standard way. This approach results in the calculus retaining confluence, as proved by Heijltjes in [9], following standard techniques [43, 112].

Before introducing a motivating example, let us take a step back. What *is* a computational effect? Despite being an extremely active area of study, there is no generally accepted formal definition. We offer the following relevant quote from Filinski.

“Informally, an effect is any deviation from the intuitive characterization of a program fragment as representing a simple function from inputs to outputs... The challenge to the semanticist is thus to admit the possibility of effects, while retaining as many as possible of the appealing properties of functional programming.”[30]

That is, a program has a side-effect if it causes some observable effect other than to consume and return its *specified* inputs and output (author’s emphasis). In other words, one common perspective on side-effects is that they can be seen as inputs or outputs of a program that *aren’t* encoded in its type.<sup>1</sup>

Consider that effectful primitives can be naively viewed as *black boxes*. For example, to *set* the value of a memory cell can be viewed as a black box which consumes the value to be written, performs the given side-effect, but has no return value. Similarly, to *get* the value of a cell can be viewed as a black box which consumes no input, but returns the value held in state. In these cases, where the value is sent or where the value comes from, respectively, is left unaccounted for. Thus the black boxes have *side-effects* which makes them sensitive to the order of operations, as in the previously given non-confluent  $\lambda$ -terms. This is because, despite there being no

---

<sup>1</sup>An example of the perspective under discussion is that of *pre-monoidal* categories, a discussion of which will follow. There do, however, exist various type systems which indicate the presence of effects.

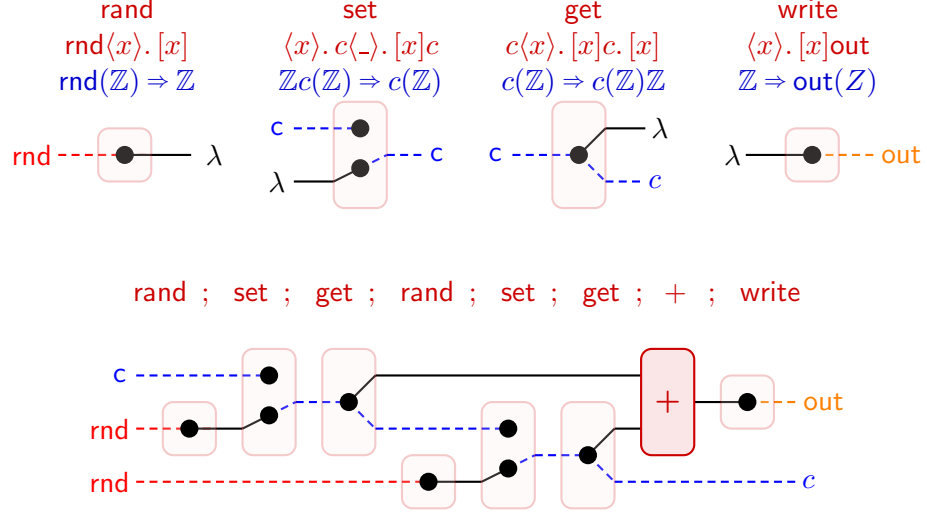


Figure 1.1: String diagrams for effectful FMC-terms

explicit dependency between the operations of *setting* and *getting*, they nevertheless are dependent on each other.

From this perspective, the achievement of the FMC is to *open* these boxes, and reveal their inner workings. Effectful primitives are *decomposed* into the plain syntax of the FMC. FMC-terms can then be typed in a relatively standard way, allowing us to see previously *hidden inputs and outputs*. This perspective is amenable to a graphical representation in via string diagrams. For an overview of these, see [107], and for related literature, see [104, 105, 51, 73, 54].

For an example of this, consider Figure 1.1.<sup>2</sup> Consider some standard effectful primitives: returning a random integer, setting and dereferencing a memory cell, and writing to output. These primitives, respectively, and their corresponding FMC-terms (in dark red), and FMC-types (blue) are given at the top of the figure. We delay discussion of the FMC-terms and -types. An example of a larger effectful process given below the depiction of primitives. The rest of this section is devoted to explaining this example in detail.

A key innovation of the FMC – which is precisely what allows for the encoding of effects – is that of parameterizing the application and abstraction of the  $\lambda$ -calculus in terms of *locations*. In the example, we use black for the *main*, location  $\lambda$ , which deals with the *standard* arguments and return values of a function. Colours and dashed lines are then used to indicate the non-main locations **rnd**, **c**, and **out**, which respectively track *random inputs*, the *value held in a memory cell c*, and *output*, e.g., printing to the screen. We then allow the black dot to depict *relocation*, as

<sup>2</sup>The term for this diagram is given modulo permutations of locations and symmetries, where appropriate. This is all discussed later, and is justified by the Permutation Lemma 4.1.2 and Remark 4.1.3

described below. The primitives given in the figure can be considered as follows. Note that the previously hidden inputs and outputs are exactly those on non-main locations, that is, the dashed wires.

- The term **rand** consumes an integer from a random stream held on location **rnd**, and relocates it to the *main* location, on which it is returned;
- The term **set c** consumes an integer from the *main* location  $\lambda$ , and one from a memory cell held at location **c**. The integer consumed from **c** is *deleted*, but the integer consumed from the main location is relocated to location **c**, taking its place;
- The term **get c** consumes an input from a memory cell held at **c** and *duplicates* it, returning one copy to **c** and one copy to the main location as its standard return value. Thus, the cell can be read from again, returning same value;
- The term **write** consumes an integer from the main location, and relocates it to an output stream held on **out**.

The expliciting of dependencies (*i.e.*, dataflow) between operations tells us when we can safely reorder effectful operations, and when we cannot, achieving a *properly monoidal* category. Further, we can now “look inside” the black box, revealing – modulo relocation – more familiar operations of duplication and deletion.

For example, it is clear from this diagram that the second call to **rand** may safely be made before the first calls to **set** and **get**. This would be illustrated by ‘sliding’ the **rand** operation along the wire, as in a monoidal category. We can also see that the second **set** is *dependent* on the first **get**: the value returned to **c** by the first **get** is discarded by the second **set**. Indeed, we will see later the composition  $(;)$  of the terms **set** and **get** creates such a beta-redex, corresponding to the expected interaction of duplication and deletion in string diagrams.

Note again that in the “black box” picture described previously, the dashed wires *are not visible*, which means one cannot know when one can soundly rearrange the order of operations: one fears violating a hidden dependency. In the study of *pre-monoidal* categories [98] a strict ordering is enforced on operations, restricting completely their rearrangement. In terms of string diagrams, this can be viewed as forgetting the dashed wires (non-main locations), and instead adding an additional *control* wire [105, 51] which is used to totally order the boxes. The FMC offers another perspective, and the relationship between string diagrams and the FMC will be discussed further later in the introduction, and in detail in Chapter 4.

The syntax of the FMC generalizes and parameterizes that of the  $\lambda$ -calculus.

- First, the variable construct is decomposed into *variable-with-continuation* and *skip* constructs, allowing a simple concatenation, or *sequential composition*, operation to be defined on terms. This gives rise to translations of the CBN and CBV  $\lambda$ -calculus, each preserving operational semantics by expressing explicitly the appropriate sequencing of computation.

- Second, the application and abstraction of the  $\lambda$ -calculus are parameterized in terms of *locations*  $\{a, b, c, \dots\}$ . This allows the encoding of effects, and the unification of their operational semantics, syntax and reduction rules with those of the  $\lambda$ -calculus.

Importantly, the first point enables the expression of both the CBN and CBV semantics of effects, as well as giving rise to a notion of *sequential composition*, familiar from imperative programming.<sup>3</sup> The second point can be seen in the example terms discussed earlier: abstraction is given by  $a\langle x \rangle.M$  and application by  $[N]a.M$ , with the argument written on the left. Both innovations comes from the consideration of the operational semantics of a simple variation of a standard stack Krivine machine which has *multiple* stacks [57], and are considered in detail in the remaining chapter.

The results contained in this thesis contribute to the program of research outlined above by verifying that the FMC does indeed preserve two fundamental *good properties* of the  $\lambda$ -calculus. The first is that the category of typed FMC-terms modulo a natural notion of observational equivalence is Cartesian closed, extending the ideas of the example described above. In fact, given an appropriate equational theory, it forms the *free Cartesian closed category*. The second contribution is to show that the type system of the FMC guarantees *strong normalisation* with respect to the (analogue of the) beta reduction relation. Both results speak to the strength of the type system.

Necessarily for the result of Cartesian closure, the calculus under study here treats locations *uniformly*: each location has an associated push- and pop- action, and its associated stack is of unbounded depth. This is in contrast with the encoding of effects described above, which requires certain special properties associated to effectful locations, *e.g.*, read- or write-only streams to model input or output, or bounded depth stacks to model memory cells. Without such assumptions, we cannot claim to accurately model the given effects. Indeed, there are more terms typeable than there would be if we were aiming to properly model the constraints associated to locations in effectful computation, and these extra terms are essential to recovering a Cartesian closed semantics: they result in there being more contexts with which to test for observational (contextual) equivalence, and in fact they are needed to define the Cartesian closed equipment (e.g. the diagonal term at certain types). As such, the categorical semantics is intended as a semantics of the calculus itself, rather than of effects, aiming to show that the calculus is semantically well-behaved despite its novelties.

With the assumption of uniformity, the inclusion of non-main locations into the calculus is semantically similar to the encoding of monadic state in simply-typed  $\lambda$ -calculus, where a stateful process of type  $A \rightarrow B$  can be considered a pure function with the larger type  $A \rightarrow (S \rightarrow B \times S)$ . In the FMC, when a term accesses a second stack, this will be recorded in its type. Thus, we can recover a Cartesian closed category of terms. Together with uniformity, it is to be understood that we

---

<sup>3</sup>The notion of sequencing described here is syntactic and semantic, but does *not* refer to reduction order.

achieve Cartesian closure because our notion of observation which determines our equivalence on terms is very strong: indeed, our type system alone tracks exactly how many times a term consults a random stream, which ought to be unobservable information.<sup>4</sup> Further investigation of the calculus as a model of computational effects is left as future work, however we note that tweaking the type system to account for read-only, write-only, or bounded depth stacks is trivial.

These results are a sanity-check for the FMC. Despite its novel perspective on effectful computation, it would appear that the calculus is very well-behaved: operationally (via the machine), semantically (via its categorical semantics), and algebraically (beta reduction is confluent and strongly normalising). These results are discussed in detail in the relevant chapters. For now, we proceed to a more detailed introduction to the FMC and its encoding of effects.

## 1.2 The $\lambda$ -calculus and the Stack Machine

The Krivine Abstract Machine (KAM) [57] is a standard call-by-name stack machine which gives an especially simple operational semantics to the  $\lambda$ -calculus. That is, the machine specifies *how* to execute a  $\lambda$ -term; in particular it implements *weak head reduction*. The machine is presented informally as follows.<sup>5</sup>

A *state* of the abstract machine is a pair  $(S, M)$  consisting of a stack  $S$  of  $\lambda$ -terms, and a  $\lambda$ -term  $M$  to be executed. The *transitions*, or *steps*, of the machine are given below, where constructors of the  $\lambda$ -calculus are interpreted as instructions for the machine:

- **Application**,  $M(N)$ , as **push**  $N$  onto the stack and continue as  $M$ ;
- **Abstraction**,  $\lambda x.M$ , as **pop** the head  $N$  off the stack and continue as  $M\{N/x\}$ .

$$\frac{(\begin{array}{c} S \\ S \cdot N \end{array}, M(N))}{(\begin{array}{c} S \\ S \cdot N \end{array}, M)} \quad \frac{(\begin{array}{c} S \cdot N \\ S \end{array}, \lambda x.M)}{(\begin{array}{c} S \\ S \end{array}, M\{N/x\})}$$

A *run*, or *execution*, of the machine is a sequence of steps. A redex  $(\lambda x.M)N$  is thus executed by a *push* followed by a *pop*, and indeed the machine correctly evaluates such a term as  $M\{N/x\}$ , as shown below, left. Considering only closed terms, variables are never encountered and the machine terminates when it reaches a state  $(\epsilon, \lambda x.M')$ , outputting the weak head normal form  $\lambda x.M'$ .<sup>6</sup> A terminating run is shown below, right, where the double line indicates a sequence of steps. The symbol

---

<sup>4</sup>One would expect that a term which reads one element from the random stream and then discards it ought to be observationally equivalent to a term which does nothing at all.

<sup>5</sup>For clarity, we present the machine with substitution rather than an environment, since we are not concerned with actual implementation.

<sup>6</sup>In general, we can run the machine with open terms, in which case it may also terminate on states of the form  $(N_n \cdots N_1, x)$ , corresponding to the weak head normal form  $xN_1 \dots N_n$ . However, we need not consider open terms.

$\epsilon$  denotes the empty stack.

$$\frac{\frac{(S, (\lambda x.M)N)}{(S \cdot N, \lambda x.M)}}{(S, M\{N/x\})} \qquad \frac{(S, M)}{(\epsilon, \lambda x.M')}$$

A call-by-name  $\lambda$ -term can thus be viewed as a sequence of *push*- and *pop*-actions, ending in a variable. In order to facilitate the reading of an application as *push N and continue as M*, let us reimagine the syntax of  $\lambda$ -terms so that the argument  $N$  is written as a prefix to  $M$ .

$$\begin{aligned} M, N &::= x \mid MN \mid \lambda x.M \\ \textcolor{red}{M}, \textcolor{red}{N} &::= \textcolor{red}{x} \mid \textcolor{red}{[N]}.M \mid \textcolor{red}{\langle x \rangle}.M \end{aligned}$$

Two successive generalizations of the  $\lambda$ -calculus are presented, both based on the intuition given by the KAM. They are naturally presented using this new syntax. First, the *Sequential  $\lambda$ -calculus (SLC)*, which incorporates a mechanism for *sequential composition* into the  $\lambda$ -calculus, and then the *Functional Machine Calculus (FMC)*, which further parameterizes actions in terms of *locations*. Each is presented informally, in turn, delaying formal definitions until Chapter 3 (see there for reference).

### 1.3 The Sequential $\lambda$ -Calculus

An important issue when dealing with computational effects is that of *evaluation order*. We saw in the introduction that a *single* effectful  $\lambda$ -term can reduce to distinct normal forms, dependent on choice of reduction strategy. We do not wish to fix a reduction strategy, as this entails a loss of expressivity – instead, we wish to be able to express both the CBN and CBV semantics of effects within the same calculus, and with natural syntax. Ergo, we aim to refine the  $\lambda$ -calculus into a new language which satisfies the following.

Given a single  $\lambda$ -term, we aim to have *distinct* call-by-name (CBN) and call-by-value (CBV) translations into *distinct* terms of some new, confluent language; each preserving the operational semantics of CBN and CBV, respectively.

Indeed, we will see in the review of related literature (Chapter 7) that something common to the various approaches to effectful  $\lambda$ -calculi is that they provide some way for the programmer to express the *sequencing* of computations *explicitly*, within syntax. The notion of sequencing we refer to is semantic, and does not refer to a notion of reduction order. In the SLC, we also have a notion of explicit sequencing which we can use to define the desired translations.

The Sequential  $\lambda$ -calculus is a novel approach towards this aim which is based on the stack machine intuition given previously. But it is not unique in its capabilities

– other languages (especially, Call-by-Push-Value [66]) provide similar capabilities. What is unique is the natural combination of our sequencing mechanism with a similarly machine-inspired perspective on effectful behaviour, which is introduced subsequently with the Functional Machine Calculus. This combination gives the programmer control over whether they want a CBN or CBV semantics for effects – and for practical purposes, we need to be able to express both – as well as resulting in a calculus which is confluent *in the presence of effects*.

### 1.3.1 Syntax

The Sequential  $\lambda$ -calculus allows for the sequencing of computations by taking seriously the operational intuition given by the Krivine machine. Observe that this perspective says that  $\lambda$ -terms are sequences of instruction; push and pop actions and variables, *where variables occur exactly at the end of the sequence*, and this definition seems overly restrictive in the following way: *the set of terms is not closed under concatenation of instruction sequences* (since this would result in a variable occurring in the middle of such a sequence). However, such an operation is unproblematic from the operational perspective: if dealing with closed terms, the machine will never encounter a variable.<sup>7</sup> Further, including some notion of sequencing is well-motivated by the simple existence of imperative programming.

To allow for sequential composition, it suffices to generalize the syntax of the  $\lambda$ -calculus accordingly and, of course, provide a unit for composition. We thus achieve the *Sequential  $\lambda$ -calculus (SLC)*.

$$M, N ::= \star \mid x.M \mid [N].M \mid \langle x \rangle.M$$

the variable  $x$  has been *decomposed* into two constructs: a *variable-with-continuation*  $x.M$  and an *identity* (or *end-of-sequence*, or *skip*),  $\star$ , with the original variable construct recoverable as  $x.\star$ . We will omit the trailing  $\star$  from terms to avoid unnecessary clutter. *Sequential composition* is then given as a *defined* operation  $N ; M$  on terms, which is capture-avoiding:<sup>8</sup>

$$\begin{aligned} \star ; M &= M & x.N ; M &= x.(N ; M) \\ [P].N ; M &= [P].(N ; M) & \langle x \rangle.N ; M &= \langle x \rangle.(N ; M) \quad (x \notin \text{fv}(M)) \end{aligned}$$

where the set of *free variables* of a term  $M$ ,  $\text{fv}(M)$ , is defined as

$$\begin{aligned} \text{fv}(\star) &= \emptyset & \text{fv}(x.M) &= \text{fv}(M) \cup \{x\} \\ \text{fv}([N].M) &= \text{fv}(M) \cup \text{fv}(N) & \text{fv}(\langle x \rangle.M) &= \text{fv}(M) \setminus \{x\} \end{aligned}$$

<sup>7</sup>Every variable is bound, and thus will be substituted for another instruction sequence before it is reached. This justifies considering variables themselves as instructions.

<sup>8</sup>It would be possible to take the construct  $N ; M$  as primitive, but this would necessitate working modulo associativity of sequencing; by taking *prefizing* as primitive, we have essentially chosen to associate to the right, considering this preferable.



There is an obvious inclusion of the set of  $\lambda$ -terms into the set of SLC-terms. The following example gives some SLC-terms which are *not* in the image of this inclusion.

**Example 1.3.1.** Consider the terms below. The first term pops the top item off the stack and discards it. The second term pops the top item off the stack, but returns two copies of it to the stack. The third term makes *no* change to the stack, while the fourth swaps the top two elements of the stack. Note how the *last-in first-out* nature of the stack is reflected in the form of these two terms.

$$\langle x \rangle \quad \langle x \rangle. [x]. [x] \quad \langle x \rangle. \langle y \rangle. [y]. [x] \quad \langle x \rangle. \langle y \rangle. [x]. [y] \quad \langle f \rangle. f. f$$

The final term is a higher-order function, which pops the top item off the stack and executes it twice, using the remaining stack items as input.

Beta reduction, shown below corresponds to an (operationally sound) optimization of the term for the machine<sup>9</sup>, eliminating consecutive *push* and *pop* actions.

$$[N]. \langle x \rangle. M \rightarrow_{\beta} \{N/x\}M$$

We formally define capture-avoiding substitution as follows. In particular, we have  $\{N/x\}x. M = N ; \{N/x\}M$ , but otherwise it is as expected for a  $\lambda$ -calculus.

$$\begin{array}{ll} \{N/x\}\star &= \star & \{N/x\}x. M &= N ; M \\ \{N/x\}[P]. M &= [\{N/x\}P]. \{N/x\}M & \{N/x\}\langle y \rangle. M &= \langle y \rangle. \{N/x\}M \end{array}$$

where, in the abstraction case,  $y \notin \text{fv}(N)$ . Note, substitution is now written on the left, matching the *pop* transition of the machine and the new syntax.

The beta law is applicable in any *sequential* context, an appropriate notion of context for the SLC, which we define in Chapter 3. It is a result of [9] that beta reduction remains confluent. Later, we will show that beta reduction is strongly normalising for well-typed terms. For our later semantic investigations, we will also give an appropriate equational theory on terms which includes that generated by the beta reduction relation, but also validates other equations such as  $\star =_{\text{id}} \langle x \rangle. [x]$ , which states that popping a term and pushing it back is the same as doing nothing.

### 1.3.2 The Sequential $\lambda$ -Calculus as a Stack Transformer Language

This generalized calculus facilitates a change in perspective from the  $\lambda$ -calculus. Previously, we saw that running a  $\lambda$ -term on the Krivine machine consumes its stack as input and terminates on the state  $(\epsilon, \lambda x. M)$ , where the remaining term is considered the output of the machine. In the SLC, however, this is considered a *failure* state: execution halts because not enough inputs have been provided.

---

<sup>9</sup>Note, we get the typical time/space tradeoff: a reduction leads to a larger term which evaluates more quickly on the machine.

Instead, like in imperative languages, we can expect a successful computation to terminate in a *skip* command, and outputs are given as the remaining stack items.

$$\frac{(S, M)}{(T, \star)}$$

Thus, the SLC is a calculus of stack *transformers*, whereas the  $\lambda$ -calculus is of one of stack *consumers*. Note that a certain input/output symmetry is thus recovered. Then  $N;M$  gives composition of runs, with the output stack of  $N$  becoming the input stack of  $M$ , as follows, and  $\star$  gives the identity run (of zero steps).

$$\text{if } \frac{(R, M)}{(S, \star)} \text{ and } \frac{(S, N)}{(T, \star)} \text{ then } \frac{\frac{(R, M;N)}{(S, N)}}{(T, \star)}$$

Following this observation, we will define in Chapter 4 a natural category of *closed* (typed) SLC-terms, with composition given by sequencing, in contrast with the  $\lambda$ -calculus, whose *open* terms form a category with composition given by *substitution*. As mentioned, closed terms are considered so that a machine run never encounters a variable.

Note that, while we can hand a term too few inputs to successfully complete a run, we cannot hand a term *too many* inputs – see below. We make later use of the *expansion* of a stack, as this is called.

$$\text{if } \frac{(S, M)}{(T, \star)} \text{ then } \frac{(RS, M)}{(RT, \star)}$$

We emphasize again that beta reduction in the SLC corresponds not to *evaluation* of a term, which is performed by the machine with respect to an input stack, but to *compilation*, *i.e.*, optimization of a term for the machine (with the usual time/space trade off - a larger term which runs more quickly, that is, in fewer steps).

We proceed to discuss how the change in perspective from that of the  $\lambda$ -calculus to that of the SLC manifests in the type system.

### 1.3.3 Simple Types for the Sequential $\lambda$ -Calculus

One of the aims of the Functional Machine Calculus is to bring the powerful reasoning tools developed for the  $\lambda$ -calculus to bear on effectful languages. Perhaps the foremost such tool is *simple types* for the  $\lambda$ -calculus. In particular, the simple type system provides a guarantee of *strong normalisation*, and is in natural correspondence with intuitionistic logic via the Curry-Howard isomorphism. Lambek further developed its categorical correspondence with the *free Cartesian closed category*, giving it a natural denotational semantics and a characterization of its models. [58, 59, 60, 61]. We present here an overview of the type system of the SLC, to be

$$\begin{array}{c}
\frac{}{\Gamma \vdash \star : \vec{\tau} \Rightarrow \vec{\tau}} \text{id} \qquad \frac{}{x : \alpha, \Gamma \vdash x : \alpha} \text{base} \\
\\
\frac{\Gamma \vdash N : \rho \quad \Gamma \vdash M : \rho \vec{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash [N]. M : \vec{\sigma} \Rightarrow \vec{\tau}} \text{app} \qquad \frac{x : \rho, \Gamma \vdash M : \vec{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash \langle x \rangle. M : \rho \vec{\sigma} \Rightarrow \vec{\tau}} \text{abs} \\
\\
\frac{x : \vec{\rho} \Rightarrow \vec{\sigma}, \Gamma \vdash M : \vec{\sigma} \vec{\tau} \Rightarrow \vec{v}}{x : \vec{\rho} \Rightarrow \vec{\sigma}, \Gamma \vdash x. M : \vec{\rho} \vec{\tau} \Rightarrow \vec{v}} \text{var}
\end{array}$$

Figure 1.2: Typing rules for the Sequential  $\lambda$ -Calculus

$$\begin{array}{c}
\frac{\Gamma \vdash M : \vec{\rho} \Rightarrow \vec{\sigma} \quad \Gamma \vdash N : \vec{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash M ; N : \vec{\rho} \Rightarrow \vec{\tau}} \text{seq} \qquad \frac{\Gamma \vdash M : \vec{\sigma} \Rightarrow \vec{v}}{\Gamma \vdash M : \vec{\sigma} \vec{\tau} \Rightarrow \vec{\tau} \vec{v}} \text{exp} \\
\\
\frac{\Gamma \vdash N : \rho \quad x : \rho, \Gamma \vdash M : \vec{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash \{N/x\}M : \vec{\sigma} \Rightarrow \vec{\tau}} \text{cut}
\end{array}$$

Figure 1.3: Admissible rules for the Sequential  $\lambda$ -calculus

extended appropriately for the FMC, while the body of this thesis concentrates on proving analogues of the above results.

Consider now the perspective of SLC-terms as stack transformers. The type of a term describes its net effect on the stack. Stacks are given a *type vector*  $\vec{\tau}$ : a type for each element of the stack. Terms  $M$  in a context  $\Gamma$  are then given type  $\tau$ , which is either a base type  $\alpha$  or an implication between type vectors.

$$\Gamma \vdash M : \tau \quad \tau ::= \alpha \mid \vec{\sigma}_A \Rightarrow \vec{\tau}_A \quad \vec{\tau} ::= \tau_1 \dots \tau_n$$

On the left of the implication is recorded the type of each element that may be consumed from the stack, and on the right the type of each element left on the stack once the run is successfully completed. We *reverse* a type vector on the left of an implication, indicated by a reversed arrow  $\vec{\tau}$ , reflecting the first-in last-out nature of the stack. Concatenation of vectors is denoted by juxtaposition. The typing rules for the SLC are shown in Figure 1.2, with some admissible rules shown in Figure 1.3.

**Example 1.3.2.** We described in Example 1.3.1 how the terms given below transform the stack. Here, we also give corresponding types. Note how, in the final two terms, the last-in first-out nature of the stack is reflected in the types as well as the terms, so that the syntax of types and terms matches.

$$\begin{array}{ll}
\langle x \rangle : \tau \Rightarrow & \langle x \rangle. [x]. [x] : \tau \Rightarrow \tau \tau \\
\langle x \rangle. \langle y \rangle. [y]. [x] : \sigma \tau \Rightarrow \tau \sigma & \langle x \rangle. \langle y \rangle. [x]. [y] : \sigma \tau \Rightarrow \sigma \tau
\end{array}$$

For a higher-order example, consider the term  $\langle f \rangle.f$ . It consumes a term from the stack, which then transforms the remaining stack in some way according to the type of that term. Thus, we can type it as below, left. The term below, right, is similar.

$$\langle f \rangle.f : (\vec{\sigma} \Rightarrow \vec{\tau}) \vec{\sigma} \Rightarrow \vec{\tau} \quad \langle f \rangle.f.f : (\vec{\tau} \Rightarrow \vec{\tau}) \vec{\tau} \Rightarrow \vec{\tau}$$

The typing derivation for a term will in fact serve as a direct proof of termination of the machine, as is proved in Chapter 3, following the proof due to Heijltjes in [9]. This gives a new perspective on types, and in particular an *operational intuition* as to their meaning. The result is that, for any stack  $S : \vec{\sigma}$  and any closed term  $M : \vec{\sigma} \Rightarrow \vec{\tau}$ , there exists a stack  $T : \vec{\tau}$  and a successful run of the machine such that

$$\frac{(S, N)}{(T, \star)}.$$

Figure 1.4 describes how to represent first-order terms (that is, where the stack only holds base types) of the typed SLC in string diagrams. The notation  $\vec{x}$  indicates a vector of variables, and  $\vec{x}$  its reverse, and we accordingly let

$$\langle \vec{x} \rangle.M = \langle x_n \rangle \dots \langle x_1 \rangle.M \quad \text{and} \quad [\vec{x}].M = [x_1] \dots [x_n].M$$

denote a sequence of abstractions and applications on  $\vec{x} = x_1 \dots x_n$ . The wires represent the input and output stacks, with the head of the stack at the top of the diagram. As is typical, where  $M$  is the identity  $\star$ , we simply draw the wires. The term  $M : \vec{\rho}\vec{\tau} \Rightarrow \vec{\tau}\vec{\sigma}$  is the term  $M : \vec{\rho} \Rightarrow \vec{\sigma}$  given a stack *expanded* by an extra  $\vec{\tau}$  elements. The term  $\langle \vec{x} \rangle.M.[\vec{x}] : \vec{\tau}\vec{\rho} \Rightarrow \vec{\sigma}\vec{\tau}$  pops the arguments for  $\vec{\tau}$  from the stack as the variables  $\vec{x}$ , evaluates  $M$  on the remaining stack of type  $\vec{\rho}$ , resulting in an output stack of type  $\vec{\sigma}$  and then returns the values bound to  $\vec{x}$  to the stack. Some terms from Example 1.3.2 are also represented in Figure 1.4. Note that we need a stronger equational theory than is generated by beta reduction alone to get all the expected equivalences between diagrams, and this is developed in Chapter 4.

An important thing to note about the type system is that no new type constructors are added. In fact, the equational theory developed in Chapter 4 makes the category of typed SLC-terms (and FMC-terms) into the free Cartesian closed category, giving a new computational interpretation of intuitionistic logic, in the style of the Curry-Howard-Lambek correspondence.<sup>10</sup> These results can suggest the canonicity of the SLC, if not the FMC itself.

### 1.3.4 Translations of the CBN and CBV $\lambda$ -calculus

As promised, the CBN and CBV translations of the  $\lambda$ -calculus into the SLC,  $(-)_n$  and  $(-)_v$ , respectively, are given below. The translation  $(-)_n$  is given by the obvious

---

<sup>10</sup>Although applying the full equational theory to the type derivations given here, which are in natural deduction style, may not seem a convincing Curry-Howard correspondence, considering intuitionistic proofs in *deep inference* systems [37, 14, 113, 38] makes for a more convincing analogy.

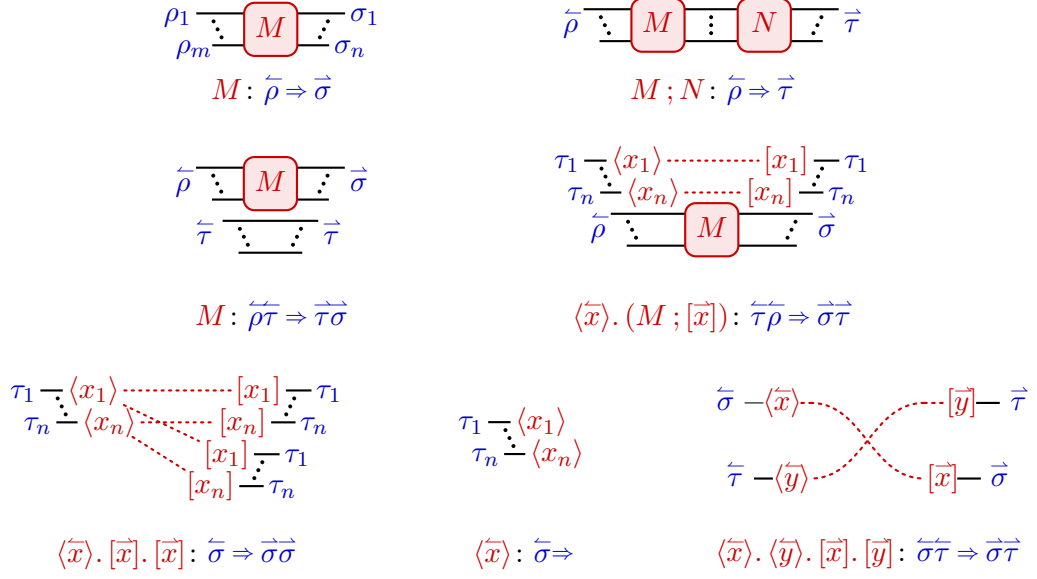


Figure 1.4: String diagrams for Sequential  $\lambda$ -calculus

inclusion of terms. The image of  $(-)_v$ , however, is different. The CBV translation of a  $\lambda$ -term returns its resulting value (a variable or abstraction) to the stack. In the application case, the argument is executed first, storing the resulting value on the stack, before executing the function. The postfix  $\langle f \rangle.f$  takes the value resulting from execution of the function and executes it with the value returned from the argument as input.

$$\begin{array}{lll}
 x_n \triangleq x & (MN)_n \triangleq [N_n].M_n & (\lambda x.M)_n \triangleq \langle x \rangle.M_n \\
 x_v \triangleq [x] & (MN)_v \triangleq N_v ; M_v ; \langle f \rangle.f & (\lambda x.M)_v \triangleq [\langle x \rangle.M_v]
 \end{array}$$

The actions of the translations on types are given as follows: if  $\Gamma \vdash M : A$  then  $\Gamma_n \vdash M_n : A_n$  and  $\Gamma_v \vdash M_v : \Rightarrow A_v$ , where

$$\begin{array}{ll}
 \alpha_n \triangleq \Rightarrow \alpha & (A_1 \rightarrow \dots \rightarrow A_k \rightarrow \alpha)_n \triangleq (A_1)_n \dots (A_k)_n \Rightarrow \alpha \\
 \alpha_v \triangleq \alpha & (A \rightarrow B)_v \triangleq A_v \Rightarrow B_v
 \end{array}$$

and  $\Gamma_n$  and  $\Gamma_v$  are given by the elementwise application of the respective translations.

Note  $(-)_n$  preserves types up to currying. In this translation, the variable which necessarily ends the term must thus consume the remaining stack and return a single element of base type  $\alpha$ . Evaluation of the CBV translation of  $\lambda$ -term returns a value to the stack, so the overall type of its translation is  $\Rightarrow A_v$ , and types are preserved up to this final action (which is again an isomorphism).

Running the translation of terms on the machine, we can see how they reflect the sequencing of CBN and CBV computation. In the former case, this is obvious, because we are working with a CBN machine. For the CBV translation, observe

how running  $((\lambda x.M)V)_v$  on the machine, where  $V$  is a value (*i.e.*, a variable or abstraction) evaluates the argument  $V_v$  *before* evaluating the function  $(\lambda x.M)_v$ . We show such a run below, where  $V_v$  translates as  $[V]$  for some  $V$ . The empty stack is given by  $\epsilon$ .

$$\begin{array}{c}
 ( \epsilon, [V]; [\langle x \rangle.M_v]; \langle f \rangle.f ) \\
 \hline
 ( V, [\langle x \rangle.M_v]; \langle f \rangle.f ) \\
 \hline
 ( V \cdot \langle x \rangle.M_v, \langle f \rangle.f ) \\
 \hline
 ( V, \langle x \rangle.M_v ) \\
 \hline
 ( \epsilon, \{V/x\}M_v )
 \end{array}$$

This is an example of how the CBN and CBV translations both preserve their respective operational semantics.<sup>11</sup> Thus we can encode both behaviours of the  $\lambda$ -calculus in the SLC, *even while working with the fixed reduction order determined by the machine*. The programmer can specify the operational behaviour they want by the choice of translation they use. Compared to the simply-typed  $\lambda$ -calculus, we have gained the ability to express within syntax *how* computation take place. Again, this does *not* mean that we *need* to fix any particular reduction order for the SLC considered as a calculus: beta reduction remains confluent and applicable in any context, so we are free to reduce redexes in any order we choose, with the same resulting semantics.<sup>12</sup>

Nevertheless, it remains the case that adding effects like probabilistic choice or state to this calculus in the naive manner shown in the opening section will still break confluence. However, we have gained the ability to express both the CBN and CBV semantics of such an operation *if we were* to fix the reduction order. We proceed to explain how, in the Functional Machine Calculus, we can deal with this remaining issue.

## 1.4 The Functional Machine Calculus

We began with the operational intuition given to higher-order computation (embodied by the application and abstraction of the  $\lambda$ -calculus) by the Krivine machine. The key realization of the Functional Machine Calculus is the following:

Just as higher-order computation be given an operational semantics in terms of push and pop actions on the Krivine machine, *so can reader/writer effects*. A machine with *multiple stacks* thus facilitates the unification of higher-order computation with these effects (and the effects with each other).

---

<sup>11</sup>More formally, a machine based operational semantics of the CBV  $\lambda$ -calculus is given by the SECD machine [87].

<sup>12</sup>In the untyped case, reduction order can still affect termination behaviour, but in the typed case, the strong normalisation result presented in Chapter 6 shows that the choice of reduction order really does not matter.

For example, consider the following effects, and their operational interpretation as actions on a stacks (or, more generally, streams):

- **Input and Output:** to **read** is a *pop* from an input stream; to **write** is a *push* to an output stream;
- **Higher-Order Global State:** **set** pops the currently held term from a memory cell (modelled as a stack of depth at most one), discards it, and *pushes* a new term; **get** *pops* the currently held value, *pushes* that same value back (for possible reuse later) and continues to subsequent use of that value;
- **Probabilistic and Non-deterministic Choice:** a *pop* from a stream of probabilistically (or non-deterministically) generated Church Booleans, and their subsequent application to a pair of terms to make a **choice** between them, following [20].

We proceed to define the Functional Machine Calculus, a natural extension of the SLC to work with multiple stacks, one for higher-order computation and one for each effect (and each memory cell). Then we *decompose* the given effects into FMC-terms, so that the syntax and the equational theory of the effects is unified with that of higher-order computation given by the SLC. Beta reduction *remains confluent*, resulting in a confluent calculus which can naturally encode reader/writer effects.

#### 1.4.1 Syntax

A natural language for a multiple-stack machine is given by a simple generalization of the SLC. Assigning to each stack a *location*  $a, b, c, \dots \in A$ , we can then *parameterize* application and abstraction (push and pop-actions) in these locations, yielding the syntax of the *Functional Machine Calculus (FMC)*.

$$M, N ::= \star \mid x.M \mid [N]a.M \mid a\langle x \rangle.M$$

The multi-stack machine transitions are given below, with constructors of the FMC interpreted as instructions for the multi-stack machine. We call an  $A$ -indexed *family* of stacks a *memory*. Notation for memories is as follows. Formally, a memory  $S_A$  can be regarded as a function from the set  $A$  to the set of stacks. Where  $B \subseteq A$ , we write  $S_B$  for the restriction of  $S_A$  to  $B$ . We write  $S_a$  for  $a \in A$  to access the stack held in memory  $S_A$  at location  $a$ , and we allow identification of  $S_{\{a\}}$  with  $S_a$ . Let  $S_B; S_C$  denote the copairing of families (considered as functions) of stacks in sets  $B$  and  $C$ . In particular this means we have  $S_A = S_{A \setminus \{a\}}; S_a$ .

- **Application on  $a, a[N].M$ ,** as **push**  $N$  onto  $a$ ; continue as  $M$ ;
- **Abstraction on  $a, a\langle x \rangle.M$ ,** as **pop** the head  $N$  off  $a$ ; continue as  $\{N/x\}M$ .

$$\frac{(\ S_{A \setminus \{a\}} ; S_a \ , \ [N]a.M \ )}{(\ S_{A \setminus \{a\}} ; S_a \cdot N \ , \ M \ )} \quad \frac{(\ S_{A \setminus \{a\}} ; S_a \cdot N \ , \ a\langle x \rangle.M \ )}{(\ S_{A \setminus \{a\}} ; S_a \ , \ \{N/x\}M \ )}$$

In addition to beta reduction, we extend the calculus with a new *permutation* reduction step allowing push- and pop-actions on a location  $a$  to interact even in the case there are intervening push- or pop-actions on *other* locations. It is easy to verify this is sound with respect to the machine above.

$$\begin{aligned} [N]a.a\langle x \rangle.M &\rightarrow_\beta \{N/x\}M \\ [N]a.b\langle x \rangle.M &\rightarrow_\pi b\langle x \rangle.[N]a.M \quad (a \neq b, x \notin \text{fv}(M)) \end{aligned}$$

By convention, we will single out a particular stack as the *main stack*, on which we consider the  $\lambda$ -calculus to operate. That is, we will always consider that the *main location*  $\lambda \in A$ , and that this holds the main stack and then we consider effect operators that transfer terms between the main stack (where they may be made further use of) and other locations. We omit the location  $\lambda$  from the syntax to avoid clutter.

What is remarkable here is that reader/writer effects and the higher-order mechanisms of the  $\lambda$ -calculus can be realized using same syntactic constructs. We proceed to show precisely how this is done.

#### 1.4.2 The FMC as a Multi-Stack Transformer Language

We introduce by example some effectful terms in the FMC, and discuss later the differing CBN and CBV semantics of effects, and corresponding extensions of the previously given translations. As examples, consider the following operations for input and output, a memory cell  $c$ , and random and non-deterministic sums as defined constructs (“sugar”) into the FMC.

$$\begin{aligned} \text{write} &\triangleq \langle x \rangle.[x]\text{out} & \text{set } c &\triangleq \langle x \rangle.c(\_).[x]c & N \oplus M &\triangleq [M].[N].\text{rnd}\langle x \rangle.x \\ \text{read} &\triangleq \text{in}\langle x \rangle.[x] & \text{get } c &\triangleq c\langle x \rangle.[x]c.[x] & N + M &\triangleq [M].[N].\text{nd}\langle x \rangle.x \end{aligned}$$

where  $(\_)$  represents a variable that does not occur elsewhere. These are explained in the following examples.

**Input and Output:** These make use of dedicated *input* and *output* locations  $\text{in}, \text{out} \in A$ , and the *pop* and *push* transitions, respectively, give the expected operational semantics. For input, the machine is initialized with a stream  $S_{\text{in}} = \cdots N_3 \cdot N_2 \cdot N_1$  (infinite to the left) at  $\text{in}$ . For output, evaluation generates a stream  $N_1, N_2, \dots$  (finite at any step) at  $\text{out}$ .

$$\begin{aligned} &\frac{(\ S_{A \setminus \{\lambda, \text{in}\}} \ ; \ S_\lambda \ ; \ S_{\text{in}} \cdot N \ ; \ \text{in}\langle x \rangle.[x] \ )}{(\ S_{A \setminus \{\lambda, \text{in}\}} \ ; \ S_\lambda \ ; \ S_{\text{in}} \ ; \ [N] \ )} \\ &\frac{(\ S_{A \setminus \{\lambda, \text{in}\}} \ ; \ S_\lambda \cdot N \ ; \ S_{\text{in}} \ ; \ \star \ )}{(\ S_{A \setminus \{\lambda, \text{in}\}} \ ; \ S_\lambda \ ; \ S_{\text{in}} \ ; \ \star \ )} \\ &\frac{(\ S_{A \setminus \{\lambda, \text{out}\}} \ ; \ S_\lambda \cdot N \ ; \ S_{\text{out}} \ ; \ \langle x \rangle.[x]\text{out} \ )}{(\ S_{A \setminus \{\lambda, \text{out}\}} \ ; \ S_\lambda \ ; \ S_{\text{out}} \ ; \ [N]\text{out} \ )} \\ &\frac{(\ S_{A \setminus \{\lambda, \text{out}\}} \ ; \ S_\lambda \ ; \ S_{\text{out}} \cdot N \ ; \ \star \ )}{(\ S_{A \setminus \{\lambda, \text{out}\}} \ ; \ S_\lambda \ ; \ S_{\text{out}} \cdot N \ ; \ \star \ )} \end{aligned}$$



**Higher-Order Global State:** A memory cell is modelled by a location  $c \in A$ . The associated stack is expected to hold at most one value, which is preserved by the encoding of the operators, and not enforced externally. In the machine, the stack for each cell is initialized with a (dummy) value, and the transitions then give the expected operational semantics. Here,  $\epsilon_c$  denotes the empty stack at location  $c$ .

$$\begin{array}{c}
\frac{(\ S_{A \setminus \{\lambda, c\}} ; S_\lambda \quad ; \epsilon_c \cdot N \ ; \ c\langle x \rangle. [x]c. [x] \ )}{(\ S_{A \setminus \{\lambda, c\}} ; S_\lambda \quad ; \epsilon_c \quad ; \quad [N]c. [N] \ )} \\
\frac{(\ S_{A \setminus \{\lambda, c\}} ; S_\lambda \quad ; \epsilon_c \cdot N \ ; \quad [N] \ )}{(\ S_{A \setminus \{\lambda, c\}} ; S_\lambda \cdot N \ ; \ \epsilon_c \cdot N \ ; \quad \star \ )} \\
\\
\frac{(\ S_{A \setminus \{\lambda, c\}} ; S_\lambda \cdot M \ ; \ \epsilon_c \cdot N \ ; \ \langle x \rangle. c\langle \_ \rangle. [x]c \ )}{(\ S_{A \setminus \{\lambda, c\}} ; S_\lambda \quad ; \epsilon_c \cdot N \ ; \quad c\langle \_ \rangle. [M]c \ )} \\
\frac{(\ S_{A \setminus \{\lambda, c\}} ; S_\lambda \quad ; \epsilon_c \quad ; \quad [M]c \ )}{(\ S_{A \setminus \{\lambda, c\}} ; S_\lambda \quad ; \epsilon_c \cdot M \ ; \quad \star \ )}
\end{array}$$

**Probabilistic and non-deterministic sums:** Following the probabilistic case [20], probabilistic and non-deterministic sums are included via dedicated locations  $\mathbf{rnd}, \mathbf{nd} \in A$ , where the machine is initialized with streams of  $\mathbf{T} \triangleq \langle x \rangle. \langle y \rangle. x$  and  $\mathbf{F} \triangleq \langle x \rangle. \langle y \rangle. y$ , generated probabilistically for  $\mathbf{rnd}$  and non-deterministically for  $\mathbf{nd}$ .<sup>13</sup> The probabilistic case is illustrated below, where, because the input stream  $\mathbf{rnd}$  is probabilistically generated, the result is the probabilistic mixture of the two machine runs.

$$\begin{array}{c}
\frac{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \quad ; S_{\mathbf{rnd}} \cdot \mathbf{T} \ ; \ [M]. [N]. \mathbf{rnd}\langle x \rangle. x \ )}{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \cdot M \quad ; S_{\mathbf{rnd}} \cdot \mathbf{T} \ ; \quad [N]. \mathbf{rnd}\langle x \rangle. x \ )} \\
\frac{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \cdot M \cdot N \ ; \ S_{\mathbf{rnd}} \cdot \mathbf{T} \ ; \quad \mathbf{rnd}\langle x \rangle. x \ )}{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \cdot M \cdot N \ ; \ S_{\mathbf{rnd}} \quad ; \quad \mathbf{T} \ )} \\
\frac{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \quad ; S_{\mathbf{rnd}} \quad ; \quad \mathbf{T} \ )}{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \quad ; S_{\mathbf{rnd}} \quad ; \quad N \ )} \\
\\
\frac{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \quad ; S_{\mathbf{rnd}} \cdot \mathbf{F} \ ; \ [M]. [N]. \mathbf{rnd}\langle x \rangle. x \ )}{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \cdot M \quad ; S_{\mathbf{rnd}} \cdot \mathbf{F} \ ; \quad [N]. \mathbf{rnd}\langle x \rangle. x \ )} \\
\frac{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \cdot M \cdot N \ ; \ S_{\mathbf{rnd}} \cdot \mathbf{F} \ ; \quad \mathbf{rnd}\langle x \rangle. x \ )}{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \cdot M \cdot N \ ; \ S_{\mathbf{rnd}} \quad ; \quad \mathbf{F} \ )} \\
\frac{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \quad ; S_{\mathbf{rnd}} \quad ; \quad \mathbf{F} \ )}{(\ S_{A \setminus \{\lambda, \mathbf{rnd}\}} ; S_\lambda \quad ; S_{\mathbf{rnd}} \quad ; \quad M \ )}
\end{array}$$

Note that choice effects are dealt with by externalizing them: the calculus remains deterministic, but is evaluated in the context of a random (or non-deterministic) oracle.

To see how beta reduction and permutation reduction allow for optimization of the above terms for the machine, consider the following example.

<sup>13</sup>We could equally generate a stream of Booleans and use a conditional statement to make the choice.

**Example 1.4.1.** To see how **set** and **get** interact through beta reduction, consider the terms below. The first term can be verified to give the same result as  $[5].\text{set } c$ , as expected. Similarly, the second term can be verified to be the same result as  $[4].\text{set } c.[4]$ , as expected.

$$\begin{aligned}
[3].\text{set } c; [5].\text{set } c &= [3].\langle x \rangle. c\langle - \rangle. [x]c. [5].\langle y \rangle. c\langle - \rangle. [y] \\
&\rightarrow_{\beta} c\langle - \rangle. [3]c. [5].\langle y \rangle. c\langle - \rangle. [y] \\
&\rightarrow_{\beta} c\langle - \rangle. [3]c; c\langle - \rangle. [5] \\
&\rightarrow_{\beta} c\langle - \rangle. [5] \\
[4].\text{set } c; \text{get } c &= [4].\langle x \rangle. c\langle - \rangle. [x]c. c\langle y \rangle. [y]c. [y] \\
&\rightarrow_{\beta} c\langle - \rangle. [4]c. c\langle y \rangle. [y]c. [y] \\
&\rightarrow_{\beta} c\langle - \rangle. [4]c. [4],
\end{aligned}$$

Observe further how permutation can unlock new redexes, for example in the case of the term below with two memory cells  $c$  and  $d$ , it allows interaction of **set**  $c$  and **get**  $c$  despite the intervening **set**  $d$ .

$$\begin{aligned}
\text{set } c; \text{set } d; \text{get } c &= \langle x \rangle. c\langle - \rangle. [x]c. \langle y \rangle. d\langle - \rangle. [y]d. c\langle z \rangle. [z]c. [z] \\
&\rightarrow_{\pi} \langle x \rangle. c\langle - \rangle. \langle y \rangle. [x]c. d\langle - \rangle. [y]d. c\langle z \rangle. [z]c. [z] \\
&\rightarrow_{\pi} \langle x \rangle. c\langle - \rangle. \langle y \rangle. d\langle - \rangle. [x]c. [y]d. c\langle z \rangle. [z]c. [z] \\
&\rightarrow_{\pi} \langle x \rangle. c\langle - \rangle. \langle y \rangle. d\langle - \rangle. [x]c. c\langle z \rangle. [y]d. [z]c. [z] \\
&\rightarrow_{\pi} \langle x \rangle. c\langle - \rangle. \langle y \rangle. d\langle - \rangle. [x]c. c\langle z \rangle. [y]d. [z]c. [z] \\
&\rightarrow_{\beta} \langle x \rangle. c\langle - \rangle. \langle y \rangle. d\langle - \rangle. [y]d. [x]c. [x]
\end{aligned}$$

### 1.4.3 Simple Types for the FMC

The type system for the FMC extends that of the SLC with *locations*. A *memory* type  $\vec{\tau}_A$  is given by an  $A$ -indexed family of type vectors, and computations track the input and output *memories*, rather than the input and output *stacks*.

$$\tau ::= \alpha \mid \vec{\sigma}_A \Rightarrow \vec{\tau}_A \quad \vec{\tau} ::= \tau_1 \dots \tau_n \quad \vec{\tau}_A \triangleq \{\vec{\tau}_a \mid a \in A\}$$

This means that the type system tracks, in addition to all function input and output, all the changes to state, all consumption from oracles, *etc.* Therefore, from the denotational perspective, we can consider terms as a pure function between *memories*, whereas by *forgetting* all information about non-main stacks we could recover something more akin to the typical notion of a process with *side-effects*.

Typing rules are given in Figure 1.5, using the following notation. A *singleton* memory type  $a(\vec{\tau})$  is empty at every location except  $a$ . We omit the main location  $\lambda$  for singleton types, so  $\lambda(\vec{\tau})$  may be written as  $\vec{\tau}$ . *Concatenation* extends to families point-wise, so  $\vec{\sigma}_A \vec{\tau}_A = \{\vec{\sigma}_a \vec{\tau}_a \mid a \in A\}$ , and similarly for the reverse of a family

$$\begin{array}{c}
\frac{}{\Gamma \vdash \star : \vec{\tau}_A \Rightarrow \vec{\tau}_A} \text{id} \qquad \frac{}{x : \alpha, \Gamma \vdash x : \alpha} \text{base} \\
\\
\frac{\Gamma \vdash N : \rho \quad \Gamma \vdash M : a(\rho) \vec{\sigma}_A \Rightarrow \vec{\tau}_A}{\Gamma \vdash [N]a.M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A} \text{app} \qquad \frac{x : \rho, \Gamma \vdash M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A}{\Gamma \vdash a\langle x \rangle.M : a(\rho) \vec{\sigma}_A \Rightarrow \vec{\tau}_A} \text{abs} \\
\\
\frac{x : \vec{\rho}_A \Rightarrow \vec{\sigma}_A, \Gamma \vdash M : \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{v}_A}{x : \vec{\rho}_A \Rightarrow \vec{\sigma}_A, \Gamma \vdash x.M : \vec{\rho}_A \vec{\tau}_A \Rightarrow \vec{v}_A} \text{var}
\end{array}$$

Figure 1.5: Typing rules for the Functional Machine Calculus

$\vec{\tau}_A$ . An equivalent way to view computation types is as an implication between two vectors of singletons, considered modulo the permutation of types on different locations, as below.

$$a_1(\sigma_1) \dots a_n(\sigma_n) \Rightarrow b_1(\tau_1) \dots b_m(\tau_m) \qquad a(\sigma) b(\tau) \sim b(\tau) a(\sigma)$$

Note that the notation  $a(\tau)$  isn't a novel type constructor, but notation for an indexed product.

**Example 1.4.2.** We can build up the type of the term given in in Figure 1.1 as follows.

$$\begin{array}{ll}
\text{write} : & \mathbb{Z} \Rightarrow \text{out}(\mathbb{Z}) \\
+ ; \text{write} : & \mathbb{Z}\mathbb{Z} \Rightarrow \text{out}(\mathbb{Z}) \\
\text{get} ; + ; \text{write} : & \mathbb{Z} c(\mathbb{Z}) \Rightarrow c(\mathbb{Z}) \text{out}(\mathbb{Z}) \\
\text{set} ; \text{get} ; + ; \text{write} : & \mathbb{Z}\mathbb{Z} c(\mathbb{Z}) \Rightarrow c(\mathbb{Z}) \text{out}(\mathbb{Z}) \\
\text{rand} ; \text{set} ; \text{get} ; + ; \text{write} : & \mathbb{Z} \text{rnd}(\mathbb{Z}) c(\mathbb{Z}) \Rightarrow c(\mathbb{Z}) \text{out}(\mathbb{Z}) \\
\text{get} ; \text{rand} ; \text{set} ; \text{get} ; + ; \text{write} : & \text{rnd}(\mathbb{Z}) c(\mathbb{Z}) \Rightarrow c(\mathbb{Z}) \text{out}(\mathbb{Z}) \\
\text{set} ; \text{get} ; \text{rand} ; \text{set} ; \text{get} ; + ; \text{write} : & \mathbb{Z} \text{rnd}(\mathbb{Z}) c(\mathbb{Z}) \Rightarrow c(\mathbb{Z}) \text{out}(\mathbb{Z}) \\
\text{rand} ; \text{set} ; \text{get} ; \text{rand} ; \text{set} ; \text{get} ; + ; \text{write} : & \text{rnd}(\mathbb{Z}\mathbb{Z}) c(\mathbb{Z}) \Rightarrow c(\mathbb{Z}) \text{out}(\mathbb{Z})
\end{array}$$

#### 1.4.4 Translations of CBN and CBV Effectful $\lambda$ -Calculi

It is now demonstrated, by example, how both CBN and CBV semantics for effects are expressible within the FMC, by extension of the CBN and CBV translations of the  $\lambda$ -calculus. We deal here with probabilistic choice and first-order state (that is, memory cells which hold base types only).

**Probabilistic Choice:** Consider the  $\lambda$ -calculus naively extended with probabilistic choice and state.

$$M ::= x \mid MN \mid \lambda x.M \mid M \oplus N \mid c := N; M \mid !c$$

Consider also the following non-confluent  $\lambda$ -term, and its CBN and CBV reductions respectively, given below.

$$(\lambda f. \lambda x. f(f(x)))(M \oplus N) \rightarrow_{\text{cbn}} \lambda x. (M \oplus N)((M \oplus N)x)$$

$$(\lambda f. \lambda x. f(f(x)))(M \oplus N) \rightarrow_{\text{cbv}} \begin{cases} 50\% & (\lambda f. \lambda x. f(fx))M \\ 50\% & (\lambda f. \lambda x. f(fx))N \end{cases}$$

The translation of probabilistic choice follows the probabilistic  $\lambda$ -calculus from [20], which decomposes the choice operator into a *generator* of a random value and a *consumer*, which makes the choice depending on the value passed to it. Here, the role of the generator is played by an abstraction on a random stream of Booleans, and the role of consumer is played by the use of an abstracted Boolean.

Recall that probabilistic choice in the  $\lambda$ -calculus is usually typed as below.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash M \oplus N : A}$$

Recall the encoding of Booleans as  $\mathbf{T} \triangleq \langle x \rangle. \langle y \rangle. x$  and  $\mathbf{F} \triangleq \langle x \rangle. \langle y \rangle. y$  and let  $\mathbb{B} = \tau\tau \Rightarrow \tau$  be the type of Booleans. Note, we could similarly deal with *actual* Booleans of base type, in combination with a conditional for making the choice. Then the translation of the types of probabilistic terms in CBN and CBV is given below. We use a CBN-style encoding here, since even in CBV it is convention not to evaluate both branches of  $M \oplus N$  before the choice is made. It is now recorded in the type system that each computation additionally reads some  $n$  Booleans from the random stream. These are greyed out in the type system to represent that this information is *missing* in the simply-typed  $\lambda$ -calculus. As before, the overall type of a CBV term is given by  $\text{rnd}(\mathbb{B}^n) \Rightarrow A_v$ .

$$\begin{aligned} \alpha_n &\triangleq \text{rnd}(\mathbb{B}^n) \Rightarrow \alpha & (A_1 \rightarrow \dots \rightarrow A_m \rightarrow \alpha)_n &\triangleq \text{rnd}(\mathbb{B}^n)(A_1)_n \dots (A_m)_n \Rightarrow \alpha \\ \alpha_v &\triangleq \alpha & (A \rightarrow B)_v &\triangleq \text{rnd}(\mathbb{B}^n)A_v \Rightarrow B_v \end{aligned}$$

We thus translate  $M \oplus N$  in the same way for both CBN and CBV, but the resulting terms get different types. For  $M \oplus N : A_1 \rightarrow \dots \rightarrow A_m \rightarrow \alpha$  in CBN and  $M \oplus N : A \rightarrow B$  in CBV, we have the following, where  $n+1$  indicates we have read one more Boolean from the stream.

$$\begin{aligned} (M \oplus N)_n &\triangleq \text{rnd}\langle b \rangle. [M_n]. [N_n]. b : \text{rnd}(\mathbb{B}^{n+1})(A_1)_n \dots (A_m)_n \Rightarrow \alpha \\ (M \oplus N)_v &\triangleq \text{rnd}\langle b \rangle. [M_n]. [N_v]. b : \text{rnd}(\mathbb{B}^{n+1}) \Rightarrow (\text{rnd}(\mathbb{B}^n)A_v \Rightarrow B_v) \end{aligned}$$

For  $M \oplus N : (A \rightarrow \alpha) \rightarrow (A \rightarrow \alpha)$  and  $x : A \rightarrow \alpha$  in CBN and  $M \oplus N : A \rightarrow A$  and  $x : A$  in CBV, we have the following translations (up to a small amount of simplification by beta reduction). We assume  $M$  and  $N$  to be pure terms (that is, which don't read from the probabilistic stream) below, in order to simplify the types.

$$\begin{aligned} ((\lambda f. f(fx))(M \oplus N))_n &= [(M \oplus N)_n]. \langle f \rangle. [[x]. f]. f : \text{rnd}(\mathbb{B})A_n \Rightarrow \alpha \\ ((\lambda f. f(fx))(M \oplus N))_v &= (M \oplus N)_v. \langle f \rangle. [x]. f. \langle y \rangle. y. f. \langle z \rangle. z : \text{rnd}(\mathbb{B}) \Rightarrow A_v \end{aligned}$$

In the CBN case, the choice, including the *generator*  $\text{rnd}\langle x \rangle$  is duplicated, leading to a probabilistic sub-term  $[x]. (M \oplus N)_n$ , which is then consumed by a second

probabilistic sub-term  $(M \oplus N)_n$ . Note, the choice in the former sub-term is decided, then, until the latter subterm applies it to some argument (for example, consider  $M, N = \lambda x.x$ ). In the CBV case, the choice between  $M$  and  $N$  is made once, at the start, and the result is duplicated (but not the generator). It is easy to verify that these terms give the expected distribution of results when evaluated on the machine.

**First-Order State:** Recall we will work with *first-order* state here, that is where the memory cell can hold only terms of base type. That is, in  $c := N; M$ , we require  $N : \alpha$ . We further make the simplifying assumption that such terms are not themselves effectful. We will elide much discussion of types in this section, but they can be dealt with similarly to the case of probabilistic choice. Consider now the non-confluent  $\lambda$ -term from the introduction.

$$3 \text{ cbv} \leftarrow c := 3; (\lambda x. !c)(c := 5; M) \rightarrow_{\text{cbn}} 5$$

Recalling the **set**  $c$  and **get**  $c$  combinators from earlier, we translate the stateful constructors of the  $\lambda$ -calculus as follows.

$$\begin{aligned} \text{set } c &\triangleq \langle x \rangle. c\langle \_ \rangle. [x]c & \text{get } c &\triangleq c\langle x \rangle. [x]c. [x] \\ (!c)_n &\triangleq \text{get } c & (c := N; M)_n &\triangleq N_n; \text{set } c; M_n \\ (!c)_v &\triangleq \text{get } c & (c := N; M)_v &\triangleq N_v; \text{set } c; M_v \end{aligned}$$

Recalling that  $\alpha_n = \alpha_v = \Rightarrow \alpha$ , it can easily be checked that this gives the expected types (up to the actions on  $c$ , which do not appear in the  $\lambda$ -calculus), where the memory cell *termc* holds a term of type  $\alpha \text{ lth } \alpha$ . We will now verify the following laws apply.

$$c := M; c := N \rightarrow c := N \quad c := M; !c \rightarrow c := M; M$$

These terms translate as follows, where we simplify by beta reduction where possible. We elide writing  $M_n, N_n, M_v$  and  $N_v$  here, and instead write simply  $M$ .

$$\begin{aligned} c := M; c := N; P &= M. c\langle \_ \rangle. \langle x \rangle. [x]c. N. c\langle \_ \rangle. \langle y \rangle. [y]c; P \\ a := N; P &= N. c\langle \_ \rangle. \langle y \rangle. [y]c; P \\ a := M; !a &= M. a\langle \_ \rangle. \langle x \rangle. [x]a. a\langle y \rangle. [y]a. [y] \\ &\rightarrow_{\beta} M. a\langle \_ \rangle. \langle x \rangle. [x]a. [x] \\ a := M; M &= M. a\langle \_ \rangle. \langle x \rangle. [x]a. M \end{aligned}$$

To see these are equivalent, we can run them on the machine and see we achieve the same results. In the following runs, let *itermM'* be the result, which is pushed to the stack, of running  $M$ . We omit writing the full memory  $S_A$  in the runs below, to save space.

$$\begin{array}{c} (S_\lambda \quad ; \epsilon_c \cdot Q \quad ; M. c\langle \_ \rangle. \langle x \rangle. [x]c. N. c\langle \_ \rangle. \langle y \rangle. [y]c; P) \\ \hline (S_\lambda \cdot M' \quad ; \epsilon_c \cdot Q \quad ; c\langle \_ \rangle. \langle x \rangle. [x]c. N. c\langle \_ \rangle. \langle y \rangle. [y]c; P) \\ \hline (S_\lambda \cdot M' \quad ; \epsilon_c \quad ; \langle x \rangle. [x]c. N. c\langle \_ \rangle. \langle y \rangle. [y]c; P) \\ \hline (S_\lambda \quad ; \epsilon_c \quad ; [M']c. N. c\langle \_ \rangle. \langle y \rangle. [y]c; P) \\ \hline (S_\lambda \quad ; \epsilon_c \cdot M' \quad ; N. c\langle \_ \rangle. \langle y \rangle. [y]c; P) \end{array}$$

$$\begin{array}{c}
( S_\lambda \quad ; \epsilon_c \cdot Q \quad ; M.c\langle \_ \rangle . \langle x \rangle . [x]c.[x] ) \quad ( S_\lambda \quad ; \epsilon_c \cdot Q \quad ; M.a\langle \_ \rangle . \langle x \rangle . [x]a.M ) \\
( S_\lambda \cdot M' \quad ; \epsilon_c \cdot Q \quad ; c\langle \_ \rangle . \langle x \rangle . [x]c.[x] ) \quad ( S_\lambda \cdot M' \quad ; \epsilon_c \cdot Q \quad ; c\langle \_ \rangle . \langle x \rangle . [x]c.M ) \\
( S_\lambda \cdot M' \quad ; \epsilon_c \quad ; \langle x \rangle . [x]c.[x] ) \quad ( S_\lambda \cdot M' \quad ; \epsilon_c \quad ; \langle x \rangle . [x]c.M ) \\
( S_\lambda \quad ; \epsilon_c \quad ; [M']c.[M'] ) \quad ( S_\lambda \quad ; \epsilon_c \quad ; [M']c.M ) \\
( S_\lambda \quad ; \epsilon_c \cdot M' \quad ; [M'] ) \quad ( S_\lambda \quad ; \epsilon_c \cdot M' \quad ; M ) \\
( S_\lambda \cdot M' \quad ; \epsilon_c \cdot M' \quad ; \star ) \quad ( S_\lambda \cdot M' \quad ; \epsilon_c \cdot M' \quad ; \star )
\end{array}$$

Note that, in the first case, we recover the expected term, although the memory now holds  $M'$  instead of the original  $Q$ . However, since  $a := N; P$  discards the term in memory, this does not matter.

In chapter 4, we will give an equational theory which extends the given reduction rules to capture more of *machine equivalence* (also introduced formally in that chapter), thus axiomatizing when two terms are equivalent with respect to the machine.

We now return to the non-confluent  $\lambda$ -term we were aiming to translate. Again, we have a CBN and a CBV translation, given below. Recall, again, that in the CBN translation  $5 : \mathbb{Z}$  is translated as  $\mathbf{5} : \Rightarrow \mathbb{Z}$ , whereas in the CBV translation we have  $\mathbf{[5]} : \Rightarrow \mathbb{Z}$ .

$$\begin{aligned}
(a := 3; (\lambda x. !a)(a := 5; M))_n &= \mathbf{3.set\ } a. [\mathbf{5.set\ } a. M_n]. \langle x \rangle. \mathbf{get\ } a \\
&\rightarrow_\beta \mathbf{3.set\ } a. \mathbf{get\ } a \\
&= a := 3; !a \\
&= a := 3; 3 \\
(a := 3; (\lambda x. !a)(a := 5; M))_v &= [\mathbf{3.set\ } a. [\mathbf{5.set\ } a. M_v; [\langle x \rangle. \mathbf{get\ } a]. \langle f \rangle. f] \\
&\rightarrow_\beta [\mathbf{3.set\ } a. [\mathbf{5.set\ } a. M_v; \langle x \rangle. \mathbf{get\ } a] \\
&= a := 3; a := 5; M_v; \langle x \rangle. !a \\
&= a := 5; M_v; \langle x \rangle. !a
\end{aligned}$$

We can easily verify this second term is equivalent to  $a := 5; 5$  on the machine. Note that the output of  $M_v$  is deleted by the abstraction  $\langle x \rangle$ .

The unification of the mechanism for effects (and higher-order functions) into beta reduction further facilitates the combination of effects within a single calculus, with the type system dealing with each effect (and with higher-order functions) in a uniform manner. While we have seen that our multi-stack machine gives the expected operational semantics of reader/writer effects, in fact it is the case that  $\beta$ -reduction captures the algebraic laws for *e.g.*, state [9]. Note that it is trivial to extend the type system with the expected restrictions used to model effects (that is, read- or write-only stacks for input and output, and one-place stacks for memory cells).

## 1.5 Summary and Outline of Thesis

The Functional Machine Calculus has been introduced and motivated, and we emphasize the following key points, before outlining the remaining content.

### 1.5.1 Summary

- The FMC generalizes the  $\lambda$ -calculus to allow for sequential composition, by decomposing the variable construct. This allows translations of both the CBN and CBV  $\lambda$ -calculus into the FMC, each preserving their respective operational semantics.
- Just as higher-order computation be given an operational semantics in terms of push and pop actions on the Krivine machine, so can reader/writer effects. The syntax and reduction rules of higher-order computation and effects are therefore unified via a parameterization of application (push) and abstraction (pop) in terms of locations.
- Beta reduction remains confluent and applicable in any context [9], despite the presence of effects.
- The calculus furthermore comes equipped with a type system which restricts and captures the behaviour of effects via their actions on non-main locations.

### 1.5.2 Outline

We now provide an overview of the remaining chapters and the main contributions of this thesis.

- In Chapter 2, the necessary categorical preliminaries for this thesis are given, namely, the Curry-Howard-Lambek correspondence in brief.
- We proceed to give formal definitions relating to the FMC in Chapter 3, in particular introducing a variant we name “!FMC” (the Functional Machine Calculus *with values*), which distinguishes between computations (which run on the machine) and values (which live on the stack), which will later allow for the most natural translation into the  $\lambda$ -calculus. We conclude by reprising the result, due to Heijltjes [9], that type derivations provide a direct proof of machine termination for their corresponding terms.
- The first main contribution of the thesis is in Chapters 4 and 5, where it is proved that the category of *closed*, simply-typed !FMC-terms taken modulo an appropriate equational theory, with composition given by *sequencing*, is equivalent to the category of simply-typed  $\lambda$ -terms. Given the ability of the Functional Machine Calculus to encode effects, this may be a surprising result. The nuance here is that we must assume that all locations are treated uniformly: in particular, we cannot restrict some locations to be read- or write-only, and we cannot limit the number of terms that can be held in a location, as was implicit in the encoding of effects described earlier in this chapter. In other words, we present an equational theory and denotational semantics for the !FMC as a *calculus*, and this is *not* intended as a semantics of effects. Chapter 4 is structured as follows.

- The full equational theory is defined, extending the equivalence generated by beta and permutation reduction as introduced above. It is then proved that these equations are sufficient to form a Cartesian closed category.
- Then, a natural notion of observational equivalence of terms based on the machine, called *machine equivalence*, is introduced, and it is shown to validate the equational theory.

Chapter 5 then proves the main result: equivalence of the category of !FMC-terms and the free Cartesian closed category. The result is proved in two stages, using the simply-typed  $\lambda$ -calculus as a language for the latter.

- First, we prove the categorical equivalence of the  $\lambda$ -calculus and the !SLC (the Sequential  $\lambda$ -calculus with values). This consists of considering the input stack to a closed !SLC-term as the context of an open  $\lambda$ -term.
- Second, we prove the categorical equivalence of the !FMC and the !SLC. This consists of collapsing the memory of the !FMC into a single large stack.

Note, the equivalence in the second point is up-to natural isomorphism.

- The second main contribution of the thesis is in Chapter 6, where it is proved that simply-typed FMC-terms are strongly normalising with respect to beta and permutation reduction. The proof is an adaptation of Gandy’s for the  $\lambda$ -calculus [31]. Apart from the theorem itself, some adaptations made in the proof are also of note. First, it reveals a latent operational intuition, whereby the measure for strong normalisation is related to counting the number of steps remaining to be made by the machine on some given input. Second, there is a small refinement made to the structures of the proof, which arises naturally from the setting of the FMC.
- Finally, in Chapter 7 we review the related literature, and in Chapter 8, we conclude and note some of the many possible further avenues for research.

Neither of the two main contributions depend on each other, however, one point of similarity to note is between the translation from the !SLC to the  $\lambda$ -calculus in Chapter 5 and the interpretation of FMC-terms (and therefore SLC-terms) into the semantic domain given in the strong normalisation proof of Chapter 6.

There are two publications associated with this thesis, the first laying out the foundations of the FMC [9], and the second [10] containing early results from this thesis (amongst contributions by other authors).



## Chapter 2

# Preliminaries: Categorical Semantics of the Lambda Calculus

This chapter presents the definition of a free Cartesian closed category, and recalls its equivalence with the category of terms of the simply-typed  $\lambda$ -calculus. This is preparation for Chapters 4 and 5, where we investigate the denotational semantics of the FMC. In particular, in order to deal with constants, we define the free Cartesian closed category generated over a signature with products and arrows, rather than over some base category. Further, we introduce a slight variant of the  $\lambda$ -calculus (which is standard) which deals with elimination of products via *pattern matching*. This proves to be a more natural variant to work with when relating to the FMC. Additionally, this  $\lambda$ -calculus has *two* rules for introducing constants (which is not standard), for *computation* and *value* constants, which are introduced in a variant definition of its signature. This is to achieve the closest possible correspondence with the FMC with values, !FMC, which is introduced in Chapter 3.

We will assume familiarity with basic category theory: the definition of categories, functors, natural isomorphisms, equivalence of categories, commuting diagrams and terminal objects. A basic introduction to category theory can be found in [71, 7, 64]. Our notation for composition is given in diagram order:  $f;g$ , instead of  $g \circ f$ . We use  $\text{id}_{\mathbb{C}}$  to denote the identity functor on a category  $\mathbb{C}$ , and  $\simeq$  to denote when two functors are naturally isomorphic. Given a terminal object  $\top$ , we write  $!_A : A \rightarrow \top$  to be the unique such morphism. Sometimes we drop subscripts denoting the components of natural transformations when it is clear. We first recall the definition of adjoint functors here for convenience.

## 2.1 Adjunctions

We recall the definition of adjoint functor that will be useful for understanding the definition of the free Cartesian closed category.

**Definition 2.1.1** (Adjoint Functors). A functor  $F : \mathbb{C} \rightarrow \mathbb{D}$  is a *left adjoint functor* if for each object  $A \in \mathbb{D}$ , there exists an object  $G(A) \in \mathbb{C}$  and a morphism  $\epsilon_A : F(G(A)) \rightarrow A$ , such that for every object  $B \in \mathbb{C}$  and every morphism  $f : F(B) \rightarrow A$  there exists a unique morphism  $g : B \rightarrow G(A)$ , such that the following diagram commutes:

$$\begin{array}{ccc} F(B) & & \\ F(g) \downarrow & \searrow f & \\ F(G(A)) & \xrightarrow{\epsilon_A} & A \end{array} .$$

A functor is a *right adjoint functor* if it satisfies the dual condition.

**Remark 2.1.2.** It can be shown that  $G$  can be turned into a functor  $G : \mathbb{D} \rightarrow \mathbb{C}$  such that  $\epsilon_D \circ F(G(f)) = f \circ \epsilon_C$ , for all morphisms  $f : C \rightarrow D$  in  $\mathbb{C}$ . In this case  $F$  is called *left adjoint* to  $G$ . Indeed,  $F$  is left adjoint to  $G$  if and only if  $G$  is right adjoint to  $F$ .

**Remark 2.1.3.** It is the case that  $F : \mathbb{C} \rightarrow \mathbb{D}$  is left adjoint to  $G : \mathbb{D} \rightarrow \mathbb{C}$  if and only if there is a natural isomorphism

$$\alpha : \text{Hom}_{\mathbb{C}}(F(-), -) \simeq \text{Hom}_{\mathbb{D}}(-, G(-)),$$

which thus specifies a family of bijections

$$\alpha_{A,B} : \text{Hom}_{\mathbb{C}}(F(B), A) \rightarrow \text{Hom}_{\mathbb{D}}(B, G(A)),$$

for every  $A \in \mathbb{C}$  and  $B \in \mathbb{D}$ .

## 2.2 Cartesian Closed Categories

We define a Cartesian closed category, each example of which gives a model of the simply-typed  $\lambda$ -calculus.

**Definition 2.2.1.** The (*Cartesian*) *product*  $(A_1 \times A_2, \pi_1, \pi_2)$  of two objects  $A_1$  and  $A_2$  in a category consists of an object  $A_1 \times A_2$  together with two *projections*  $\pi_1 : A_1 \times A_2 \rightarrow A_1$  and  $\pi_2 : A_1 \times A_2 \rightarrow A_2$  such that for every object  $B$  and morphisms  $f_1 : B \rightarrow A_1$  and  $f_2 : B \rightarrow A_2$ , there exists a unique morphism  $\langle f_1, f_2 \rangle : B \rightarrow A_1 \times A_2$  such that the following diagram commutes:

$$\begin{array}{ccccc} & & B & & \\ & \swarrow f_1 & \downarrow \exists! \langle f_1, f_2 \rangle & \searrow f_2 & \\ A_1 & \xleftarrow{\pi_1} & A_1 \times A_2 & \xrightarrow{\pi_2} & A_2 \end{array} .$$

We will often write the product as simply  $A_1 \times A_2$ , omitting the equipment.

**Remark 2.2.2.** If the Cartesian product of two objects exists, it is unique up to unique isomorphism, and so we will speak of *the* Cartesian product.

**Proposition 2.2.3.** *The morphism  $\langle f_1, f_2 \rangle$  is unique if for every  $g : B \rightarrow A_1 \times A_2$ ,  $\langle g; \pi_1, h; \pi_2 \rangle = g$ .*

*Proof.* For  $\langle f_1, f_2 \rangle$  to be unique means that if  $f_1 = u; \pi_1$  and  $f_2 = u; \pi_2$  then  $u = \langle f_1, f_2 \rangle$ . Let  $g : A_1 \times A_2 \rightarrow B$  be such that  $g; \pi_1 = f_1$  and  $g; \pi_2 = f_2$ . Then, by hypothesis,  $g = \langle g; \pi_1, g; \pi_2 \rangle = \langle f_1, f_2 \rangle$ .  $\square$

A Cartesian category is one which contains the product of any two objects and a terminal object; a Cartesian functor preserves this structure. This amounts to the following definition.

**Definition 2.2.4.** A *Cartesian category* is a category  $\mathbb{C}$ , such that for any two objects  $A, B \in \mathbb{C}$ , there exists their product  $A \times B$ , and such that there exists a terminal object  $\top$ . We denote a Cartesian category with its equipment as  $(\mathbb{C}, \times, \top)$ , but often write simply  $\mathbb{C}$ , omitting the equipment. A *Cartesian functor*  $F : (\mathbb{C}, \times_{\mathbb{C}}, \top_{\mathbb{C}}) \rightarrow (\mathbb{D}, \times_{\mathbb{D}}, \top_{\mathbb{D}})$  between Cartesian categories is given by

- a functor  $F : \mathbb{C} \rightarrow \mathbb{D}$ , which additionally:
- *preserves products* in the sense that for every product  $(A \times_{\mathbb{C}} B, \pi_1, \pi_2)$  in  $\mathbb{C}$ ,  $(F(A \times_{\mathbb{C}} B), F(\pi_1), F(\pi_2))$  is the product of  $F(A)$  and  $F(B)$  in  $\mathbb{D}$ , and
- *preserves the terminal object* in the sense that  $F(\top_{\mathbb{C}})$  is terminal in  $\mathbb{D}$ .

We call two Cartesian categories *equivalent* when there exists a Cartesian functor between them which is an equivalence when considered as a plain functor.

**Definition 2.2.5.** The *exponential*  $(A \Rightarrow C, \epsilon)$  of two objects  $A, C \in \mathbb{C}$  consists of an object  $A \Rightarrow C$  together with an *evaluation* morphism  $\epsilon_{A,C} : A \times (A \Rightarrow C) \rightarrow C$  such that for every object  $A \in \mathbb{C}$  and morphism  $g : A \times B \rightarrow C$ , there exists a unique morphism  $\text{curry}(g) : B \rightarrow (A \Rightarrow C)$ , which we call *currying*, such that

$$\begin{array}{ccc} A \times B & & \\ \text{id}_A \times \text{curry}(g) \downarrow & \searrow g & \\ A \times (A \Rightarrow C) & \xrightarrow{\epsilon_{A,C}} & C \end{array}$$

We will often write the exponential as simply  $A \Rightarrow C$ , omitting the equipment.

**Remark 2.2.6.** If the exponential of two objects exists, it is unique up to unique isomorphism, and so we will speak of *the* exponential.

**Proposition 2.2.7.** *The morphism  $\text{curry}(g)$  is unique if for every  $h : B \rightarrow (A \Rightarrow C)$ ,*

$$h = \text{curry}((\text{id}_A \times h); \epsilon_{A,C})$$

*Proof.* For  $\text{curry}(g)$  to be unique means that if  $g = \text{id}_A \times h; \text{ev}_{B,C}$ , then  $h = \text{curry}(g)$ . Let  $g = \text{id}_A \times h; \text{ev}_{B,C}$ . Then, by hypothesis,  $h = \text{curry}((\text{id}_B \times h); \epsilon_{A,C}) = \text{curry}(g)$ .  $\square$

A Cartesian closed category is one which has all exponentials, products, and a terminal object; a Cartesian closed functor is a functor which preserves this structure. This amounts to the following definition.

**Definition 2.2.8.** A *Cartesian closed category (CCC)* is a Cartesian category  $(\mathbb{C}, \times, \top)$ , such that for every two objects  $A, C \in \mathbb{C}$ , there exists their exponential  $A \Rightarrow C$ . We denote a CCC with its equipment as  $(\mathbb{C}, \Rightarrow, \times, \top)$ , but often write simply  $\mathbb{C}$ , omitting the equipment. A *Cartesian closed functor*  $F : (\mathbb{C}, \Rightarrow_{\mathbb{C}}, \times_{\mathbb{C}}, \top_{\mathbb{C}}) \rightarrow (\mathbb{D}, \Rightarrow_{\mathbb{D}}, \times_{\mathbb{D}}, \top_{\mathbb{D}})$  is given by

- a Cartesian functor  $F : (\mathbb{C}, \times_{\mathbb{C}}, \top_{\mathbb{C}}) \rightarrow (\mathbb{D}, \times_{\mathbb{D}}, \top_{\mathbb{D}})$  between the underlying Cartesian categories, which additionally:
- *preserves exponentials* in the sense that if  $(A \Rightarrow B, \epsilon)$  is an exponential in  $\mathbb{C}$ ,  $(F(A \Rightarrow B), F(\epsilon))$  is the exponential of  $F(A)$  and  $F(B)$  in  $\mathbb{D}$ .

We call two Cartesian closed categories *equivalent* if there exists a Cartesian closed functor between them, which is an equivalence when considered as a plain functor.

**Remark 2.2.9.** The definition of a Cartesian closed category is equivalent to asking for right adjoints to the functor  $A \times -$ , for all objects  $A$ . Equivalently this can be expressed as the existence of a bijection between the hom-sets

$$\text{Hom}(A \times B, C) \cong \text{Hom}(B, A \Rightarrow C),$$

which is natural in  $A$  and  $B$ .

Note that we could equivalently specify a CCC by asking for a right adjoint to  $- \times A$ , however it is important to use the formulation given in order to give the most natural correspondence with the FMC.

**Remark 2.2.10.** As expected, two Cartesian (closed) categories  $\mathbb{C}$  and  $\mathbb{D}$  are equivalent by the above definition(s), if and only if there exist Cartesian (closed)-functors  $F : \mathbb{C} \rightarrow \mathbb{D}$  and  $G : \mathbb{D} \rightarrow \mathbb{C}$  such that  $F; G \simeq \text{id}_{\mathbb{C}}$  and  $G; F \simeq \text{id}_{\mathbb{D}}$ , using an appropriate definition of *Cartesian (closed) natural isomorphisms*, which we do not give here. In particular, if we have a Cartesian closed functor  $F : \mathbb{C} \rightarrow \mathbb{D}$  with a plain inverse functor  $G : \mathbb{D} \rightarrow \mathbb{C}$ , then  $G$  is automatically a Cartesian closed functor.

**Example 2.2.11.** The category **Set** of sets and functions is a Cartesian closed category, with the product given by the Cartesian product of sets and the exponential object  $B \Rightarrow C$  given by the set of all functions from  $B$  to  $C$ .

## 2.3 Signatures and The Free Cartesian Closed Category

The free Cartesian closed category is, intuitively, a Cartesian closed category containing some *generating* objects and morphisms, where two morphisms are equal if and only if their equality can be derived by the axioms of a CCC. That is, no more morphisms are identified than is strictly necessary in order for the category to be a CCC. We generate the free CCC over a set of primitive morphisms of forms such as  $f : (A \rightarrow B) \rightarrow (C \times D)$ , in order to adequately deal with (higher-order) constants with similar types.

We first define a notion of higher-order signature, which is the data over which we will generate a free Cartesian closed category.

**Definition 2.3.1.** A (*Cartesian closed*) *signature* is a tuple  $\Sigma = (\Sigma_0, \Sigma_1, \text{dom}, \text{cod})$ , where  $\Sigma_0$  is a set of sorts,  $\Sigma_1$  is a set of function symbols and  $\text{dom}, \text{cod} : \Sigma_1 \rightarrow \Sigma_0^{\times \rightarrow}$  are a pair of functions specifying the *domain* and *co-domain* of the function symbols, where  $\Sigma_0^{\times \rightarrow}$  is freely generated by the grammar

$$A, B ::= \alpha \mid \top \mid A \times B \mid A \rightarrow B,$$

with  $\alpha \in \Sigma_0$ . A *Cartesian* signature is defined similarly, except replacing  $\Sigma_0^{\times \rightarrow}$  with  $\Sigma_0^\times$ , which is freely generated by the grammar

$$A, B ::= \alpha \mid \top \mid A \times B.$$

We will sometimes write  $f : A \rightarrow B$  when  $f \in \Sigma_1$  to denote that  $\text{dom}(f) = A$  and  $\text{cod}(f) = B$ .

**Example 2.3.2.** We can form a signature  $\Sigma = (\Sigma_0, \Sigma_1, \text{dom}, \text{cod})$  with two base types  $\Sigma_0 = \{\alpha, \beta\}$  and constants of type

$$a : 1 \rightarrow \alpha \quad m : \alpha \times \alpha \rightarrow \alpha \quad b : 1 \rightarrow \beta \quad n : \beta \times \beta \rightarrow \beta$$

by setting  $\Sigma_1 = \{a, b, m, n\}$  and  $\text{dom}, \text{cod} : \Sigma_1 \rightarrow \Sigma_0^{\times \rightarrow}$  such that

$$\begin{array}{ll} \text{dom}(a) = 1 & \text{cod}(a) = \alpha \\ \text{dom}(m) = \alpha \times \alpha & \text{cod}(m) = \alpha \\ \text{dom}(b) = 1 & \text{cod}(b) = \beta \\ \text{dom}(n) = \beta \times \beta & \text{cod}(n) = \beta. \end{array}$$

**Definition 2.3.3.** Given signatures  $\Sigma = (\Sigma_0, \Sigma_1, \text{dom}_\Sigma, \text{cod}_\Sigma)$  and  $\Pi = (\Pi_0, \Pi_1, \text{dom}_\Pi, \text{cod}_\Pi)$ , a *signature homomorphism*  $f : \Sigma \rightarrow \Pi$  is defined as a pair of functions  $f_0 : \Sigma_0 \rightarrow \Pi_0$  and  $f_1 : \Sigma_1 \rightarrow \Pi_1$  such that the following diagrams commute:

$$\begin{array}{ccc} \Sigma_1 & \xrightarrow{\text{dom}_\Sigma} & \Sigma_0^{\times \rightarrow} \\ f_1 \downarrow & & \downarrow f_0^{\times \rightarrow} \\ \Pi_1 & \xrightarrow{\text{dom}_\Pi} & \Pi_0^{\times \rightarrow} \end{array} \quad \begin{array}{ccc} \Sigma_1 & \xrightarrow{\text{cod}_\Sigma} & \Sigma_0^{\times \rightarrow} \\ f_1 \downarrow & & \downarrow f_0^{\times \rightarrow} \\ \Pi_1 & \xrightarrow{\text{cod}_\Pi} & \Pi_0^{\times \rightarrow} \end{array}$$

where  $f_0^{\times \rightarrow}$  is the lifting of  $f_0 : \Sigma_0 \rightarrow \Pi_0$  to  $f_0^{\times \rightarrow} : \Sigma_0^{\times \rightarrow} \rightarrow \Pi_0^{\times \rightarrow}$  defined in the obvious pointwise way.

**Example 2.3.4.** Consider the signature  $\Sigma$  from Example 2.3.2, and a second signature  $\Pi = (\Pi_0, \Pi_1, \text{dom}_\Pi, \text{cod}_\Pi)$  with a single base type  $\Sigma_0 = \{\gamma\}$  and constants of type

$$c : 1 \rightarrow \gamma \quad p : \gamma \times \gamma \rightarrow \gamma ,$$

that is, we have  $\Pi_1 = \{c, p\}$  and  $\text{dom}_\Pi, \text{cod}_\Pi : \Pi_1 \rightarrow \Pi_0^{\times \rightarrow}$  such that

$$\begin{aligned} \text{dom}_\Pi(c) &= 1 & \text{cod}_\Pi(c) &= \gamma \\ \text{dom}_\Pi(p) &= \gamma \times \gamma & \text{cod}_\Pi(p) &= \gamma . \end{aligned}$$

Then an example of a signature homomorphism  $f : \Sigma \rightarrow \Pi$  is given by a pair of functions  $f_0 : \Sigma_0 \rightarrow \Pi_0$  and  $f_1 : \Sigma_1 \rightarrow \Pi_1$  such that

$$\begin{aligned} f_0(\alpha) &= f_0(\beta) = \gamma \\ f_1(a) &= f_1(b) = c \\ f_1(m) &= f_1(n) = p , \end{aligned}$$

where in particular the required commuting diagrams express that  $f_1$  must respect the (co-)domain of constant symbols with respect to the transformation of base types described by  $f_0$ .

Let **CCCSig** be the category of Cartesian closed signatures and their homomorphisms, and **CCCcat** be the category of Cartesian closed categories and Cartesian closed functors. There is an evident *forgetful* functor  $U : \text{CCCcat} \rightarrow \text{CCCSig}$ , whose left-adjoint is the *free* functor  $F : \text{CCCSig} \rightarrow \text{CCCcat}$ .

$$\text{CCCSig} \begin{array}{c} \xrightarrow{F} \\ \perp \\ \xleftarrow{U} \end{array} \text{CCCcat}$$

For a signature  $\Sigma$ , we call  $F(\Sigma)$  the *free Cartesian closed category over  $\Sigma$* . Note, the free CCC is indeed unique up to equivalence. Given such a signature and a CCC  $(\mathbb{C}, \Rightarrow, \times, \top)$ , the universal property of the adjunction then guarantees that for every signature homomorphism  $f : \Sigma \rightarrow U(\mathbb{C})$ , there exists a unique Cartesian closed functor  $[-] : F(\Sigma) \rightarrow \mathbb{C}$  such that:

$$\begin{array}{ccc} \Sigma & \hookrightarrow & UF(\Sigma) \\ & \searrow f & \downarrow U([-]) \\ & & U(\mathbb{C}) \end{array}$$

commutes. This leads to the following definition.

**Definition 2.3.5.** A Cartesian (closed) category  $\mathbb{X}$  is the *free Cartesian (closed) category over a signature  $\Sigma$*  if it is equipped with an *interpretation*  $i : \Sigma \rightarrow U(\mathbb{X})$  such that for every Cartesian (closed) category  $\mathbb{C}$  and signature homomorphism  $f : \Sigma \rightarrow U(\mathbb{C})$ , there exists a unique Cartesian (closed) functor  $\{-\} : \mathbb{X} \rightarrow \mathbb{C}$  such that the following diagram commutes

$$\begin{array}{ccc} \Sigma & \xrightarrow{i} & U(\mathbb{X}) \\ & \searrow f & \downarrow U(\{-\}) \\ & & U(\mathbb{C}) \end{array}$$

We will denote the free Cartesian closed category over  $\Sigma$  by  $\text{CCC}(\Sigma)$ .

Stated in other terms, the free CCC over  $\Sigma$  is initial in the category of CCCs equipped with the extra structure of a specified set of morphisms  $\Sigma$ , and functors which respect this extra structure. As such ‘the free CCC over  $\Sigma$ ’ is to be read not as ‘the (free CCC) over  $\Sigma$ ’, where  $\Sigma$  is some generating base category, but rather as ‘the free (CCC over  $\Sigma$ )’.

## 2.4 The Simply-Typed $\lambda$ -Calculus with Patterns

We define the simply-typed  $\lambda$ -calculus with pattern matching [56, 52]. We will use a signature with a little extra structure, in order to capture the distinction between values and computations that will be made in the FMC, introduced in Chapter 3. In particular, we split the set of constants  $\Sigma_1$  into the sets  $\Sigma_c$  and  $\Sigma_v$  of computation and value symbols, and require value constants to have the ‘empty’ domain  $\top$ . Correspondingly, note that there are two typing rules for constants: one for values, and one for computations.

**Definition 2.4.1.** An  $\lambda$ -signature with values is a signature  $(\Sigma_0, \Sigma_c \uplus \Sigma_v, \text{dom}, \text{cod})$ , where  $\Sigma_c \uplus \Sigma_v$  is the disjoint sum of sets  $\Sigma_c$  and  $\Sigma_v$  and where  $\text{dom}(v) = \top$  whenever  $v \in \Sigma_v$ . We will write  $f : A \rightarrow B \in \Sigma_c$  for a computation symbol  $f \in \Sigma_1$  such that  $\text{dom}(f) = A$  and  $\text{cod}(f) = B$ . For a value symbol  $v$ , we will write  $v : A \in \Sigma_v$  when  $\text{cod}(v) = A$ . When writing a signature with values, we will often then omit the (co-)domain, so that we write  $\Sigma = (\Sigma_0, \Sigma_c, \Sigma_v)$ . We will often refer to this as just a *signature*, when there is no ambiguity.

**Definition 2.4.2.** The  $\lambda$ -calculus with pattern matching generated by sets  $\Sigma_v$  and  $\Sigma_c$  of value and computation symbols is given by the following grammar:

$$\begin{aligned} M, N &::= x \mid v \mid c @ M \mid M @ N \mid \lambda p. M \mid (M_1, \dots, M_n) \\ p, q, r, s, t &::= x \mid (p_1, \dots, p_n) \end{aligned}$$

where from left to right the *term* constructors are a *variable*, a *value constant* where  $v \in \Sigma_v$ , a *computation constant* where  $f \in \Sigma_c$ , an *application*, an *abstraction* on the

$$\begin{array}{c}
\frac{}{\Pi, x : A, \Pi' \vdash x : A} \text{var} \quad \frac{v : A \in \Sigma_v}{\Pi \vdash v : A} \text{vconst} \quad \frac{\Pi \vdash M : A \quad c : A \rightarrow B \in \Sigma_c}{\Pi \vdash c @ M : B} \text{cconst} \\
\\
\frac{\{\Pi \vdash M_i : A_i\}_{i \in I}}{\Pi \vdash (M_1, \dots, M_n) : A_1 \times \dots \times A_n} \text{tuple} \quad \frac{p_1 : A_1, \dots, p_n : A_n, \Pi \vdash M : B}{(p_1, \dots, p_n) : A_1 \times \dots \times A_n, \Pi \vdash M : B} \text{pm} \\
\\
\frac{\Pi \vdash N : A \quad \Pi \vdash M : A \rightarrow B}{\Pi \vdash M @ N : B} \text{app} \quad \frac{p : A, \Pi \vdash M : B}{\Pi \vdash \lambda p. M : A \rightarrow B} \text{abs}
\end{array}$$

Figure 2.1: Typing rules for the  $\lambda$ -calculus with pattern matching

pattern  $p$ , which binds the variables of  $p$  in  $M$ , and a  $n$ -ary tuple of terms. *Pattern* constructors are given by a *variable* or a *tuple of patterns*, respectively. We freely allow coercion from patterns to terms. Terms are considered modulo  $\alpha$ -equivalence.

**Definition 2.4.3.** Given a signature  $\Sigma = (\Sigma_0, \Sigma_c, \Sigma_v)$ , *simple types* are given by the following grammar

$$A, B ::= \alpha \mid \top \mid A \times B \mid A \rightarrow B$$

for  $\alpha \in \Sigma_0$ . The *typing rules* for the *simply-typed  $\lambda$ -calculus with pattern matching (STLC)* over  $\Sigma$ , for terms generated over  $\Sigma_v$  and  $\Sigma_c$ , and types generated over  $\Sigma_0$ , are given in Figure 2.1. A *typing judgement* is of the form

$$\Pi \vdash M : A,$$

where a *context*  $\Pi$  is a finite sequence of variables, each associated with a type:  $x_1 : A_1, \dots, x_n : A_n$ .

**Definition 2.4.4.** The *equational theory* of the STLC is the least equivalence generated by the following laws, closed under any context:

$$\begin{array}{lll}
\text{Beta (Function):} & (\lambda p. M) @ P & =_\beta M \{P/p\} : A \\
\text{Eta (Function):} & \lambda p. M @ p & =_\eta M : A \rightarrow B \\
\text{Eta (Product):} & (\pi_1(M), \dots, \pi_n(M)) & =_\pi M : A_1 \times \dots \times A_n
\end{array}$$

where for  $\eta$ ,  $\text{fv}(p) \cap \text{fv}(M) = \emptyset$ , and in the last case, we define  $\pi_i = \lambda(x_1, \dots, x_n). x_i$ . Substitution *pattern matches* in the sense that  $\{(P_1, \dots, P_n)/(p_1, \dots, p_n), \dots\} = \{P_1/p_1, \dots, P_n/p_n, \dots\}$ . Otherwise, (simultaneous) substitution is capture-avoiding and defined as standard.

Note that in this calculus, the beta law for products  $(\lambda(x_1, x_2). x_i)(M_1, M_2) = M_i$  is given by the pattern matching performed by the definition of substitution. Admissible typing rules are given in Figure 2.2.



$$\begin{array}{c}
\frac{\Pi \vdash N : A \quad p : A, \Pi \vdash M : B}{\Pi \vdash M\{N/p\} : B} \text{ cut} \\
\\
\frac{\Pi, p : A, q : B, \Pi' \vdash M : B}{\Pi, q : B, p : A, \Pi' \vdash M : B} \text{ exch} \quad \frac{\Pi \vdash M : B}{\Pi, p : A \vdash M : B} \text{ weak}
\end{array}$$

Figure 2.2: Admissible rules for the the  $\lambda$ -calculus with pattern matching

## 2.5 The Category of Simply-Typed $\lambda$ -Terms

Finally, we define the category of simply-typed  $\lambda$ -terms and recall its equivalence with the free Cartesian closed category generated over the same signature, which is due to Lambek [58, 59, 60].

We give more equipment than is necessary in the following definition, since this will make comparison with the FMC easier later on. Following the definition, we verify that the extra equipment defined in fact arises from the necessary equipment in the usual way.

**Definition 2.5.1.** Given a signature  $\Sigma = (\Sigma_0, \Sigma_v, \Sigma_c)$ , the category of simply-typed  $\lambda$ -terms over  $\Sigma$   $\Lambda(\Sigma)$  is defined with:

- Objects: simple types generated over  $\Sigma_0$ ,
- Morphisms:  $Hom(A, B)$  is given by the set of simply-typed  $\lambda$ -terms  $p : A \vdash M : B$  over  $\Sigma$ , modulo the equational theory  $=$ ,
- Composition: given morphisms  $q : A \vdash N : B \in Hom(A, B)$  and  $p : B \vdash M : C \in Hom(B, C)$ ,  $N; M \in Hom(A, C)$  is given by substitution  $q : A \vdash M\{N/p\} : C$ ,
- Identity: given on every type  $A$  by  $p : A \vdash p : A$ ,
- Products: given on types  $A$  and  $B$  by  $A \times B$ , with unit  $\top$ , and its action on morphisms and associated natural transformations:

$$\begin{array}{lll}
M \times N : A \times B \vdash A' \times B' & = & (p, q) \vdash (M, N) \\
! : A \vdash \top & = & p \vdash () \\
\Delta : A \vdash A \times A & = & p \vdash (p, p) \\
\langle M, N \rangle : A \vdash B \times C & = & r \vdash (M\{r/p\}, N\{r/q\}) \\
\pi_1 : A \times B \vdash A & = & (p, q) \vdash p \\
\pi_2 : A \times B \vdash B & = & (p, q) \vdash q \\
\text{sym} : A \times B \vdash B \times A & = & (p, q) \vdash (q, p)
\end{array}$$

for  $p : A \vdash M : A'$  and  $q : B \vdash N : B'$ ,

- Exponents: on types  $A$  and  $B$  by  $A \rightarrow B$ , with its action on morphisms and associated natural transformations:

$$\begin{array}{lll}
\epsilon : A \times (A \rightarrow B) \vdash B & = & (p, f) \vdash f@p \\
\eta : A \vdash B \rightarrow (B \times A) & = & p \vdash \lambda q.(q, p) \\
\text{curry}(M) : B \vdash (A \rightarrow C) & = & q \vdash \lambda p.M \\
P \rightarrow Q : (A \rightarrow B) \vdash (C \rightarrow D) & = & f \vdash \lambda p.Q\{f(P)/q\}
\end{array}$$

for  $(p, q) : A \times B \vdash M : C$  and  $p : C \vdash P : A$  and  $q : B \vdash Q : D$ .

- Associators and unitors:

$$\begin{array}{lll}
\alpha : A \times (B \times C) \vdash (A \times B) \times C & = & (p, (q, r)) : \vdash ((p, q), r) \\
\lambda : \top \times A \vdash A & = & ((), p) : \vdash p \\
\rho : A \times \top \vdash A & = & (p, ()) \vdash p
\end{array}$$

We may sometimes omit the bracketing of the context, since it is unambiguous.

**Remark 2.5.2.** We have overspecified the necessary equipment, but the equipment given is consistent in the expected way: we can define the following equipment in terms of that required for the definition of a Cartesian closed category.

$$\begin{array}{ll}
\Delta & \triangleq \langle \text{id}, \text{id} \rangle \\
M \times N & \triangleq \langle \pi_1 ; M, \pi_2 ; N \rangle \\
\text{sym} & \triangleq \langle \pi_2, \pi_1 \rangle \\
\eta & \triangleq \text{curry}(\text{id}) \\
M \rightarrow N & \triangleq \text{curry}(\langle \pi_1 ; M, \pi_2 \rangle ; \epsilon ; N)
\end{array}$$

The corresponding terms given in Definition 2.5.1 can be seen to match this, up to a small amount of simplification using the equational theory. Conversely, we can define the following equipment in terms of the natural transformations  $\Delta, !, \eta, \epsilon$  and the bifunctors  $\times$  and  $\rightarrow$ .

$$\begin{array}{ll}
\langle M, N \rangle & \triangleq \Delta ; (M \times N) \\
\pi_1 & \triangleq (! \times \text{id}) \\
\pi_2 & \triangleq (\text{id} \times !) \\
\text{curry}(P) & \triangleq \eta ; (\text{id} \rightarrow P)
\end{array}$$

We will elide associativity and unitality in the remaining thesis, which will be justified by the following theorem.

The following theorem is due to Lambek [60], and extends the Curry-Howard correspondence between the simply-typed  $\lambda$ -calculus and intuitionistic logic to include their categorical counterpart, Cartesian closed categories. We sketch the proof, since we are working with a mostly standard definition of the  $\lambda$ -calculus. We will deal with the *strict* free CCC, where associativity and unitor isomorphisms are true

identities. This is justified by MacLane's *coherence theorem*, which says that the free monoidal category is (monoidally) equivalent to the *strict* free monoidal category [71]. Pay particular attention to the interpretation of computation constants in the following proof.

**Theorem 2.5.3** (Curry-Howard-Lambek). *The category  $\Lambda(\Sigma)$  is Cartesian closed and equivalent to  $\text{CCC}(\Sigma)$ .*

*Proof.* We sketch the proof. The category  $\Lambda(\Sigma)$  is Cartesian closed, as follows. To see the existence of products, let  $q_1 : A_1 \vdash M_1 : A_1$  and  $q_2 : A_2 \vdash M_2 : A_2$  in

$$\begin{aligned} \langle M_1, M_2 \rangle; \pi_i &= p \vdash (M_1\{p/q_1\}, M_2\{p/q_2\}; (s_1, s_2) \vdash s_i \\ &= p \vdash M_i\{p/q_i\} \\ &=_{\alpha} q_i \vdash M_i, \end{aligned}$$

where we recall composition is substitution, which implements pattern-matching. For uniqueness of products, let  $s : A \vdash M : B \times C$ ,  $M' = M\{r/s\}$  and  $\pi'_i = \lambda(p_1, p_2).p_i$  in

$$\begin{aligned} \langle M; \pi_1, M; \pi_2 \rangle &= r \vdash (M'; (q_1, q_2) \vdash q_1, M'; (q'_1, q'_2) \vdash q'_2) \\ &=_{\eta} r \vdash ((\pi'_1 @ M', \pi'_2 @ M'); (q_1, q_2) \vdash q_1, \\ &\quad (\pi'_1 @ M', \pi'_2 @ M'); (q'_1, q'_2) \vdash q'_2) \\ &= r \vdash (\pi'_1 @ M', \pi'_2 @ M') \\ &=_{\eta} r \vdash M' \\ &=_{\alpha} s \vdash M \end{aligned}$$

For existence of exponents, let  $(q, r) : (A \times B) \vdash M : C$  in

$$\begin{aligned} (\text{id} \times \text{curry}(M)); \epsilon &= (p, r) \vdash (p, \lambda q.M); (p', f) \vdash f @ p' \\ &= (p, r) \vdash (\lambda q.M) @ p \\ &=_{\beta} (p, r) \vdash M\{p/q\} \\ &=_{\alpha} (q, r) \vdash M \end{aligned}$$

For uniqueness of exponents, let  $q : A \vdash N : B \rightarrow C$  in

$$\begin{aligned} \text{curry}((\text{id} \times N); \epsilon) &= \text{curry}((p, q) \vdash (p, N); (p', f) \vdash f @ p') \\ &= \text{curry}((p, q) \vdash N @ p) \\ &= q \vdash \lambda p. N @ p \\ &=_{\eta} q \vdash N \end{aligned}$$

This defines a functor from  $\text{CCC}(\Sigma)$  to  $\Lambda(\Sigma)$  by freeness of  $\text{CCC}(\Sigma)$ . We present the inverse functor  $\{-\} : \Lambda(\Sigma) \rightarrow \text{CCC}(\Sigma)$  below. We work with a *strict* CCC,

as justified by MacLane's strictification theorem [71]. We present the binary and nullary cases of tuples, with the  $n$ -ary case the obvious generalization.

$$\begin{aligned}
\{\Gamma, x : A, \Delta \vdash x : A\} &= !_{\Gamma} \times \text{id}_A \times !_{\Delta} \\
\{\Pi \vdash v : A\} &= !_{\Pi} ; v \\
\{\Pi \vdash c @ M : B\} &= \{\Pi \vdash M : A\} ; c \\
\{\Pi \vdash (M, N) : A \times B\} &= [\{\Pi \vdash M : A\}, \{\Pi \vdash N : B\}] \\
\{(p, q) : A \times B, \Pi \vdash M : C\} &= \{p : A, q : B, \Pi \vdash M : C\} \\
\{() : \top, \Pi \times B \vdash M : C\} &= \{\Pi \vdash M : C\} \\
\{\Pi \vdash () : \top\} &= !_{\Pi} \\
\{\Pi \vdash M @ N : B\} &= [\{\Pi \vdash N : A\}, \{\Pi \vdash M : A \rightarrow B\}] ; \epsilon \\
\{\Pi \vdash \lambda p. M : A \rightarrow B\} &= \eta ; (\text{id}_A \rightarrow \{\Pi, p : A \vdash M : B\}) \quad \square
\end{aligned}$$

**Remark 2.5.4.** The translation of the admissible rules of cut, exchange and weakening into a Cartesian closed category are as follows.

$$\begin{aligned}
\{\Pi', \Pi \vdash M\{N/p\} : B\} &= \Delta_{\Pi', \Pi}; (\text{id}_{\Pi'} \times \{\Pi', \Pi \vdash N : A\} \times \text{id}_{\Pi}); \\
&\quad \{\Pi', p : A, \Pi \vdash M : B\} \\
\{\Pi, p : A, q : B, \Pi' \vdash M : C\} &= (\text{id}_{\Pi} \times \text{sym}_{A, B} \times \text{id}_{\Pi'}); \{\Pi, q : B, p : A, \Pi' \vdash M : C\} \\
\{\Pi, p : A \vdash M : B\} &= (!_A \times \text{id}_{\Pi}); \{\Pi \vdash M : C\}
\end{aligned}$$

Note how the equational theory of the  $\lambda$ -calculus corresponds to the axiomatization of a CCC in terms of its universal properties, as given earlier in the chapter. We will see later how this compares to the approach of the FMC.

**Remark 2.5.5.** By restricting the  $\lambda$ -calculus so that it doesn't include application or abstraction, we can similarly achieve a correspondence between the *first-order* fragment of the  $\lambda$ -calculus and the free Cartesian category.

From now on, we can use the simply-typed  $\lambda$ -calculus as a convenient notation for the free Cartesian closed category. With this correspondence in mind, we will write  $\text{CCC}(\Sigma)$  in place of  $\Lambda(\Sigma)$ , and, in particular, we will work modulo associativity and unitality of the  $\lambda$ -calculus.

## Chapter 3

# The Functional Machine Calculus

The material presented in this chapter was developed by Heijltjes and published in [9]. The Functional Machine Calculus was amply motivated in the introduction, so we focus on concise definitions here for reference.

There is one significant difference between the calculus presented in the introduction and the one presented here. Here, we make a distinction between *values* (which live on the stack) and *computations* (which act on the stack), following Call-by-Push-Value [66]. We further introduce coercions *force* and *thunk*, taking values to computations and vice-versa. This variant was mentioned, but not developed, in previous publications [9]. The reason for the distinction is to achieve in Chapter 4 the most natural correspondence with the  $\lambda$ -calculus. In particular, it turns out that *force* and *thunk* are naturally translated as the *application* and *abstraction*, respectively, of the  $\lambda$ -calculus. Furthermore, such a variant is natural for the inclusion of terms of base type: these highlight the difference between *values*, which live on the stack, and *computations*, which run on the machine, a natural operational distinction.<sup>1</sup>

After introducing the *Functional Machine Calculus with Values* – which we will abbreviate !FMC, after the *thunk* constructor *!M* – the definitions relating to the “classic” FMC, given in the introduction to this thesis, are collected. The classic variant of the Functional Machine Calculus will be used in the proof of strong normalisation of Chapter 6, where the distinction between values and computations is no longer useful.

We fix a countable set of locations  $A = \{a, b, c, \dots\}$  throughout this chapter, and the remaining thesis, except where otherwise mentioned. We take  $a, b$  to range over all locations.

---

<sup>1</sup>One suggestion that the original system is not quite right for the inclusion of base types is hidden in the typing rules of Figure 1.5 in the introduction. Recalling that all terms defined in the original grammar end with a trailing  $\star$ , which is usually omitted, we see that we have really typed  $x.\star : \alpha$ , thus giving a term of base type a continuation (albeit the trivial one).

### 3.1 The FMC with Values: !FMC

**Definition 3.1.1.** The *terms* of the *Functional Machine Calculus with Values*, denoted !FMC, generated by sets  $\Sigma_v$  and  $\Sigma_c$  of value and computation symbols, respectively, are given by the following grammar of *computations* and *values*, respectively:

$$\begin{aligned} M, N, P &::= \star \mid c.M \mid [V]a.M \mid a\langle x \rangle.M \mid ?V.M \\ V, W &::= x \mid v \mid !M \end{aligned}$$

where, from left to right, the *computation* constructors are the *identity*, a *sequential constant* with  $c \in \Sigma_c$ , *application* or *push action* on location  $a$ , an *abstraction* or *pop action* on location  $a$ , which binds  $x$  in  $M$ , and a *sequential execution* (or *force*) of a value. The *value* constructors are a *variable*, a *value constant* with  $v \in \Sigma_v$ , and a *thunk* of a computation. Terms are considered modulo  $\alpha$ -equivalence.

Formally, the set of *free variables* of a computation  $M$  or value  $V$ ,  $\text{fv}(M)$  or  $\text{fv}(V)$ , respectively, is defined as follows.

$$\begin{aligned} \text{fv}(\star) &= \emptyset & \text{fv}(x) &= \{x\} \\ \text{fv}(c.M) &= \text{fv}(M) & \text{fv}(v) &= \emptyset \\ \text{fv}([V]a.M) &= \text{fv}(V) \cup \text{fv}(M) & \text{fv}(!M) &= \text{fv}(M) \\ \text{fv}(a\langle x \rangle.M) &= \text{fv}(M) \setminus \{x\} \\ \text{fv}(?V.M) &= \text{fv}(V) \cup \text{fv}(M) \end{aligned}$$

Note how the scoping of the  $?$  and  $!$  constructs is unambiguous. We will omit the trailing  $\star$  from a computation, and use a *main* location  $\lambda$ , omitted from the notation. We use Barendregt's variable convention, *i.e.*, that all bound variables are chosen to be different from free variables.

**Definition 3.1.2.** Capture-avoiding *composition*, or *sequencing*,  $N;M$  is given by

$$\begin{aligned} \star;M &= M & c.N;M &= c.(N;M) \\ [V]a.N;M &= [V]a.(N;M) & a\langle x \rangle.N;M &= a\langle x \rangle.(N;M) \\ ?V.N;M &= ?V.(N;M) \end{aligned}$$

where, in the abstraction case,  $x \notin \text{fv}(M)$ . Capture-avoiding *substitution of values*  $\{V/x\}M$  and  $\{V/x\}W$  into computations and values, respectively, is given by

$$\begin{aligned} \{V/x\}\star &= \star & \{V/x\}x &= V \\ \{V/x\}c.M &= c.\{V/x\}M & \{V/x\}y &= y \\ \{V/x\}[W]a.M &= [\{V/x\}W]a.\{V/x\}M & \{V/x\}v &= v \\ \{V/x\}a\langle y \rangle.M &= a\langle y \rangle.\{V/x\}M & \{V/x\}!M &= !\{V/x\}M \\ \{V/x\}?W.M &= ?\{V/x\}W.\{V/x\}M \end{aligned}$$

where  $x \neq y$  and, in the abstraction case,  $y \notin \text{fv}(V)$ .

**Remark 3.1.3.** Sequencing can easily be shown to be associative. It would be possible to take sequencing as a primitive constructor, but it would necessitate working modulo associativity. Instead, we take prefixing as the primitive constructor, which can be seen as a choice to associate to the right.

**Definition 3.1.4.** The *functional abstract machine* is given by the following data. A *stack* of terms  $S$  is defined below left, and written with the top element to the right. A *memory*  $S_A$  is a family of stacks or streams in  $A$ , defined below right, and can formally be regarded as a function from the set of locations  $A$  to the set of stacks.

$$S ::= \epsilon \mid S \cdot V \qquad S_A ::= \{ S_a \mid a \in A \}$$

Where  $B \subseteq A$ , we write  $S_B$  for the restriction of  $S_A$  to  $B$ . We write  $S_a$  for  $a \in A$  to access the stack held in memory  $S_A$  at location  $a$ , and we allow identification of  $S_{\{a\}}$  with  $S_a$ . Let  $S_B; S_C$  denote the copairing of families (considered as functions) of stacks in disjoint sets  $B$  and  $C$ . In particular this means we have  $S_A = S_{A \setminus \{a\}}; S_a$ . A *state* is a pair  $(S_A, M)$  of a memory and a term. The *transitions* or *steps* of the machine are given below.

$$\frac{(\ S_{A \setminus \{a\}}; S_a \ , \ [V]a.M \ )}{(\ S_{A \setminus \{a\}}; S_a \cdot V \ , \ M \ )} \quad \frac{(\ S_{A \setminus \{a\}}; S_a \cdot V \ , \ a\langle x \rangle.M \ )}{(\ S_{A \setminus \{a\}}; S_a \ , \ \{V/x\}M \ )} \quad \frac{(\ S_A \ , \ ?!N.M \ )}{(\ S_A \ , \ N; M \ )}$$

A *run* of the machine is a sequence of steps, written as  $(S_A, M) \Downarrow (T_A, N)$  or with a double line as below.

$$\frac{(\ S_A \ , \ M \ )}{(\ T_A \ , \ N \ )}.$$

A *successful run* of the machine is a run  $(S_A, M) \Downarrow (T_A, \star)$ , and in this case we will denote  $T_A$  by simply  $(S_A, M) \Downarrow$ .

We will introduce the reduction relation for the !FMC, which will be studied throughout this thesis, but first we will need a notion of *sequential context*.

**Definition 3.1.5.** A *(sequential) context* is defined as an !FMC-term with either a hole  $\{ \}_c$  in place of a sequential constant, or a hole  $\{ \}_v$  in place of a variable:

$$\begin{aligned} K &\triangleq \{ \}_c.M \mid c.K \mid [V]a.K \mid [L]a.M \mid a\langle x \rangle.K \mid ?V.K \mid ?L.M \\ L &\triangleq \{ \}_v \mid !K \end{aligned}$$

We will write  $K\{ \}_c$  or  $K\{ \}_v$  to denote whether the context contains a hole for a computation or a value and refer to *computation* or *value* contexts, respectively. Given a computation  $M$  and a value  $V$ , define  $K\{M\}_c$  and  $K\{V\}_v$  to be the terms given by substituting  $M$  and  $V$  in for the holes  $\{ \}_c$  and  $\{ \}_v$ , respectively. Substitution is defined as usual for values, except that in  $a\langle x \rangle.K\{V\}$ ,  $x$  captures in  $V$ . *Substitution of computations* is defined similarly, except with the case  $\{N\}_c.M = N; M$ .

**Definition 3.1.6.** We define the *beta*, *force*, *thunk* and *permutation reductions* as the following rewrite rules on !FMC-terms, respectively, applicable in all contexts:

$$\begin{aligned} [V]a.a\langle x \rangle.M &\rightarrow_\beta \{V/x\}M \\ ?!N.M &\rightarrow_\phi N; M \\ !?V &\rightarrow_\tau V \\ [V]a.b\langle x \rangle.M &\rightarrow_\pi b\langle x \rangle.[V]a.M, \end{aligned}$$

where in the final case,  $x \notin \text{fv}(V)$  and  $a \neq b$ . We write  $\rightarrow$  to indicate the union of these reductions and  $\rightarrow^*$  to indicate the reflexive, symmetric, transitive closure of  $\rightarrow$ .

**Remark 3.1.7.** Note that the definition of sequential (computation) context means each equation applies in the presence of some continuation  $M$ , as in  $\{N\}_c.M = N; M$ , and that the abstractions appearing in the equational theory will not bind in this continuation (because of the capture-avoiding definition of sequencing). Note, further, that the definition of sequential context means that a redex can be identified as being in more than one context. However, this does not matter for the purposes of the equational theory: we have

if  $M = N$  then  $\{M\}_c.P; Q = \{N\}_c.P; Q$  if and only if  $\{M; P\}_c.Q = \{N; P\}_c.Q$ , which can easily be seen to follow from the definitions and associativity of sequencing.

Beta reduction is analogous to that of the  $\lambda$ -calculus. Permutation reduction allows push- and a pop-action on the same location to interact (by beta reduction) despite the presence of intervening pushes and pops on *other* locations.

In the following chapters on categorical semantics, we will give an equational theory which extends the one generated by the reduction relation, for example with equations such as  $\star =_{\text{id}} \langle x \rangle.[x]$  mentioned in the introduction. In Chapter 4, we formalize an appropriate definition of *machine equivalence* of terms (a kind of contextual equivalence based on the abstract machine), and show that each equation is valid with respect to this.

We introduce the following notation, which will be used throughout the thesis.

**Notation 3.1.8.** The *empty memory* is denoted  $\epsilon_A$ . A *singleton* memory  $a(S)$  is empty at every location except  $a$ , where it contains  $S$ , that is:  $a(S)_a = S$  and  $a(S)_b = \epsilon$ , for  $a \neq b$ . We omit the main location  $\lambda$  for singleton types, so  $\lambda(S)$  may be written as  $S$ . *Concatenation* of stacks is denoted by  $(\cdot)$ , which we will sometimes omit, so it is denoted instead by juxtaposition, and this extends to families pointwise, where it is denoted  $(;)$ , so  $S_A; T_A = \{S_a \cdot T_a \mid a \in A\}$ .

We use *vector* notation to denote a stack of variables, and *reverse* a vector by pointing the arrow left:

$$\vec{x} = x_1 \cdots x_n \quad \overleftarrow{x} = x_n \cdots x_1.$$

*Concatenation* of vectors given by juxtaposition. Note that, although stacks grow to the right, we count elements from the left, in order to avoid reindexing when a new element is added. This notation is extended to families, so that  $\vec{x}_A = \{\vec{x}_a \mid a \in A\}$ , the *reverse* of a family  $\vec{x}_A$  and concatenation defined pointwise: e.g.,  $\vec{x}_A \vec{y}_A = \{\vec{x}_a \vec{y}_a \mid a \in A\}$ .

We then lift the notation to sequences of abstractions and applications: given  $\vec{x}$  and  $\vec{x}_A$  as above, a stack  $S = V_1 \cdots V_n$  and memory  $S_A = \{S_a \mid a \in A\}$ , let

$$\begin{aligned} \langle \vec{x} \rangle.N &= \langle x_n \rangle \dots \langle x_1 \rangle.N & [S].N &= [V_1] \dots [V_n].N \\ \langle \overleftarrow{x}_A \rangle.M &= a_1 \langle \overleftarrow{x}_{a_1} \rangle \dots a_n \langle \overleftarrow{x}_{a_n} \rangle.M & [S_A].M &= [S_{a_1}]a_1 \dots [S_{a_n}]a_n.M \end{aligned}$$



for  $A = \{a_1, \dots, a_n\}$ . We will see later, in Lemma 4.1.2, that any choice of ordering on the locations above gives terms equal in the equational theory. This notation is extended to simultaneous substitutions as

$$\{S/\vec{x}\} = \{V_1/x_1, \dots, V_n/x_n\} \quad \{S_A/\vec{x}_A\} = \{S_{a_1}/\vec{x}_{a_1}, \dots, S_{a_n}/\vec{x}_{a_n}\},$$

so in particular it respects beta reduction in the sense that we have

$$[S].\langle\vec{x}\rangle.M \rightarrow_{\beta}^* \{S/\vec{x}\}M \quad [S_A].\langle\vec{x}_A\rangle.M \rightarrow_{\beta}^* \{S_A/\vec{x}_A\}M.$$

We let  $\text{fv}(\vec{x})$  and  $\text{fv}(\vec{x}_A)$  denote the underlying sets of variables of  $\vec{x}$  and  $\vec{x}_A$ , respectively.

**Remark 3.1.9.** Normal forms arrived at from the reduction rules are of the form

$$\langle\vec{x}_A\rangle.[S_A].(?x.\langle\vec{y}_A\rangle.[T_A])^*,$$

where  $(-)^*$  indicates some number of iterations of terms of that form.

## 3.2 Simple Types for the Functional Machine Calculus

We begin with the definition of !FMC-types (which coincide with FMC-types). Consider the elements which comprise an !FMC machine state: we have *values*, which comprise stack elements, the *stack* itself, a *memory* which is an indexed product of stacks. In the higher-order setting, values may of course be (thunks of) terms themselves. Correspondingly, we have the following grammar for types.

**Definition 3.2.1.** Given a set  $\Sigma_0$  of *base types*, *simple value*, *stack* and *memory types* over  $\Sigma_0$  are given, respectively, by:

$$\tau ::= \alpha \mid \vec{\sigma}_A \Rightarrow \vec{\tau}_A \quad \vec{\tau} ::= \tau_1 \dots \tau_n \quad \vec{\tau}_A ::= \{\vec{\tau}_a \mid a \in A\}$$

where  $\alpha$  is drawn from  $\Sigma_0$ . We also refer to stack and memory types as type vectors and type families, respectively.

We now define an !FMC-signature, which holds the sets of constant value and computation symbols and their types, over which the simply-typed !FMC will be generated.

**Definition 3.2.2.** An !FMC-*signature* is a tuple  $\Sigma_A = (\Sigma_0, \Sigma_c, \Sigma_v, \text{dom}, \text{cod}, \text{val})$ , where  $\Sigma_0$  is a set of sorts,  $\Sigma_c$  and  $\Sigma_v$  are sets of function and value symbols, respectively, and  $\text{dom}, \text{cod} : \Sigma_c \rightarrow \Sigma_0^m$  are a pair of functions specifying the *domain* and *co-domain* of the computation function symbols, respectively, and  $\text{val} : \Sigma_v \rightarrow \Sigma_0^v$  specifies the *value* of the value function symbols, where  $\Sigma_0^v$  and  $\Sigma_0^m$  are given by the set of value and memory types, respectively, generated over  $\Sigma_0$ . We will write  $c : \vec{\sigma}_A \Rightarrow \vec{\tau}_A \in \Sigma_c$  for a computation symbol  $c \in \Sigma_c$  such that  $\text{dom}(c) = \vec{\sigma}_A$  and  $\text{cod}(c) = \vec{\tau}_A$ . Similarly, we will write  $v : \tau \in \Sigma_v$  for a value symbol  $v \in \Sigma_v$ , such that  $\text{val}(v) = \tau$ . We often then denote an !FMC-signature as  $\Sigma_A = (\Sigma_0, \Sigma_c, \Sigma_v)$ , omitting the (co-)domain and value functions, and refer to this as just a *signature*, when there is no ambiguity.

In addition to values, stacks and a memory, we have the computation term itself which is under execution: this receives the type of an implication between memories. Overall, the type system will have two sorts of judgements: one for values, and one for computations.

**Definition 3.2.3.** Given an !FMC-signature  $\Sigma_A = (\Sigma_0, \Sigma_c, \Sigma_v)$ , the typing rules for the *simply-typed* !FMC over  $\Sigma_A$  assign to terms generated over  $\Sigma_v$  and  $\Sigma_c$  types generated over  $\Sigma_0$ , and are given in Figure 3.1 using notation given subsequently. A *computation type* is an implication between memory types:  $\vec{\sigma}_A \Rightarrow \vec{\tau}_A$ . *Computation* and *value* judgements are of the form

$$\Gamma \vdash_v V : \tau \quad \Gamma \vdash_c M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

respectively, where in each case the *context*  $\Gamma$  is a finite sequence of variables, each associated with a value type:  $x_1 : \tau_1, \dots, x_n : \tau_n$ . We will sometimes write  $\Gamma \vdash_c M, N : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  to indicate two terms of the same type, typed in the same context.

Admissible rules are shown in Figure 3.2.

**Notation 3.2.4.** Similar to Notation 3.1.8, we introduce the following. The *empty* type vector is  $\epsilon$ , and the empty memory type  $\epsilon_A$ . A *singleton* memory type  $a(\vec{\tau})$  is empty at every location except  $a$ , where it has  $\vec{\tau}$ : that is,  $a(\vec{\tau})_a = \vec{\tau}$  and  $a(\vec{\tau})_b = \epsilon$ , for  $a \neq b$ . We omit the main location  $\lambda$  for singleton types, so  $\lambda(\vec{\tau})$  may be written as  $\vec{\tau}$ . *Concatenation* of type vectors is denoted by juxtaposition, and extends to families point-wise,  $\vec{\sigma}_A \vec{\tau}_A = \{\vec{\sigma}_a \vec{\tau}_a \mid a \in A\}$ . The *reverse* of a type vector  $\vec{\tau} = \tau_1 \dots \tau_n$  is written  $\tilde{\tau} = \tau_n \dots \tau_1$ , with the reverse of a type family also defined pointwise. For a context  $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ , we may write  $\vec{x} : \vec{\tau}$ .

An equivalent way to view computation types is as an implication between two vectors of singletons:

$$a_1(\sigma_1) \dots a_n(\sigma_n) \Rightarrow b_1(\tau_1) \dots b_m(\tau_m) \quad a(\sigma) b(\tau) = b(\tau) a(\sigma) ,$$

considered modulo the permutation of types on different locations, as above.

**Remark 3.2.5** (Expansion and Sequencing). Note that the *strict sequencing* (seq) and *expansion* (exp) rules can be combined to give the (*general*) *left- and right-sequencing* (lseq and rseq, respectively). Conversely, we can recover the expansion rule by left-sequencing a term  $M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  with  $\star : \vec{\sigma}_A \vec{\rho}_A \Rightarrow \vec{\rho}_A \vec{\sigma}_A$ . Note further that the in the !FMC, the “left” or “right” orientation of a composition may differ on a per location basis. That is, one could type, for example, the composition of terms  $M : \vec{\rho}_A \Rightarrow a(\vec{\sigma})b(\vec{\sigma}'\vec{\tau}')$  and  $N : a(\vec{\sigma}\vec{\tau})b(\vec{\tau}') \Rightarrow \vec{v}_A$ .

We will often work with left sequencing as primary, which we will call simply *sequencing*, rather than with strict sequencing and expansion.

$$\begin{array}{c}
\frac{}{\Gamma \vdash_c \star : \vec{\tau}_A \Rightarrow \vec{\tau}_A} \text{id} \quad \frac{}{\Gamma, x : \tau, \Delta \vdash_v x : \tau} \text{var} \\
\\
\frac{\Gamma \vdash_c M : \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{v}_A \quad c : \vec{\rho}_A \Rightarrow \vec{\sigma}_A \in \Sigma_c}{\Gamma \vdash_c c.M : \vec{\rho}_A \vec{\tau}_A \Rightarrow \vec{v}_A} \text{cconst} \quad \frac{v : \tau \in \Sigma_v}{\Gamma \vdash_v v : \tau} \text{vconst} \\
\\
\frac{\Gamma \vdash_v V : \rho \quad \Gamma \vdash_c M : a(\rho) \vec{\sigma}_A \Rightarrow \vec{\tau}_A}{\Gamma \vdash_c [V]a.M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A} \text{app} \quad \frac{x : \rho, \Gamma \vdash_c M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A}{\Gamma \vdash_c a\langle x \rangle.M : a(\rho) \vec{\sigma}_A \Rightarrow \vec{\tau}_A} \text{abs} \\
\\
\frac{\Gamma \vdash_c M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A}{\Gamma \vdash_v !M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A} \text{thunk} \quad \frac{\Gamma \vdash_v V : \vec{\rho}_A \Rightarrow \vec{\sigma}_A \quad \Gamma \vdash_c M : \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{v}_A}{\Gamma \vdash_c ?V.M : \vec{\rho}_A \vec{\tau}_A \Rightarrow \vec{v}_A} \text{force}
\end{array}$$

Figure 3.1: Typing rules for the Functional Machine Calculus with Values

$$\begin{array}{c}
\frac{\Gamma \vdash_c M : \vec{\rho}_A \Rightarrow \vec{\sigma}_A \quad \Gamma \vdash_c N : \vec{\sigma}_A \Rightarrow \vec{\tau}_A}{\Gamma \vdash_c M ; N : \vec{\rho}_A \Rightarrow \vec{\tau}_A} \text{seq} \quad \frac{\Gamma \vdash_c M : \vec{\sigma}_A \Rightarrow \vec{v}_A}{\Gamma \vdash_c M : \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{\tau}_A \vec{v}_A} \text{exp} \\
\\
\frac{\Gamma \vdash_c M : \vec{\rho}_A \Rightarrow \vec{\sigma}_A \quad \Gamma \vdash_c N : \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{v}_A}{\Gamma \vdash_c M ; N : \vec{\rho}_A \vec{\tau}_A \Rightarrow \vec{v}_A} \text{lseq} \\
\\
\frac{\Gamma \vdash_c M : \vec{\rho}_A \Rightarrow \vec{\sigma}_A \vec{\tau}_A \quad \Gamma \vdash_c N : \vec{\tau}_A \Rightarrow \vec{v}_A}{\Gamma \vdash_c M ; N : \vec{\rho}_A \Rightarrow \vec{\sigma}_A \vec{v}_A} \text{rseq} \\
\\
\frac{\Gamma \vdash_v V : \rho \quad x : \rho, \Gamma \vdash_c M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A}{\Gamma \vdash_c \{V/x\}M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A} \text{ccut} \quad \frac{\Gamma \vdash_v V : \rho \quad x : \rho, \Gamma \vdash_c W : \tau}{\Gamma \vdash_v \{V/x\}W : \tau} \text{vcut} \\
\\
\frac{\Gamma, x : \pi, y : \rho, \Gamma' \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}}{\Gamma, y : \rho, x : \pi, \Gamma' \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}} \text{exch} \quad \frac{\Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}}{x : \rho, \Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}} \text{weak}
\end{array}$$

Figure 3.2: Admissible rules for the Functional Machine Calculus with Values

**Remark 3.2.6.** It can be proved that the *computation cut* (ccut), *value cut* (vcut) and *sequencing* rules are admissible by applying the definitions of substitution and sequencing to type derivations. Further, from this, preservation of types during reduction can be proved [9]. Additionally, it can be shown by a simple induction on type derivations that the usual structural rules of *exchange* and *weakening* are admissible in this system.

We will from now on assume that all terms mentioned are well-typed.

### 3.3 Variants

Finally, we recall the definition of the original FMC (that discussed in the introduction), without constants, which we will use in Chapter 6 for the proof of strong normalisation.

**Definition 3.3.1.** The *terms* of the (*classic*) *Functional Machine Calculus* (FMC), are given by the following grammar.

$$M, N, P ::= \star \mid x.M \mid [N]a.M \mid a\langle x \rangle.M$$

where the second term constructor is a *sequential variable*. Terms are considered modulo  $\alpha$ -equivalence. Capture-avoiding *sequencing*,  $N;M$  and *substitution* is defined similarly to Definition 3.1.2, except with the following cases instead of the cases for values or the sequential execution construct.

$$x.N;M = x.(N;M) \quad \{N/x\}x.M = N;\{N/x\}M$$

*Sequential contexts* are the obvious variation of Definition 3.1.5, and reductions are as in Definition 3.1.6, except without the force or thunk reductions. The abstract machine is as in Definition 3.1.4, except without the transition for terms  $?!M$ . Types are as in Definition 3.2.3, but with typing rules from Figure 1.5. We do not consider constants for this calculus. All notation is similar.

Note that there is an unambiguous translation between the two versions of the calculus without constants.

**Definition 3.3.2.** The *Sequential  $\lambda$ -calculus with values* (!SLC) is defined as the !FMC with one location,  $\lambda$ . In particular, it is generated over an !FMC-signature with one location,  $\Sigma_{\{\lambda\}}$ , which we call an !SLC-*signature* and denote simply  $\Sigma$ . We will freely confuse stack types and memory types generated over a single location, which are clearly isomorphic. Similarly, we can define the ‘classic’ sequential  $\lambda$ -calculus, SLC, analogously as the FMC with one location.

We can define a first-order version of the FMC as follows.

**Definition 3.3.3.** We can define the *Machine Calculus (MC)* as the FMC without higher-order value types:

$$M, N \triangleq \star \mid c.M \mid [x]a.M \mid a\langle x \rangle.M$$

In particular, it makes use of a first-order variant of a signature  $\Sigma_A$  generated over only *products* of (and not arrows between) base types. This is a first-order calculus in the sense that the only values are variables or value constants, and thus the argument of any application must be a variable or a constant. The only value types are base types  $\alpha$ , omitting higher-order types  $\vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , meaning a variable must be of base type and every computation (and sequential constant) must be an implication between stacks of base types. The type system for this calculus is then easily derived by restriction from that of the !FMC. Similarly, we can define a first-order version of the SLC.

This variant turns out to be a natural term language for string diagrams, as detailed in Chapters 4 and 5.

### 3.4 Machine Termination

We formalize the intuitive meaning of types as describing the initial and final stack of a run of the machine. This is an adaptation from the SLC to the !FMC of a proof from [9], which is due to Heijltjes. A similar proof also applies for the classic FMC. This proof is referred to later, in Chapter 6. It is also a good demonstration of the operational intuition the Functional Machine Calculus can bring to old results of the  $\lambda$ -calculus.

We consider here stacks to hold only closed terms, and work with the case of an empty computation signature, and a value signature which holds only base types. This is necessary, because the machine would otherwise halt on states  $(S_A, c.M)$  or  $(S_A, ?v.M)$ .

**Definition 3.4.1.** The set  $\text{RUN}(\vec{\sigma}_A \Rightarrow \vec{\tau}_A)$  is the set of !FMC computations  $M$  such that for any memory  $S_A \in \text{RUN}(\vec{\sigma}_A)$  there is a memory  $T_A \in \text{RUN}(\vec{\tau}_A)$  and a run of the machine

$$\frac{(S_A, M)}{(T_A, \star)}$$

where  $\text{RUN}(\vec{\tau}_A)$  is the set of memories  $T_A$  such that each  $T_a \in \text{RUN}(\vec{\tau}_a)$  and  $\text{RUN}(\rho_1 \dots \rho_n)$  is the set of stacks  $V_1 \dots V_n$  such that either  $\rho_i$  is a base type or else  $?V_i \in \text{RUN}(\rho_i)$ .<sup>2</sup>

---

<sup>2</sup>Note that  $\text{RUN}(\vec{\sigma}_A \Rightarrow \vec{\tau}_A)$  can denote the set of values (where  $\vec{\sigma}_A \Rightarrow \vec{\tau}_A$  denotes the type of a singleton stack), or the set of computations of type  $\vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , but which will be unambiguous from context.

We will show that  $M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  implies  $M \in \text{RUN}(\vec{\sigma}_A \Rightarrow \vec{\tau}_A)$ , thus proving termination. The proof is by a direct induction on type derivations, using the properties of machine runs, indicating that a type derivation constitutes a certificate of termination. The proof is analogous to a Tait's reducibility proof [111], with  $\text{RUN}(\vec{\sigma}_A \Rightarrow \vec{\tau}_A)$  taking the role of the reducibility set.

**Lemma 3.4.2.** *If  $\vec{w}: \vec{\omega} \vdash_c M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  then for any  $W \in \text{RUN}(\vec{\omega})$ ,  $\{W/\vec{w}\}M \in \text{RUN}(\vec{\sigma}_A \Rightarrow \vec{\tau}_A)$ .*

*Proof.* We proceed by induction on the type derivation of  $\vec{w}: \vec{\omega} \vdash_c M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ . In each case, let  $W$  be a stack in  $\text{RUN}(\vec{\omega})$  and let  $\sigma = \{W/\vec{w}\}$ .

- For the base case,

$$\vec{w}: \vec{\omega} \vdash_c M \equiv \star: \vec{\tau}_A \Rightarrow \vec{\tau}_A,$$

observe that  $\sigma\star = \star$  and that  $(S_A, \star)$  trivially terminates in zero steps, for any memory  $S_A: \vec{\tau}_A$ .

- For the abstraction case,

$$\vec{w}: \vec{\omega} \vdash_c M \equiv a\langle x \rangle. M': \rho\vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

the inductive hypothesis gives for any  $V \in \text{RUN}(\rho)$  and  $S_A \in \text{RUN}(\vec{\sigma}_A)$  a run on  $\{V/x\}\sigma M'$  as below, left, to some  $T_A \in \text{RUN}(\vec{\tau}_A)$ . We then construct the run for  $\sigma(a\langle x \rangle. M') = a\langle x \rangle. \sigma M'$  as below, right.

$$\frac{(\frac{S_A, \{V/x\}\sigma M'}{T_A}, \star)}{(\frac{S_A \setminus \{a\}; S_a \cdot V, a\langle x \rangle. \sigma M'}{S_A}, \{V/x\}\sigma M')} \quad \frac{(\frac{S_A \setminus \{a\}; S_a \cdot V, a\langle x \rangle. \sigma M'}{S_A}, \{V/x\}\sigma M')}{(\frac{S_A \setminus \{a\}; S_a \cdot V, a\langle x \rangle. \sigma M'}{T_A}, \star)}$$

- For the application case, with  $V: \rho$ ,

$$\vec{w}: \vec{\omega} \vdash_c M \equiv [V]a. M': \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

the inductive hypothesis gives that  $\sigma V \in \text{RUN}(\rho)$ , and for any  $S_A \in \text{RUN}(\vec{\sigma}_A)$  a run on  $\sigma M'$  as below, left, to some  $T_A \in \text{RUN}(\vec{\tau}_A)$ . We then construct the run for  $\sigma[V]a. M' = [\sigma V]a. \sigma M'$  as below, right.

$$\frac{(\frac{S_A \setminus \{a\}; S_a \cdot \sigma V, \sigma M'}{T_A}, \star)}{(\frac{S_A \setminus \{a\}; S_a \cdot \sigma V, \sigma M'}{T_A}, \star)} \quad \frac{(\frac{S_A \setminus \{a\}; S_a \cdot \sigma V, \sigma M'}{T_A}, \star)}{(\frac{S_A \setminus \{a\}; S_a \cdot \sigma V, [\sigma V]a. \sigma M'}{S_A}, \sigma M')}$$

- For the sequential execution case,

$$\vec{w}: \vec{\omega} \vdash_c M \equiv ?V. M': \vec{\rho}_A \vec{\tau}_A \Rightarrow \vec{v}_A,$$

Consider first that  $\vec{w}:\vec{\omega} \vdash_v V \equiv x$ . Take  $U \in \text{RUN}(\vec{\rho}_A \Rightarrow \vec{\sigma}_A)$ , so that for  $R_A \in \text{RUN}(\rho_A)$  there is a run on  $?U$  as below, left, to some  $S_A \in \text{RUN}(\sigma_A)$ . The inductive hypothesis then gives for  $T_A \in \text{RUN}(\vec{\tau}_A)$ , a run on  $\{U/x\}\sigma M'$  as below, right, to some  $U_A \in \text{RUN}(v_A)$ .

$$\frac{(\frac{R_A, ?U}{S_A, \star})}{(\frac{T_A S_A, \{U/x\}\sigma M'}{U_A, \star})}$$

We then construct the run for  $\{U/x\}\sigma ?x.M' = ?U; \{U/x\}\sigma M'$  by composing the two runs as follows, expanding the stack on the run for  $U$  by  $T$ .

$$\frac{(\frac{T_A R_A, ?U; \{U/x\}\sigma M'}{T_A S_A, \{U/x\}\sigma M'})}{(U_A, \star)}$$

Consider second that  $\vec{w}:\vec{\omega} \vdash_v V \equiv !N$ , so that the inductive hypothesis gives for  $R_A \in \text{RUN}(\rho_A)$  a run on  $\sigma N$  as below, left, to some  $S_A \in \text{RUN}(\sigma_A)$ . The inductive hypothesis also gives for  $T_A \in \text{RUN}(\vec{\tau}_A)$ , a run on  $\sigma M'$  as below, right, to some  $U_A \in \text{RUN}(v_A)$ .

$$\frac{(\frac{R_A, \sigma N}{S_A, \star})}{(\frac{T_A S_A, \sigma M'}{U_A, \star})}$$

We then construct the run for  $\sigma ?V.M' = ?!\sigma N.\sigma M'$  by composing the two runs as follows, expanding the stack on the run for  $?U$  by  $T_A$ .

$$\frac{(\frac{T_A R_A, ?!\sigma N.\sigma M'}{T_A R_A, \sigma N; \sigma M'})}{(\frac{T_A S_A, \sigma M'}{U_A, \star})}$$

□

**Theorem 3.4.3** (Termination). *For any closed, typed !FMC-term the machine terminates.*

*Proof.* Given some closed term  $\vdash_c M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , Lemma 3.4.2 tells us that  $M \in \text{RUN}(\vec{\sigma}_A \Rightarrow \vec{\tau}_A)$ . The same lemma gives us that any memory  $R_A: \vec{\rho}_A$  is such that  $R_A \in \text{RUN}(\vec{\sigma}_A)$  and thus that the machine terminates on  $(R_A, M)$ . □

## Chapter 4

# The Functional Machine Category

In this chapter, we define the Functional Machine Category,  $!FMC(\Sigma_A)$ , consisting of *closed* simply-typed  $!FMC$ -terms generated by a signature  $\Sigma_A$  and taken modulo an appropriate equational theory, with composition given by *sequencing*. A main contribution of this thesis is that this is *equivalent* to the familiar category of simply-typed  $\lambda$ -terms. In other words, the Functional Machine Category forms the *free* Cartesian closed category. Note that, as expected, the denotational perspective collapses all operational behaviour (*i.e.*, the distinction between stacks and the expliciting of sequencing), which are exactly the refinements made of the  $\lambda$ -calculus by the  $!FMC$ . In this chapter, we develop the necessary equational theory and show that the Functional Machine Category is a Cartesian closed category. In Chapter 5, the main result is proved. Note that the  $!FMC$  remains an *operational* refinement of the  $\lambda$ -calculus, even if it is not a *denotational* one.

Throughout these two chapters we will refer to the Functional Machine Calculus with Values as simply ‘the Functional Machine Calculus’; however we will continue to write  $!FMC$  in abbreviation and in formal statements to distinguish the two calculi. It is, however, an easy exercise to transfer results between the  $!FMC$  and the  $FMC$ .

The result stated above may be surprising, given the ability of the Functional Machine Calculus to encode effects, and requires some explanation. Most importantly, this is a semantics of the *calculus itself*, and *not* a semantics of effects. In particular, the result requires every location to be treated uniformly. So, for example, while we would encode random input in the  $!FMC$  using a read-only stream of probabilistically generated values (with no associated *push* action), for this result we must assume that every location is both read- and write-able. Similarly, we would encode state as a stack of depth (at most) one. However, we must further assume that every location can hold an arbitrary number of values. Imposing that a location is read-only, write-only, or has some maximum depth breaks the Cartesian closed semantics, as we will demonstrate in a later example.<sup>1</sup>

---

<sup>1</sup>Note, however, that the uniform treatment of locations corresponds to the uniform treatment



Given this consideration, there is precedent for calculi which can encode effects maintaining the semantics of a Cartesian closed category. For example, the encoding of monadic effects in simply-typed  $\lambda$ -calculus, where, for example, *state* encodes as the monad  $S \rightarrow (- \times S)$ . Here, a stateful process of type  $A \rightarrow B$  can be considered a pure function with the larger type  $A \rightarrow (S \rightarrow B \times S)$ . Something analogous happens with the !FMC: that is, a term which accesses a second stack will record this in its type.

Now we have justified why the main result is *plausible*, let us summarize the motivation *for* the result of Cartesian closure.

- We can define a natural notion of observational equivalence for typed !FMC-terms based on the stack machine, which we call *machine equivalence*, and prove the category of !FMC-terms modulo this equivalence forms a Cartesian closed category (although, not the free one). This result is proved at the end of this chapter. Note, we will see that it is the strength of the type system and of the corresponding notion of observation implicit in the definition of machine equivalence which allows us to achieve a Cartesian closed semantics here.<sup>2</sup>
- The result serves as a sanity check: neither the introduction of sequencing or locations introduces anything that is semantically unexpected.
- We preserve one of the most important semantic properties of the  $\lambda$ -calculus: its categorical semantics. In combination with the preservation of confluence and strong normalisation (the latter proved in Chapter 6), we argue that the Functional Machine Calculus is very well-behaved, despite its novelties.

The content of the first section is as follows. A coarser equational theory than is generated by the rewrites in Definition 3.1.6 is indeed necessary to show the category of !FMC-terms is Cartesian closed. This equational theory is first motivated using string diagrams, before being defined. Following the choice of equations, it is then proved that this equational theory is indeed sufficient to make the Functional Machine Category !FMC( $\Sigma_A$ ) Cartesian closed.

The second section shows the equational theory is validated by a natural notion of observational equivalence, which considers terms equivalent when the machine maps equivalent inputs to equivalent outputs. As a corollary, we achieve that the category of !FMC-terms modulo machine equivalence is Cartesian closed.

However, analogous to the case of the simply-typed  $\lambda$ -calculus, there are not enough typed contexts to separate all terms which “should” be distinct (at least, without introducing some constants such as natural numbers and addition), so machine equivalence necessarily turns out to be coarser than the equational theory. In

---

of effects with each other, and with the higher-order machinery of the  $\lambda$ -calculus, in the operational semantics of the !FMC. Hence it is the right assumption for modelling the *calculus*, if not for modelling *effects*.

<sup>2</sup>In particular, the assumption of uniformity of locations gives us more terms, and thus more contexts with which to distinguish between terms, than we might expect if we were attempting to give a semantic account of effects.

brief, by restricting inputs to the machine to be only well-typed terms, there are then too few possible inputs with which to ‘test’ adequately for distinguishability. Thus, we are justified in seeking a finer equivalence: namely, the aforementioned equational theory, which nevertheless captures more of our notion of observational equivalence than does the reduction relation presented previously.

## 4.1 String Diagrams

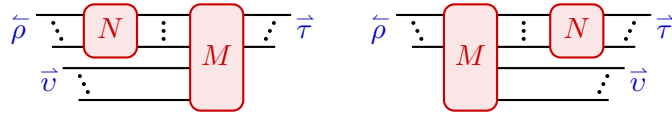
Before defining the full equational theory, we will motivate it step-by-step. We begin by considering the first-order calculus, and detail the first-order fragment of the equational theory using string diagrams. The equations specific to locations are then discussed. Finally, we leave string diagrams behind to discuss the higher-order fragment, along with the full definition. Note that the equational theory will subsume the reduction relation from Definition 3.1.6.

### 4.1.1 The First-Order Fragment

In motivating the first-order fragment, let us simplify matters for now, by considering the !FMC with just a single location  $\lambda$ , that is, the *Sequential  $\lambda$ -calculus with Values* (!SLC), returning to the string diagrammatic representation of terms with multiple locations afterwards.

We begin with the identity equation  $\langle x \rangle. [x] =_{\text{id}} \star$  mentioned previously: it says that popping a value from the stack and then pushing it back is the same as doing nothing.<sup>3</sup>

Next, recall from Figure 1.4 the representation of the sequencing of !SLC-terms as string diagrams. Left-sequencing and right-sequencing of computations is given below, for  $N: \vec{\rho} \Rightarrow \vec{\sigma}$  and  $M: \vec{\rho}\vec{v} \Rightarrow \vec{\tau}$ , and  $M: \vec{\rho} \Rightarrow \vec{v}\vec{\sigma}$  and  $N: \vec{\sigma} \Rightarrow \vec{\tau}$ , respectively.



Further, recall from Figure 1.4 the representations of *expansion* of a term  $M: \vec{\rho} \Rightarrow \vec{\sigma}$  by a stack  $\vec{\tau}$ , and of a term  $\langle \vec{x} \rangle. (M; [\vec{x}]): \vec{\tau}\vec{\rho} \Rightarrow \vec{\sigma}\vec{\tau}$ . This term pops the arguments for  $\vec{\tau}$  from the stack as the variables  $\vec{x}$ , evaluates  $M$ , and then returns the values bound to  $\vec{x}$  to the stack. In combination with sequencing, this allows us to understand

---

<sup>3</sup>This could have been presented as a rewrite, from left to right, and strong normalisation would still hold in its presence, but confluence would not. Analogous to the  $\lambda$ -calculus, one would expect instead to *expand* typed terms by this equation to retain confluence, but consideration of this is beyond the scope of the current work.

what will be the *interchange equation* ( $\iota$ ).

$$\begin{array}{ccc}
\begin{array}{c} \overleftarrow{\tau} \text{---} \boxed{N} \text{---} \langle y_1 \rangle \cdots \langle y_m \rangle \text{---} [y_1] \cdots [y_m] \text{---} \overrightarrow{v} \\ \overleftarrow{\rho} \text{---} \boxed{M} \text{---} \overrightarrow{\sigma} \end{array} & =_{\iota} & \begin{array}{c} \overleftarrow{\tau} \text{---} \langle x_1 \rangle \cdots \langle x_n \rangle \text{---} [x_1] \cdots [x_n] \text{---} \boxed{N} \text{---} \overrightarrow{v} \\ \overleftarrow{\rho} \text{---} \boxed{M} \text{---} \overrightarrow{\sigma} \end{array} \\
N ; \langle \overleftarrow{y} \rangle . (M ; \overrightarrow{y}) & =_{\iota} & \langle \overleftarrow{x} \rangle . (M ; \overrightarrow{x}) ; N
\end{array}$$

Note, the action of the *product* on terms,  $M \times N$ , in the category of closed !FMC-terms will be given as above (in either form – it does not matter). As expected from the diagrams, the interchange equation thus axiomatizes the *bifunctionality* of the product. In the setting of the FMC, with its multiple locations, this equation becomes

$$N ; \langle \overleftarrow{y}_A \rangle . (M ; \overrightarrow{y}_A) =_{\iota} \langle \overleftarrow{x}_A \rangle . (M ; \overrightarrow{x}_A) ; N .$$

**Remark 4.1.1.** Without the interchange equation, and in particular if we take the equational theory to be generated by *just*  $\{\text{id}, \beta, \phi, \pi\}$ , the category of terms would only be *pre-monoidal*. Indeed, this is a category with the axioms of a monoidal category, except where bifunctionality of the tensor product fails in general. Pre-monoidal categories are discussed later, in the chapter on related literature (Chapter 7).

Consider now that we can *duplicate* and *delete* elements from the top of the stack with the first two terms illustrated below, respectively. The diagram corresponding to the term swapping the top two elements of the stack is also given below, right.

$$\begin{array}{ccc}
\begin{array}{c} \tau_1 \text{---} \langle x_1 \rangle \cdots [x_1] \text{---} \tau_1 \\ \tau_n \text{---} \langle x_n \rangle \cdots [x_n] \text{---} \tau_n \end{array} & \begin{array}{c} \tau_1 \text{---} \langle x_1 \rangle \\ \tau_n \text{---} \langle x_n \rangle \end{array} & \begin{array}{c} \overleftarrow{\sigma} \text{---} \langle \overleftarrow{x} \rangle \cdots [y] \text{---} \overrightarrow{\tau} \\ \overleftarrow{\tau} \text{---} \langle \overleftarrow{y} \rangle \cdots [x] \text{---} \overrightarrow{\sigma} \end{array} \\
\langle \overleftarrow{x} \rangle . [\overrightarrow{x}] . [\overrightarrow{x}] : \overrightarrow{\sigma} \Rightarrow \overrightarrow{\sigma} \overrightarrow{\sigma} & \langle \overleftarrow{x} \rangle : \overrightarrow{\sigma} \Rightarrow & \langle \overleftarrow{x} \rangle . \langle \overleftarrow{y} \rangle . [\overrightarrow{x}] . [\overrightarrow{y}] : \overrightarrow{\sigma} \overrightarrow{\tau} \Rightarrow \overrightarrow{\sigma} \overrightarrow{\tau}
\end{array}$$

The following *diagonal* ( $\Delta$ ) and *terminal* (!) equations axiomatize naturality of these operations in the FMC; that is, evaluating a computation and duplicating its output is the same as evaluating the term twice (in each case, given one of its duplicated inputs), and evaluating a term and discarding its output is the same as discarding its input and never evaluating the term.

$$\begin{aligned}
M ; \langle \overleftarrow{x}_A \rangle . [\overrightarrow{x}_A] . [\overrightarrow{x}_A] &=_{\Delta} \langle \overleftarrow{w}_A \rangle . [\overrightarrow{w}_A] . M . [\overrightarrow{w}_A] . M \\
M ; \langle \overleftarrow{x}_A \rangle &=_{!} \langle \overleftarrow{w}_A \rangle
\end{aligned}$$

Note that naturality of symmetry is implicit in our  $\Delta$  equation, and we detail this in a later remark

Such equations seem well-motivated for terms without side-effects, as in the !SLC, but may be surprising in the setting of the !FMC, with its ability to encode effects. Recall that we are not trying to give a semantics of effects here, but rather trying to capture a notion of observational equivalence for the calculus, which we have delayed introduction of.

### 4.1.2 The Locational Fragment

The locational fragment of the equational theory consists of the *relocation* and the *permutation* equations. The first axiomatizes the interaction of relocation isomorphisms with each other. Note, this is indeed subsumed by the beta rule, but we include it here for reasons that will become clear shortly.

$$a\langle x \rangle. [x]b. b\langle y \rangle. [y]c =_{\rho} a\langle x \rangle. [x]c$$

This equation implies that each relocation term  $a\langle x \rangle. b[x]$  is indeed an isomorphism, with inverse  $b\langle x \rangle. [x]a$ :

$$a\langle x \rangle. [x]b. b\langle y \rangle. [y]a =_{\rho} a\langle x \rangle. [x]a =_{\text{id}} \star.$$

The role of the permutation equation, in contrast, is to *collapse symmetries* on types at different locations into *identities*. The following *permutation lemma* shows that the equations generated from the identity and  $\{\beta, \pi\}$  reductions suffices for permutation of applications and abstractions on different locations. We will refer to the equations  $\pi'$  given in the following lemma as the *permutation equivalence*.

**Lemma 4.1.2** (Permutation). *The following two equations are derivable:*

$$[V]a. [W]b =_{\pi'} [W]b. [V]a \quad a\langle x \rangle. b\langle y \rangle. M =_{\pi'} b\langle y \rangle. a\langle x \rangle. M.$$

*Proof.*

$$\begin{aligned} [V]a. [W]b &=_{\text{id}} [V]a. \underline{[W]b. a\langle x \rangle. [x]a} \\ &=_{\pi} \underline{[V]a. a\langle x \rangle. [W]b. [x]a} \\ &=_{\beta} [W]b. [V]a \\ a\langle x \rangle. b\langle y \rangle. M &=_{\text{id}} b\langle z \rangle. \underline{[z]b. a\langle x \rangle. b\langle y \rangle. M} \\ &=_{\pi} b\langle z \rangle. a\langle x \rangle. \underline{[z]b. b\langle y \rangle. M} \\ &=_{\beta} b\langle z \rangle. a\langle x \rangle. \{z/y\}M \\ &=_{\alpha} b\langle y \rangle. a\langle x \rangle. M \end{aligned} \quad \square$$

**Remark 4.1.3.** The identity and symmetry at type  $a(\vec{\sigma})b(\vec{\tau})$  coincide. First note, this is possible because, at the level of types, we have

$$a(\vec{\sigma})b(\vec{\tau}) \Rightarrow a(\vec{\sigma})b(\vec{\tau}) = a(\vec{\sigma})b(\vec{\tau}) \Rightarrow b(\vec{\tau})a(\vec{\sigma}).$$

This is a result of the permutation equivalence, as follows.

$$a\langle \vec{x} \rangle. b\langle \vec{y} \rangle. [\vec{x}]a. [\vec{y}]b =_{\pi} a\langle \vec{x} \rangle. [\vec{x}]a. b\langle \vec{y} \rangle. [\vec{y}]b =_{\text{id}} \star$$

We can represent terms of the !FMC (as opposed to the !SLC) using (a variant of) *nominal* string diagrams [5]. Whereas typical string diagrams can be used to

represent actions on type *vectors*, we can use nominal diagrams to represent actions on type *families*, where the order between values in different families is no longer a consideration (due to the above remark). There are technical concerns to be considered when dealing with these diagrams (especially, composition of diagrams), but we elide a detailed discussion and instead refer the reader to the aforementioned citation. Given this, we make use of these diagrams only informally.

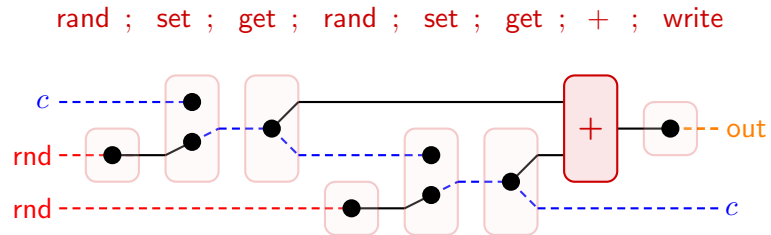
Such diagrams associate with each wire a label (its location) and include a *re-naming* generator (corresponding to the relocation isomorphism), which allows the changing of labels. The interaction of two such generators corresponds to our *relocation*  $\rho$  equation, pictured below, left. We use colour to distinguish the wires with different labels (*i.e.*, at different locations).

*nominal generator*

Assuming that we have at most one value held at each location, we can envisage these diagrams as being three-dimensional instead of planar, so that symmetries acting on wires become true identities, as above, right. In the general case (and ignoring relocation) since wires on a single location are still ordered, we can imagine  $n$  planes, one for each location, where symmetry of two wires on a given plane remains a true symmetry.

We now reprise the example from the introduction, which uses such diagrams, aiming to illustrate why we would expect the !FMC to form a Cartesian (closed) category, and how *uniform* treatment of locations is crucial to achieving a CCC semantics. This assumption of uniformity corresponds to our attempt to model the calculus itself, and *not* computational effects.

**Example 4.1.4.** Consider the following diagram representing the term given below, recalling the definition of the combinators from Figure 1.1. Dashed lines here simply denote that a wire is on a non-main location.



Due to the type system, we see all the dependencies between operations. For example, the second call to **rand** may safely be made before the first calls to **set** and **get**. This would be illustrated by ‘sliding’ the **rand** operation along the wire. We can also see that the second **set** is dependent on the first **get**: the value pushed to **c** by the first **get** is discarded by the second **set**. Indeed, the sequential composition of the terms **set** and **get** creates such a beta-redex, corresponding to the expected interaction of ! with  $\Delta$  in a Cartesian category. Indeed, modulo renaming of wires,

these effectful operations are encoded by the diagonal and terminal operations of a Cartesian category.

Note, one can see in the above example that applying the naturality of the diagonal would result in a duplication on the location **rnd**. This relies on **rnd** being a location with no special status, and in particular, having an associated *push* action, similar to the main location **λ**. If we were to enforce that **rnd** was a read-only stream, then this duplication would no longer be possible and the semantics can no longer be Cartesian. Similar issues arise for memory cells, which ought to have depth (at most) one. We leave consideration of the particular properties of encoded effects for future work.

## 4.2 The Equational Theory

We have seen in the previous section the two terms which allow for duplication and deletion. For the higher-order calculus, we also expect the force and thunk equations, corresponding to the force and thunk reduction rules

$$?!N. M =_{\phi} N ; M \quad !?V =_{\tau} V$$

We delay further discussion of these equations until the next chapter. We will discuss the remaining two: *beta* and *eta*.

There are two terms which are central to making the calculus higher-order, shown below.

$$\langle f \rangle . ?f : (\vec{\sigma}_A \Rightarrow \vec{\tau}_A) \vec{\sigma}_A \Rightarrow \vec{\tau}_A \quad \langle \vec{x}_A \rangle . [![\vec{x}_A]. M] : \vec{\sigma}_A \Rightarrow (\vec{\rho}_A \Rightarrow \vec{\tau}_A)$$

The first term allows the execution of a thunk held on the stack. The second term gives us the *currying* of  $M : \vec{\sigma}_A \vec{\rho}_A \Rightarrow \vec{\tau}_A$ , by its partial application and thunking. It will be evident that the beta and eta equations detailed below relate to these terms.

Remarkably, the beta law can be reduced to a more *local* form: that is, we do not require full generality of  $[V]. \langle x \rangle . M =_{\beta} \{V/x\}M$ , but rather we can make do with the more restricted form below.

$$[V]. \langle x \rangle . ?x =_{\beta} ?V$$

After presenting the full equational theory, it will be proved that the general, global beta law is in fact derivable. The idea behind the proof of this is to propagate an “explicit substitution”  $[V]. \langle x \rangle . M$  along  $M$  by applying the naturalities (and interchange) discussed in the first-order fragment. Duplication and deletion are handled solely by  $\Delta$  and  $!$ , with the interchange rule also playing an essential role in the propagation. When the redex meets a variable, the *local* beta rule above can be applied, “unboxing” the value. However, there must also be an analogous rule for pushing the redex *into* boxes (applications). This is the eta rule, given below.

$$S ; \langle \vec{x} \rangle . [![\vec{x}_A]. M] =_{\eta} [!S ; M]$$

In other words, the global beta reduction rule is internalized in the calculus by the given set of naturalities.

We now introduce the equational theory we will work with. Recall that we fix a set of location  $A = \{a_1, \dots, a_n\}$  and we take  $a, b$  to range over all locations.

Note, some of these rules have been simplified from the form they were presented in above, but this is simply by an application of the identity law which is explained subsequently. Note, further, that the equational theory implicitly assumes each location is treated uniformly, as mentioned in the introduction of this chapter.

**Definition 4.2.1.** We define the *equational theory*  $=_{\text{eqn}}$  of the !FMC to be the least equivalence generated by the following laws, closed under all contexts:

Identity:	$\langle \vec{x}_A \rangle. [\vec{x}_A] =_{\text{id}} \star$	$\vec{\sigma}_A \Rightarrow \vec{\sigma}_A$
Local Beta:	$[V]a. a\langle f \rangle. ?f =_{\beta} ?V$	$\vec{\sigma}_A \Rightarrow \vec{\tau}_A$
Force:	$?!M =_{\phi} M$	$\vec{\sigma}_A \Rightarrow \vec{\tau}_A$
Thunk:	$!?V =_{\tau} V$	$\vec{\sigma}_A \Rightarrow \vec{\tau}_A$
Eta:	$S; \langle \vec{x}_A \rangle. [! [x_A]. N]a =_{\eta} [!S; N]a$	$\Rightarrow a(\vec{\tau}_A \Rightarrow \vec{v}_A)$
Diagonal:	$S; \langle \vec{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A] =_{\Delta} S; S$	$\Rightarrow \vec{\sigma}_A \vec{\sigma}_A$
Terminal:	$S; \langle \vec{x}_A \rangle =_{!} \star$	$\Rightarrow$
Interchange:	$S; \langle \vec{x}_A \rangle. (P; [\vec{x}_A]) =_{\iota} P; S$	$\vec{\pi}_A \Rightarrow \vec{\rho}_A \vec{\sigma}_A$
Relocation:	$[V]a. a\langle y \rangle. [y]b =_{\rho} [V]b$	$\Rightarrow b(\tau)$
Permutation:	$[V]b. a\langle x \rangle. M =_{\pi} a\langle x \rangle. [V]b. M$	$a(\rho)\vec{\sigma}_A \Rightarrow \vec{\tau}_A$

where, for eta,  $\text{fv}(\vec{x}_A) \cap \text{fv}(N) = \emptyset$ , for interchange,  $\text{fv}(\vec{x}_A) \cap \text{fv}(P) = \emptyset$  and for permutation,  $x \notin \text{fv}(V)$  and  $a \neq b$ , for relocation  $a \neq b$ , and with  $x: \vec{\sigma}_A, y: \tau, f: \vec{\sigma}_A \Rightarrow \vec{\tau}_A, M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A, N: \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{v}_A, S: \Rightarrow \vec{\sigma}_A, P: \vec{\pi}_A \Rightarrow \vec{\rho}_A$  throughout. We will often write simply  $=$  for  $=_{\text{eqn}}$ , and *e.g.*  $=_{\beta}$  for the least equivalence generated by  $=_{\beta}$  closed under all contexts, and similarly for the other equations.

**Remark 4.2.2.** Note, we could restrict identity to apply to a singleton type and recover the original by iteration. By using the identity law, we could equivalently state the interchange equation in a more *symmetric* manner, as follows:

$$Q; \langle \vec{x}_A \rangle. (P; [\vec{x}_A]) =_{\iota} \langle \vec{w}_A \rangle. (P; [\vec{w}_A]. Q),$$

with  $P: \vec{\pi}_A \Rightarrow \vec{\rho}_A, Q: \vec{\sigma}_A \Rightarrow \vec{\tau}_A, \vec{w}_A: \sigma_A$  and  $\vec{x}_A: \tau_A$ .<sup>4</sup> Similarly, we can apply the identity to terminality to recover

$$\begin{aligned} M; \langle \vec{x}_A \rangle. [! [\vec{x}_A]. N] &=_{\eta} \langle \vec{w}_A \rangle. [! [\vec{w}_A]. M; N] \\ M; \langle \vec{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A] &=_{\Delta} \langle \vec{w}_A \rangle. [\vec{w}_A]. M. [\vec{w}_A]. M \\ M; \langle \vec{x}_A \rangle &=_{!} \langle \vec{w}_A \rangle \end{aligned}$$

for  $M: \vec{\rho}_A \Rightarrow \vec{\sigma}_A$ . We will sometimes refer to the above equations as just  $\eta, !, \Delta$  or  $\iota$ .

<sup>4</sup>This is derived as  $Q; \langle \vec{x}_A \rangle. (P; [\vec{x}_A]) =_{\text{id}} \langle \vec{y}_A \rangle. [\vec{y}_A]. Q; \langle \vec{x}_A \rangle. (P; [\vec{x}_A]) =_{\iota} \langle \vec{y}_A \rangle. (P; [\vec{y}_A]. Q)$ .

It is now proved that the *global* beta law is derivable from the given equational theory. We then proceed to show that the equational theory above is sufficient to make the !FMC category Cartesian closed.

**Proposition 4.2.3** (Soundness of beta). *The global beta equation (below) is derivable in  $=_{eqn}$ .*

$$[V]a. a\langle x \rangle. M =_{\beta} \{V/x\}M$$

*Proof.* We prove a slightly stronger statement of the proposition, in conjunction with two appropriate statements on values. For all memories  $S_A$ , and all locations  $a$ , we claim the following are derivable:

$$\begin{aligned} [S_A]. \langle \vec{x}_A \rangle. M &= \{S_A/\vec{x}_A\}M \\ [S_A]. \langle \vec{x}_A \rangle. [V]a &=_1 [\{S_A/\vec{x}_A\}V]a \\ [S_A]. \langle \vec{x}_A \rangle. ?V &=_2 ?\{S_A/\vec{x}_A\}V. \end{aligned}$$

Proceed by (mutual) induction on the type derivation of  $M$  and  $V$ . Let  $M' = \{\vec{x}'_A/\vec{x}_A\}M$  so that in each case of the induction, we have  $M = [\vec{x}_A]. \langle \vec{x}'_A \rangle. M'$ . Similarly, let  $V'' = \{x''/x\}V$  so that in each case of the induction, we have  $[V] = [\vec{x}'_A]. \langle \vec{x}_A \rangle. [V'']$  and  $?V = [\vec{x}'_A]. \langle \vec{x}_A \rangle. ?V''$ .

- Identity:

$$\begin{aligned} [S_A]. \langle \vec{x}_A \rangle. \star &= ! \star \\ (\text{defn. subst.}) &= \{S_A/\vec{x}_A\}\star \end{aligned}$$

- Application:

$$\begin{aligned} [S_A]. \langle \vec{x}_A \rangle. [V]a. M \text{ (I.H.)} &= [S_A]. \langle \vec{x}_A \rangle. [\vec{x}_A]. \langle \vec{x}''_A \rangle. [V'']a. [\vec{x}_A]. \langle \vec{x}'_A \rangle. M' \\ &=_{\iota} [S_A]. \langle \vec{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A]. \langle \vec{y}_A \rangle. \langle \vec{x}''_A \rangle. [V'']a. [\vec{y}_A]. \langle \vec{x}'_A \rangle. M' \\ &=_{\Delta} [S_A]. [S_A]. \langle \vec{y}_A \rangle. \langle \vec{x}''_A \rangle. [V'']a. [\vec{y}_A]. \langle \vec{x}'_A \rangle. M' \\ &=_{\iota} [S_A]. \langle \vec{x}''_A \rangle. [V'']a. [S_A]. \langle \vec{x}'_A \rangle. M' \\ \text{(I.H.)} &= [\{S_A/\vec{x}_A\}V]. \{S_A/\vec{x}_A\}M \\ (\text{defn. subst.}) &= \{S_A/\vec{x}_A\}[V]. M \end{aligned}$$

- Abstraction:

$$\begin{aligned} [S_A]. \langle \vec{x}_A \rangle. a\langle y \rangle. M &=_{\text{id}} a\langle y' \rangle. [y']a. [S_A]. \langle \vec{x}_A \rangle. a\langle y \rangle. M \\ \text{(I.H.)} &= a\langle y' \rangle. \{a(y')S_A/a(y)\vec{x}_A\}M \\ &=_{\alpha} a\langle y \rangle. \{S_A/\vec{x}_A\}M \\ (\text{defn. subst.}) &= \{S_A/\vec{x}_A\}a\langle y \rangle. M \end{aligned}$$



- Sequential Execution:

$$\begin{aligned}
[S_A]. \langle \vec{x}_A \rangle. ?V. M \text{ (I.H.)} &= [S_A]. \langle \vec{x}_A \rangle. \underline{[\vec{x}_A]. \langle \vec{x}'_A \rangle. ?V''. [\vec{x}_A]. \langle \vec{x}'_A \rangle. M'} \\
&=_{\iota} [S_A]. \langle \vec{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A]. \langle \vec{y}_A \rangle. \langle \vec{x}'_A \rangle. \vec{V}'''. [\vec{y}_A]. \langle \vec{x}'_A \rangle. M' \\
&=_{\Delta} [S_A]. \underline{[S_A]. \langle \vec{y}_A \rangle. \langle \vec{x}'_A \rangle. ?V. [\vec{y}_A]. \langle \vec{x}'_A \rangle. M'} \\
&=_{\iota} [S_A]. \langle \vec{x}'_A \rangle. ?V'''. [S_A]. \langle \vec{x}'_A \rangle. M' \\
\text{(I.H.)} &= ?\{\vec{S}_A/\vec{x}\}V. \{S_A/\vec{x}_A\}M \\
\text{(defn. subst.)} &= \{S_A/\vec{x}\} ?V. M
\end{aligned}$$

- Variable ( $y \in \vec{x}_A$ ): let  $V$  be the element of  $S_A$  corresponding to  $y$ , and let  $[S_A] = [S'_A]. [V]b. [S''_A]$  and  $\langle \vec{x}_A \rangle. M = \langle \vec{x}'_A \rangle. b\langle y \rangle. \langle \vec{x}_A \rangle. M$ .

$=_1$ :

$$\begin{aligned}
[S_A]. \langle \vec{x}_A \rangle. [y]a &=_{\rho} [S_A]. \langle \vec{x}_A \rangle. [y]b. b\langle z \rangle. [z]a \\
&=_{!} [S'_A]. [V]b. b\langle y \rangle. \langle \vec{x}'_A \rangle. [y]b. b\langle z \rangle. [z]a \\
&=_{\iota} [S'_A]. \langle \vec{x}'_A \rangle. [V]b. b\langle z \rangle. [z]a \\
&=_{!} [V]b. b\langle y \rangle. [y]a \\
&=_{\rho} [V]a \\
\text{(defn. subst.)} &= [\{S_A/\vec{x}_A\}y]a
\end{aligned}$$

$=_2$ :

$$\begin{aligned}
[S_A]. \langle \vec{x}_A \rangle. ?y &=_{\beta} [S_A]. \langle \vec{x}_A \rangle. [y]b. b\langle z \rangle. ?z \\
&=_{\beta} [S'_A]. [V]b. b\langle y \rangle. \langle \vec{x}'_A \rangle. [y]b. b\langle z \rangle. ?z \\
&=_{\iota} [S'_A]. \langle \vec{x}'_A \rangle. [V]b. b\langle z \rangle. ?z \\
&=_{!} [V]b. b\langle z \rangle. ?z \\
&=_{\beta} ?V \\
\text{(defn. subst.)} &= ?\{S_A/\vec{x}_A\}y
\end{aligned}$$

- Variable ( $y \notin \vec{x}_A$ ):

$=_1$ :

$$\begin{aligned}
[S_A]. \langle \vec{x}_A \rangle. [y]b &=_{!} [y]b \\
\text{(defn. subst.)} &= [\{S_A/\vec{x}_A\}y]b
\end{aligned}$$

$=_2$ :

$$\begin{aligned}
[S_A]. \langle \vec{x}_A \rangle. ?y &=_{!} ?y \\
\text{(defn. subst.)} &= ?\{S_A/\vec{x}_A\}y
\end{aligned}$$

- Thunk:

$=_1:$

$$\begin{aligned}
[S_A]. \langle \vec{x}_A \rangle. [!M]b \text{ (I.H.)} &= [S_A]. \langle \vec{x}_A \rangle. [![\vec{x}_A]. \langle \vec{x}'_A \rangle. M']b \\
&=_{\eta} [![S_A]. \langle \vec{x}'_A \rangle. M']b \\
\text{(I.H.)} &= [!\{S_A/\vec{x}_A\}M]b \\
\text{(defn. subst.)} &= [\{S_A/\vec{x}_A\}!M]b
\end{aligned}$$

$=_2:$

$$\begin{aligned}
[S_A]. \langle \vec{x}_A \rangle. ?!M &=_{\phi} [S_A]. \langle \vec{x}_A \rangle. M \\
\text{(I.H.)} &= \{S_A/\vec{x}_A\}M \\
&=_{\phi} ?!\{S_A/\vec{x}_A\}M \\
\text{(defn. subst.)} &=_{\phi} ?\{S_A/\vec{x}_A\}!M \quad \square
\end{aligned}$$

We can now make use of this derived rule freely in our later calculations, and will henceforth use  $=_{\beta}$  to refer to non-local beta equation, as usual. Technically, the admissibility of the global beta equation will allow us to bypass the usual work of showing that our translations between calculi (the STLC, !SLC and !FMC) respect substitution.

**Remark 4.2.4.** Naturality of symmetry is implicit in the  $\Delta$  equation. To see this, observe how  $\Delta$  swaps the order of the central occurrences of  $T$  and  $S$  in the right-hand side of the following equation.

$$T; S; \langle \vec{x}_A \rangle. \langle \vec{y}_A \rangle. [\vec{y}_A]. [\vec{x}_A]. [\vec{y}_A]. [\vec{x}_A] =_{\Delta} T; S; T; S$$

More precisely, we can calculate the following.

$$\begin{aligned}
&T; S; \langle \vec{x}_A \vec{y}_A \rangle. [\vec{x}_A \vec{y}_A] \\
&=_{\beta} T; S; \langle \vec{x}_A \vec{y}_A \rangle. [\vec{y}_A \vec{x}_A \vec{y}_A \vec{x}_A]. \langle \vec{x}'_A \vec{y}'_A \vec{x}'_A \vec{y}'_A \rangle. [\vec{x}'_A \vec{y}'_A] \\
&=_{\Delta} T; S; T; S; \langle \vec{x}'_A \vec{y}'_A \vec{x}'_A \vec{y}'_A \rangle. [\vec{x}'_A \vec{y}'_A] \\
&=_{!} T; S; T; \langle \vec{y}'_A \vec{x}'_A \vec{y}'_A \rangle. [\vec{x}'_A \vec{y}'_A] \\
&=_{\iota} T; \langle \vec{y}'_A \rangle. S; T \\
&=_{!} S; T
\end{aligned}$$

For this reason, in the case of the !FMC with a *linear* variable policy, where we would drop the  $\Delta$  and  $!$  equations, we would have to add an equation axiomatizing the naturality of *symmetry*:

$$S. \langle \vec{x}_A \rangle. \langle \vec{w}_A \rangle. [\vec{x}_A]. [\vec{w}_A] =_{\sigma} \langle \vec{w}_A \rangle. S. [\vec{w}_A]: \vec{\rho}_A \Rightarrow \vec{\sigma}_A \vec{\rho}_A,$$

where  $w_A: \vec{\rho}_A, x_A: \vec{\sigma}_A$ . This would be natural for achieving a correspondence between symmetric monoidal (closed) categories and the !FMC.

### 4.3 The Functional Machine Category is Cartesian Closed

Similar to our definition of the category of  $\lambda$ -calculus-terms, we will overspecify the equipment of the CCC. However, it is easy to verify that all the equipment given is consistent in the expected way.

**Definition 4.3.1.** Given an !FMC signature  $\Sigma_A = (\Sigma_0, \Sigma_c, \Sigma_v)$ , the *Functional Machine Category* of *closed* !FMC-terms  $!FMC(\Sigma_A)$  is defined with

- Objects: families of type vectors  $\vec{\tau}_A = \vec{\tau}_A$  over  $\Sigma_0$ ,
- Morphisms:  $Hom(\vec{\sigma}_A, \vec{\tau}_A)$  is given by the set of closed typed !FMC computations  $M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  over  $\Sigma_A$ , modulo  $=_{eqn}$ ,
- Composition: given morphisms  $M \in Hom(\vec{\rho}_A, \vec{\sigma}_A)$  and  $N \in Hom(\vec{\sigma}_A, \vec{\tau}_A)$ ,  $M; N \in Hom(\vec{\rho}_A, \vec{\tau}_A)$  is given by sequencing  $M; N : \vec{\rho}_A \Rightarrow \vec{\tau}_A$ ,
- Identity: given on every type  $\vec{\tau}_A$  by the term  $\star : \vec{\tau}_A \Rightarrow \vec{\tau}_A$ .
- Products: given on objects by by concatenation:  $\vec{\sigma}_A \times \vec{\tau}_A = \vec{\sigma}_A \vec{\tau}_A$ , with the unit object given by  $\epsilon_A$ . The action on morphisms and the associated canonical natural transformations, *terminal*  $!$ , the *diagonal*  $\Delta$ , *pairing*  $\langle P, Q \rangle$ , the *projections*  $\pi_1$  and  $\pi_2$  and *symmetry*  $\text{sym}$ , are respectively:

$$\begin{aligned}
M \times N : \vec{\sigma}_A \times \vec{\rho}_A &\longrightarrow \vec{\tau}_A \times \vec{v}_A &= \langle \vec{w}_A \rangle. P. [\vec{w}_A]. Q : \vec{\rho}_A \vec{\sigma}_A \Rightarrow \vec{\tau}_A \vec{v}_A, \\
! : \vec{\tau}_A &\longrightarrow 1 &= \langle \vec{x}_A \rangle : \vec{\sigma}_A \Rightarrow \\
\Delta : \vec{\sigma}_A &\longrightarrow \vec{\sigma}_A \times \vec{\sigma}_A &= \langle \vec{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A] : \vec{\sigma}_A \Rightarrow \vec{\sigma}_A \vec{\sigma}_A \\
\langle P, Q \rangle : \vec{\sigma}_A &\longrightarrow \vec{\tau}_A \times \vec{v}_A &= \langle \vec{x}_A \rangle. ([\vec{x}_A]. M; [\vec{x}_A]. N) : \vec{\sigma}_A \Rightarrow \vec{\tau}_A \vec{v}_A \\
\pi_1 : \vec{\tau}_A \times \vec{\sigma}_A &\longrightarrow \vec{\sigma}_A &= \langle \vec{x}_A \rangle : \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{\tau}_A \\
\pi_2 : \vec{\tau}_A \times \vec{\sigma}_A &\longrightarrow \vec{\tau}_A &= \langle \vec{x}_A \rangle. \langle \vec{y}_A \rangle. [\vec{x}_A] : \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{\sigma}_A \\
\text{sym} : \vec{\tau}_A \times \vec{\sigma}_A &\longrightarrow \vec{\sigma}_A \times \vec{\tau}_A &= \langle \vec{x}_A \rangle. \langle \vec{y}_A \rangle. [\vec{x}_A]. [\vec{y}_A] : \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{\sigma}_A \vec{\tau}_A
\end{aligned}$$

where  $\vec{w}_A : \vec{\rho}_A, \vec{x}_A : \vec{\sigma}_A, \vec{y}_A : \vec{\tau}_A, M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A, N : \vec{\sigma}_A \Rightarrow \vec{v}_A, P : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  and  $Q : \vec{\rho}_A \Rightarrow \vec{v}_A$ ,

- Exponential: given on objects by  $\vec{\sigma}_A \rightarrow \vec{\tau}_A = \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ . The action on morphisms and associated canonical transformations, *evaluation*  $\epsilon$ , *expansion*  $\eta$  and *currying*  $\text{curry}$ , are respectively:

$$\begin{aligned}
\epsilon : \vec{\sigma}_A \times (\vec{\sigma}_A \rightarrow \vec{\tau}_A) &\longrightarrow \vec{\tau}_A &= \langle f \rangle. ?f : (\vec{\sigma}_A \Rightarrow \vec{\tau}_A) \vec{\sigma}_A \Rightarrow \vec{\tau}_A \\
\eta : \vec{\sigma}_A &\longrightarrow (\vec{\tau}_A \rightarrow \vec{\tau}_A \times \vec{\sigma}_A) &= \langle \vec{x}_A \rangle. [! [\vec{x}_A]] : \vec{\sigma}_A \Rightarrow (\vec{\tau}_A \Rightarrow \vec{\tau}_A \vec{\sigma}_A) \\
\text{curry}(M) : \vec{\sigma}_A &\longrightarrow (\vec{\rho}_A \longrightarrow \vec{\tau}_A) &= \langle \vec{x}_A \rangle. [! [\vec{x}_A]. M] : \vec{\sigma}_A \Rightarrow (\vec{\rho}_A \Rightarrow \vec{\tau}_A) \\
P \rightarrow Q : (\vec{\sigma}_A \rightarrow \vec{\tau}_A) &\longrightarrow (\vec{\rho}_A \rightarrow \vec{v}_A) &= \langle f \rangle. [! P; ?f. Q] : (\vec{\sigma}_A \Rightarrow \vec{\tau}_A) \Rightarrow (\vec{\rho}_A \Rightarrow \vec{v}_A)
\end{aligned}$$

where  $f : \vec{\sigma}_A \Rightarrow \vec{\tau}_A, \vec{x}_A : \vec{\sigma}_A$  and  $M : \vec{\sigma}_A \vec{\rho}_A \Rightarrow \vec{\tau}_A, P : \vec{\rho}_A \Rightarrow \vec{\sigma}_A$  and  $Q : \vec{\tau}_A \Rightarrow \vec{v}_A$ ,

- the associators and unitors for each object are given by  $\star$ .

We have that sequencing is associative and that  $\star$  is its identity, so we can indeed expect this to form a category.

**Remark 4.3.2.** We have defined exponentials relative to the *main* location  $\lambda$  – but there is nothing *special* about this location. Indeed, we can equivalently define an exponential  $\vec{\sigma}_A \xrightarrow{a} \vec{\tau}_A$  on *any* location  $a$  as  $a(\vec{\sigma}_A \Rightarrow \vec{\tau}_A)$ , with the following equipment

$$\begin{aligned} \epsilon^a : \vec{\sigma}_A \times a(\vec{\sigma}_A \rightarrow \vec{\tau}_A) &\longrightarrow \vec{\tau}_A &= a\langle f \rangle. ?f : a(\vec{\sigma}_A \Rightarrow \vec{\tau}_A) \vec{\sigma}_A \Rightarrow \vec{\tau}_A \\ \eta^a : \vec{\sigma}_A &\longrightarrow a(\vec{\tau}_A \rightarrow \vec{\sigma}_A \times \vec{\sigma}_A) &= \langle \vec{x}_A \rangle. [![\vec{x}_A]]a : \vec{\sigma}_A \Rightarrow a(\vec{\tau}_A \Rightarrow \vec{\tau}_A \vec{\sigma}_A) \\ \text{curry}^a(M) : \vec{\sigma}_A &\longrightarrow a(\vec{\rho}_A \longrightarrow \vec{\tau}_A) &= \langle \vec{x}_A \rangle. [![\vec{x}_A]. M]a : \vec{\sigma}_A \Rightarrow a(\vec{\rho}_A \Rightarrow \vec{\tau}_A) \\ P \xrightarrow{a} Q : a(\vec{\sigma}_A \rightarrow \vec{\tau}_A) &\longrightarrow a(\vec{\rho}_A \rightarrow \vec{v}_A) &= \langle f \rangle. [!P; ?f. Q]a : a(\vec{\sigma}_A \Rightarrow \vec{\tau}_A) \Rightarrow a(\vec{\rho}_A \Rightarrow \vec{v}_A) \end{aligned}$$

Any two of these bifunctors  $\xrightarrow{a}$  and  $\xrightarrow{b}$  are naturally isomorphic to each other, with the natural isomorphism given at type type  $\tau$  by the *relocation* isomorphism.

$$a\langle x \rangle. [x]b : a(\tau) \Rightarrow b(\tau).$$

This will be explained in detail by Chapter 5.

**Remark 4.3.3.** Because symmetry isomorphisms on types at distinct locations are in fact a true identity, two different evaluation maps can apply to the same object: for example, given a stack of type  $\vec{\sigma}_A a(\vec{\sigma}_A \Rightarrow \vec{\tau}_A) b(\vec{\sigma}_A \Rightarrow \vec{v}_A)$ , where  $\vec{\sigma}_A$  contains no elements at  $a$  or  $b$ , we can apply either  $a\langle f \rangle. ?f$  to get  $b(\vec{\sigma}_A \Rightarrow \vec{v}_A) \vec{\tau}_A$  or  $b\langle g \rangle. ?g$  to get  $a(\vec{\sigma}_A \Rightarrow \vec{\tau}_A) \vec{v}_A$ . Categorically, this is because a CCC typically has a left- and right-closure which are modulated by a symmetry; here, some symmetries are now identities.

The following lemmata give us that the necessary equations for uniqueness of products and exponents are indeed derivable from the equational theory, and we will use these in the subsequent proof that the category of !FMC-terms is a CCC. The following law will be used to give us to uniqueness of products.

**Lemma 4.3.4.** *The following equation is sound in  $=_{eqn}$ :*

$$P ; \langle \vec{x}_A \rangle ; P ; \langle \vec{x}'_A \rangle. \langle y_A \rangle. [\vec{x}'_A] =_{\times} P : \Rightarrow \vec{\sigma}_A \vec{\tau}_A$$

where  $\vec{x}_A, \vec{x}'_A : \vec{\tau}_A$  and  $\vec{y}_A : \vec{\sigma}_A$ .

*Proof.* We calculate:

$$\begin{aligned} P &=_{\beta} P. \langle \vec{x}_A \vec{y}_A \rangle. [\vec{y}_A \vec{x}_A] \\ &=_{\beta} P. \langle \vec{x}_A \vec{y}_A \rangle. [\vec{y}_A]. [\vec{y}_A \vec{x}_A]. \langle \vec{x}'''_A \vec{y}''_A \rangle. [\vec{x}''_A] \\ &=_{\beta} P. \langle \vec{x}_A \vec{y}_A \rangle. [\vec{y}_A \vec{x}_A \vec{y}_A \vec{x}_A]. \langle \vec{x}'_A \vec{y}'_A \vec{x}''_A \rangle. [\vec{y}'_A \vec{x}'_A]. \langle \vec{x}'''_A \vec{y}''_A \rangle. [\vec{x}''_A] \\ &=_{\Delta} P ; P ; \langle \vec{x}'_A \vec{y}'_A \vec{x}''_A \rangle. [\vec{y}'_A \vec{x}'_A]. \langle \vec{x}'''_A \vec{y}''_A \rangle. [\vec{x}''_A] \\ &=_{\iota} P ; \langle \vec{x}'_A \rangle. P. \langle \vec{x}'''_A \rangle. \langle \vec{y}''_A \rangle. [\vec{x}''_A]. \end{aligned}$$

□

The following law will be used to give us uniqueness of exponents.

**Lemma 4.3.5.** *The following equation sound in  $=_{eqn}$ :*

$$S =_{\eta'} [!S; a\langle f \rangle. ?f]a : \Rightarrow a(\vec{\sigma}_A \Rightarrow \vec{\tau}_A),$$

where  $f : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ .

*Proof.* We calculate:

$$[!S; \langle f \rangle. ?f] =_{\eta} S; \langle g \rangle. [![g]. \langle f \rangle. ?f] =_{\beta} S; \langle g \rangle. [! ?g] =_{\tau} S; \langle g \rangle. [g] =_{id} S. \quad \square$$

**Remark 4.3.6.** In the presence of  $\{id, \beta, \phi\}$ ,  $\eta$  is derivable from  $\eta'$ :

$$\begin{aligned} S; \langle \vec{x}_A \rangle. [![\vec{x}_A]. M]a &=_{\eta'} [!S; \langle \vec{x}_A \rangle. [![\vec{x}_A]. M]a. a\langle f \rangle. ?f]a \\ &=_{\beta} [!S; \langle \vec{x}_A \rangle. ?![\vec{x}_A]. M]a \\ &=_{\phi} [!S; \langle \vec{x}_A \rangle. [\vec{x}_A]. M]a \\ &=_{id} [!S; M]a \end{aligned}$$

so that in the presence of  $\{id, \beta, \phi, \tau\}$ , the two are equivalent.

It is now shown that the equational theory suffices to make the category of !FMC-terms Cartesian closed.

**Theorem 4.3.7.** *The category !FMC( $\Sigma_A$ ) is Cartesian closed, with equipment given by Definition 4.3.1.*

*Proof.* Recalling Proposition 2.2.3 and Proposition 2.2.7, to prove we have a Cartesian closed category, we show existence of a terminal object and existence and uniqueness of products and existence and uniqueness of exponentials. Terminality of the empty type family follows from the terminality (!) equation. To verify existence of products, we show for  $M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ ,  $N : \vec{\sigma}_A \Rightarrow \vec{\nu}_A$ :

$$\begin{aligned} \langle M, N \rangle ; \pi_1 &= \langle \vec{x}_A \rangle. ([\vec{x}_A]. M ; [\vec{x}_A]. N ; \langle \vec{y}_A \rangle) \\ &=_{!} \langle \vec{x}_A \rangle. [\vec{x}_A]. M \\ &=_{id} M \\ \langle M, N \rangle ; \pi_2 &= \langle \vec{x}_A \rangle. ([\vec{x}_A]. M ; [\vec{x}_A]. N ; \langle \vec{y}_A \rangle. \langle \vec{z}_A \rangle. [\vec{y}_A]) \\ &=_{\iota} \langle \vec{x}_A \rangle. ([\vec{x}_A]. M ; \langle \vec{y}_A \rangle. [\vec{x}_A]. N) \\ &=_{!} \langle \vec{x}_A \rangle. [\vec{x}_A]. N \\ &=_{id} N \end{aligned}$$

where  $\vec{x}_A, \vec{s}_A, \vec{y}_A : \vec{\nu}_A$  and  $\vec{z}_A : \vec{\tau}_A$ , respectively. To verify the uniqueness of the product, we show that for  $M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ :

$$\begin{aligned} \langle M; \pi_1, M; \pi_2 \rangle &= \langle \vec{x}_A \rangle. ([\vec{x}_A]. M ; \pi_1 ; [\vec{x}_A]. M ; \pi_2) \\ &= \langle \vec{x}_A \rangle. [\vec{x}_A]. M ; \langle \vec{z}_A \rangle. [\vec{x}_A]. M ; \langle \vec{z}'_A \rangle. \langle \vec{y}_A \rangle. [\vec{z}'_A] \\ &=_{\times} \langle \vec{x}_A \rangle. [\vec{x}_A]. M \\ &=_{!\times} M \end{aligned}$$

where  $\vec{x}_A: \vec{\sigma}_A$ . To verify existence of exponentials, we show that for  $M: \vec{\sigma}_A \vec{\tau}_A \Rightarrow \vec{v}_A$ :

$$\begin{aligned} (\text{id}_{\vec{\rho}_A} \times \text{curry}(M)) ; \epsilon &= \langle \vec{x}_A \rangle. [![\vec{x}_A]. M] ; \langle f \rangle. ?f \\ &=_{\beta} \langle \vec{x}_A \rangle. ?![\vec{x}_A]. M \\ &=_{\phi} \langle \vec{x}_A \rangle. [\vec{x}_A]. M \\ &=_{\text{id}} M \end{aligned}$$

where  $\vec{x}_A: \vec{\sigma}_A$  and  $f: \vec{\rho}_A \Rightarrow \vec{\tau}_A$ . To verify uniqueness of exponentials, we show that for  $M: \vec{\sigma}_A \Rightarrow (\vec{\tau}_A \Rightarrow \vec{v}_A)$ :

$$\begin{aligned} \text{curry}((\text{id}_{\vec{\sigma}_A} \times M) ; \epsilon) &= \langle \vec{x}_A \rangle. [![\vec{x}_A]. M ; \langle f \rangle. ?f] \\ &=_{\eta'} \langle \vec{x}_A \rangle. [\vec{x}_A]. N \\ &=_{\text{id}} M \end{aligned}$$

where  $\vec{x}_A: \vec{\sigma}_A$ ,  $f: \vec{\rho}_A \Rightarrow \vec{\tau}_A$ . □

Note that the associators and unitors are given by identities, which makes the category a *strict* Cartesian closed category.

Let us proceed to show that the equational theory give above is valid with respect to the operational semantics given by the machine.

## 4.4 Machine Equivalence

The functional abstract machine gives the operational semantics of the !FMC. The notion of a *successful* run of the machine, where the run terminates in the state  $(S_A, \star)$ , is a novelty with respect to the  $\lambda$ -calculus. In Theorem 3.4.3, it was proved that every typed term successfully runs on the machine when given appropriately typed inputs. Informally, we can thus consider a term  $M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  as generating an input/output function from memories of type  $\vec{\sigma}_A$  to memories of type  $\vec{\tau}_A$ , giving a model of the !FMC. Indeed, this model gives rise to a natural notion of equivalence on !FMC-terms (*i.e.*, by considering which terms generate the same function). We call this *machine equivalence*, and it is a type of observational (or contextual) equivalence.

For the purposes of this section, we consider the computation signature to be empty, and the value signature to contain only values of base types. This is because the machine cannot take any further steps if it were to reach the states  $(S_A, c.M)$  or  $(S_A, ?v.M)$ , causing a premature failure state. Alternatively, we could consider constants as free variables, or include machine transitions for constants. Indeed, adding transitions for *e.g.* an addition function on a base type of natural numbers would be a sensible way to generate a finer equivalence on terms than is studied here.

The following is a definition which captures machine equivalence. Because the model we aim to capture is higher-order, the equivalence is naturally a *logical relation*, as originally described by Plotkin [91], and subsequently used in the study of

program equivalence by Pitts and Stark [84, 83]. We proceed to show this machine equivalence is a congruence on terms, and then show that the equational theory is valid with respect to it.

Note that the definition of machine equivalence models a strong notion of observational equivalence, which would not be appropriate if we included *e.g.* a location intended to model a stream of random variables.

**Definition 4.4.1.** Closed computations  $M, N : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  are *machine equivalent at type*  $\vec{\sigma}_A \Rightarrow \vec{\tau}_A$  if for equivalent inputs the machine gives equivalent outputs:

$$M \sim M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A \quad \triangleq \quad \forall S_A \sim S'_A : \vec{\sigma}_A. (S_A, M) \Downarrow \sim (S'_A, M') \Downarrow : \vec{\tau}_A,$$

where, similarly, closed values  $V, V' : \tau$  are machine equivalent at  $\tau$  if their executions are equivalent (or, at base type, if they are identical constants)

$$V \sim V' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A \quad \triangleq \quad ?V \sim ?V' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A \quad v \sim v' : \alpha \quad \triangleq \quad v \equiv v' : \alpha$$

where two memories are equivalent if their values are pairwise equivalent. Equivalence extends to open terms as follows.

$$\begin{aligned} \vec{u} : \vec{v} \vdash_c M \sim M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A &\triangleq \forall U \sim U' : \vec{v}. \{U/\vec{u}\}M \sim \{U'/\vec{u}\}M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A. \\ \vec{u} : \vec{v} \vdash_v V \sim V' : \tau &\triangleq \forall U \sim U' : \vec{v}. \{U/\vec{u}\}V \sim \{U'/\vec{u}\}V' : \tau. \end{aligned}$$

In the following, by congruence, we mean an equivalence relation  $\sim$  on terms such that equivalent terms in equivalent contexts are equivalent.

**Proposition 4.4.2.** *Machine equivalence ( $\sim$ ) is a congruence relation.*

*Proof.* We must show that  $(\sim)$  is symmetric, transitive, reflexive and closed under all contexts. Symmetry is immediate by the symmetry of the definition. Transitivity can be shown by a simple induction on the size of types. Reflexivity is not a priori clear, but follows trivially from closure under all contexts. We proceed to show by induction on type derivations that  $\sim$  is closed under all contexts. Throughout, let  $W \sim W' : \vec{\omega}_A$  and  $\sigma = \{W/\vec{w}\}$  and  $\sigma' = \{W'/\vec{w}\}$ , being wary that we use  $W$  here as a stack and not a value.

- Identity:  $\star \sim \star$  is tautologous.
- Sequential Execution: given  $\vec{w} : \vec{\omega} \vdash_v V \sim V' : \vec{\rho}_A \Rightarrow \vec{v}_A$  and  $\vec{w} : \vec{\omega} \vdash_c M \sim M' : \vec{v}_A \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , we must show

$$\sigma ?V'. M \sim \sigma' ?V'. M'.$$

Let  $S_A \sim S'_A : \vec{\sigma}_A$  and  $R_A \sim R'_A : \vec{\rho}_A$ . Applying the definition of substitution, we must thus show that  $T_A \sim T'_A : \vec{\tau}_A$ .

$$\frac{\frac{(S_A R_A, ?\sigma V. \sigma M)}{(S_A U_A, \sigma M)}}{(T_A, \star)} \quad \frac{\frac{(S'_A R'_A, ?\sigma' V'. \sigma' M')}{(S'_A U'_A, \sigma' M')}}{(T'_A, \star)}$$

We have  $U_A \sim U'_A : \vec{v}_A$  by  $\sigma V \sim \sigma' V'$ , and then the required result follows by  $\sigma M \sim \sigma' M'$ .

- Application: given  $\vec{w} : \omega \vdash_c M \sim M' : a(\rho)\vec{\sigma}_A \Rightarrow \vec{\tau}_A$  and  $\vec{w} : \omega \vdash_v V \sim V' : \rho$ , we must show

$$\sigma[V]a.M \sim \sigma'[V']a.M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A.$$

Let  $S_A \sim S'_A : \vec{\sigma}_A$ . Applying the definition of substitution, we must thus show that  $T_A \sim T'_A : \vec{\tau}_A$  in

$$\frac{\frac{(S_A, [\sigma V]a.\sigma M)}{(S_{A \setminus \{a\}}; S_a \cdot \sigma V, \sigma M)}}{(T_A, \star)} \quad \text{and} \quad \frac{\frac{(S'_A, [\sigma' V']a.\sigma' M')}{(S'_{A \setminus \{a\}}; S'_a \cdot \sigma' V', \sigma' M')}}{(T'_A, \star)}$$

This follows from  $\sigma M \sim \sigma' M'$  and  $\sigma V \sim \sigma' V'$ .

- Abstraction: given  $\vec{w} : \vec{\omega} \vdash_c M \sim M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , we must show

$$\sigma a\langle x \rangle.M \sim \sigma' a\langle x \rangle.M' : a(\rho)\vec{\sigma}_A \Rightarrow \vec{\tau}_A$$

Let  $S_A \sim S'_A : \vec{\sigma}_A$  and  $V \sim V' : \rho$ . Applying the definition of substitution, we must thus show that  $T_A \sim T'_A : \vec{\tau}_A$  in

$$\frac{\frac{(S_{A \setminus \{a\}}; S_a \cdot V, a\langle x \rangle.\sigma M)}{(S_A, \{V/x\}\sigma M)}}{(T_A, \star)} \quad \text{and} \quad \frac{\frac{(S'_{A \setminus \{a\}}; S'_a \cdot V', a\langle x \rangle.\sigma' M')}{(S'_A, \{V'/x\}\sigma' M')}}{(T'_A, \star)}$$

The result follows from  $\{V/x\}\sigma M \sim \{V'/x\}\sigma' M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ .

- Thunk: given  $\vec{w} : \vec{\omega} \vdash_c M \sim M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , we must show

$$\sigma !M \sim \sigma' !M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A.$$

Let  $S_A \sim S'_A : \vec{\sigma}_A$ . Applying the definition of substitution, we must thus show that  $T_A \sim T'_A : \vec{\tau}_A$  in

$$\frac{(S_A, ?!\sigma M)}{(S_A, \sigma M)} \quad \text{and} \quad \frac{(S'_A, ?!\sigma' M')}{(S'_A, \sigma' M')}$$

This follows from  $\sigma M \sim \sigma' M'$ . □

The following result closes this section. We work here using the derived *global* beta-equivalence, which clearly immediately implies the local beta and relocation equations which it replaces.



**Proposition 4.4.3** (Soundness). *For all closed, typed computations  $M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  and  $N: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , we have that*

$$M =_{\text{eqn}} N \quad \text{implies} \quad M \sim N: \vec{\sigma}_A \Rightarrow \vec{\tau}_A.$$

*Proof.* We verify the statement for each equation generating  $=_{\text{eqn}}$ , which suffices since by Proposition 4.4.2, which says machine equivalence  $\sim$  is closed under all contexts. In each case, we must show that the two sides of the equation, when evaluated on equivalent inputs, give equivalent outputs. Recall that by reflexivity of  $\sim$ , running a term  $M$  on two equivalent inputs results in equivalent outputs. Throughout, let  $S_A \sim S'_A: \vec{\sigma}_A$  and let  $(\epsilon_A, S) \Downarrow S_A$ , and that the evaluation of  $S: \Rightarrow \vec{\sigma}_A$  results a memory  $S_A$  pushed to the stack, whereas  $M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  takes  $S_A: \vec{\sigma}_A$  to  $T_A: \vec{\tau}_A$  and  $M: \vec{\sigma}_A \Rightarrow \vec{v}_A$  takes  $S_A: \vec{\sigma}_A$  to  $U_A: \vec{\tau}_A$ , etc..

- Identity: immediate from the following runs.

$$\frac{\frac{(\ S_A\ ,\ \langle \vec{x}_A \rangle \cdot [\vec{x}_A] \ )}{(\ \epsilon_A\ ,\ \ [S_A] \ )}}{(\ S_A\ ,\ \ \star \ )} \quad (\ S'_A\ ,\ \ \star \ )$$

- Beta: we show for the global beta case, with local beta a special case.  $T_A \sim T'_A$  follows from  $S_A \sim S'_A$  and the following runs.

$$\frac{\frac{(\ S_A\ ,\ \ [N]a \cdot a \langle x \rangle \cdot M \ )}{(\ S_{A \setminus \{a\}} ; S_a \cdot N\ ,\ \ a \langle x \rangle \cdot M \ )}}{(\ S_A\ ,\ \ \{N/x\}M \ )}}{(\ T_A\ ,\ \ \star \ )} \quad \frac{(\ S'_A\ ,\ \ \{N/x\}M \ )}{(\ T'_A\ ,\ \ \star \ )}$$

- Force:  $T_A \sim T'_A$  follows from  $S_A \sim S'_A$  and the following runs.

$$\frac{(\ S_A\ ,\ \ ?!M \ )}{(\ S_A\ ,\ \ M \ )}}{(\ T_A\ ,\ \ \star \ )} \quad \frac{(\ S'_A\ ,\ \ M \ )}{(\ T'_A\ ,\ \ \star \ )}$$

- Thunk: consider the following runs, assuming  $S_A \sim S'_A$ .

$$\frac{(\ S_A\ ,\ \ [!V] \ )}{(\ S_A \cdot !V\ ,\ \ \star \ )}}{(\ S'_A\ ,\ \ [V] \ )}}{(\ S'_A \cdot V\ ,\ \ \star \ )}$$

Then, we must prove  $?!V \sim ?V: \vec{\rho}_A \Rightarrow \vec{\tau}_A$ . Indeed,  $T_A \sim T'_A: \vec{\tau}_A$  follows from  $R_A \sim R'_A: \vec{\rho}_A$ , in the following runs.

$$\frac{(\ R_A\ ,\ \ ?!V \ )}{(\ R_A\ ,\ \ ?V \ )}}{(\ T_A\ ,\ \ \star \ )} \quad \frac{(\ R'_A\ ,\ \ ?V \ )}{(\ T'_A\ ,\ \ \star \ )}$$

- Eta: consider the following runs.

$$\begin{array}{c}
\frac{(\epsilon_A, S; \langle \vec{x}_A \rangle. [![\vec{x}_A]. M]a)}{(\vec{S}_A, \langle \vec{x}_A \rangle. [![\vec{x}_A]. M]a)} \\
\frac{(\epsilon_A, [![\vec{S}_A]. M]a)}{(\epsilon_{A \setminus \{a\}}; \epsilon_a \cdot [![\vec{S}_A]. M], \star)} \\
\\
\frac{(\epsilon_A, [!S; M]a)}{(\epsilon_{A \setminus \{a\}}; \epsilon_a \cdot S; M, \star)}
\end{array}$$

We are thus required to show that  $?![S_A]. M \sim ?!S; M: \vec{\tau}_A \Rightarrow \vec{v}_A$ . Let  $T_A \sim T'_A: \vec{\tau}_A$ . Then the required result of  $U_A \sim U'_A: \vec{v}_A$  is immediate from the following runs.

$$\begin{array}{c}
\frac{(\vec{T}_A, ?![S_A]; M)}{(\vec{T}_A, [S_A]; M)} \\
\frac{(\vec{T}_A \cdot \vec{S}_A, M)}{(\vec{U}_A, \star)} \\
\\
\frac{(\vec{T}'_A, ?!S; M)}{(\vec{T}'_A, S; M)} \\
\frac{(\vec{T}'_A \cdot \vec{S}_A, M)}{(\vec{U}'_A, \star)}
\end{array}$$

- Diagonal: immediate from the following runs.

$$\begin{array}{c}
\frac{(\epsilon_A, S; \langle \vec{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A])}{(\vec{S}_A, \langle \vec{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A])} \\
\frac{(\epsilon, [\vec{S}_A]. [\vec{S}_A])}{(\vec{S}_A, [\vec{S}_A])} \\
\frac{(\vec{S}_A, [\vec{S}_A])}{(\vec{S}_A \vec{S}_A, \star)} \\
\\
\frac{(\epsilon_A, S; S)}{(\vec{S}_A, S)} \\
\frac{(\vec{S}_A \vec{S}_A, \star)}{(\vec{S}_A \vec{S}_A, \star)}
\end{array}$$

- Interchange: Let  $R_A \sim R'_A: \vec{\rho}_A$ ,  $T_A \sim T'_A: \vec{\tau}_A$ . Then  $S_A \sim S'_A: \vec{\sigma}_A$  and  $U_A \sim U'_A: \vec{v}_A$  follows by assumption and the following runs.

$$\begin{array}{c}
\frac{(\vec{R}_A \vec{T}_A, \langle \vec{x}_A \rangle. M. [\vec{x}_A]. N)}{(\vec{R}_A, M. [\vec{T}_A]. N)} \\
\frac{(\vec{S}_A, [\vec{T}_A]. N)}{(\vec{S}_A \vec{T}_A, N)} \\
\frac{(\vec{S}_A \vec{T}_A, N)}{(\vec{S}_A \vec{U}_A, \star)} \\
\\
\frac{(\vec{R}'_A \vec{T}'_A, N. \langle \vec{y}_A \rangle. M. [\vec{y}_A])}{(\vec{R}'_A \vec{U}'_A, \langle \vec{y}_A \rangle. M. [\vec{y}_A])} \\
\frac{(\vec{R}'_A, M. [\vec{U}'_A])}{(\vec{S}'_A, [\vec{U}'_A])} \\
\frac{(\vec{S}'_A, [\vec{U}'_A])}{(\vec{S}'_A \vec{U}'_A, \star)}
\end{array}$$

- Terminal: Note that both runs (by their type) must return the empty stack, which is trivially related to itself by  $\sim$ , the result follows immediately. We illustrate the two runs anyway.

$$\begin{array}{c}
\frac{(\epsilon_A, S; \langle \vec{x}_A \rangle)}{(\vec{S}_A, \langle \vec{x}_A \rangle)} \\
\frac{(\epsilon_A, \star)}{(\epsilon_A, \star)}
\end{array}
\quad
(\epsilon_A, \star)$$

- Permutation: Let  $W \sim W' : \rho$ , then  $T_A \sim T'_A : \vec{\tau}_A$  follows from this assumption,  $S_A \sim S'_A$ , and the following runs.

$$\begin{array}{c}
( S_{A \setminus \{a,b\}}; S_a \cdot W; S_b \quad , \quad [V]b. a\langle w \rangle. M ) \\
( S_{A \setminus \{a,b\}}; S_a \cdot W; S_b \cdot V \quad , \quad a\langle w \rangle. M ) \\
( S_{A \setminus \{a,b\}}; S_a; S_b \cdot V \quad , \quad \{W/w\}M ) \\
\hline
( T_A \quad , \quad \star ) \\
\hline
( S'_{A \setminus \{a,b\}}; S'_a \cdot W'; S_b \quad , \quad a\langle w \rangle. [V]b. M ) \\
( S'_{A \setminus \{a,b\}}; S'_a; S'_b \quad , \quad [V]b. \{W'/w\}M ) \\
( S'_{A \setminus \{a,b\}}; S'_a; S'_b \cdot V \quad , \quad \{W'/w\}M ) \\
\hline
( T'_A \quad , \quad \star )
\end{array}$$

□

**Remark 4.4.4.** In the case there are no constants, it trivially follows from the definition of  $(\sim)$  that all terms of the same type are identified. Consider then the !FMC with one value constant  $c : \alpha$  for each base type  $\alpha$ : every computation type is inhabited by the following term:

$$(\vec{\sigma}_A \Rightarrow \vec{\tau}_A)^* = \langle \vec{x}_A \rangle. [\vec{\tau}_A^*] \quad \alpha^* = c .$$

However, in this case, all terms of the same type are machine equivalent. For example,  $\langle x \rangle. [c] \sim \langle x \rangle. [x] : \alpha \Rightarrow \alpha$ , since for given any input (which must be the unique constant), the outputs are the same. Given *two* value constants,  $c, c' : \alpha$ , the machine equivalence becomes non-degenerate. Consider again the two terms above: they become distinguishable by the input  $c'$ . Nevertheless, for any number of value constants of base type, the machine equivalence is strictly coarser than the equational theory. Similar to the simply-typed  $\lambda$ -calculus, there are not enough contexts to distinguish between all equationally distinct terms. To see this, consider again that the only distinguishable terms at type  $\alpha \Rightarrow \alpha$  are the identity and constant functions, as above. Then the Church numerals  $\langle f \rangle. [f^n] : (\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha)$ , for  $n \geq 1$ , are not distinguishable by any of their possible inputs (namely, the identity and constant functions).

We can construct a category in the same way as !FMC( $\Sigma_A$ ), except with terms taken modulo machine equivalence. In fact, this is a Cartesian closed category, too, although it is too degenerate to be the *free* one.

**Corollary 4.4.5.** !FMC-terms over a signature whose only constants are values of base type, taken modulo machine equivalence, form a Cartesian closed category.

*Proof.* Machine equivalence  $(\sim)$  contains the equational theory  $=_{\text{eqn}}$  (Theorem 4.4.3), and terms modulo  $=_{\text{eqn}}$  form a Cartesian closed category (Theorem 4.3.7). □

In fact, the category given by terms modulo machine equivalence is an extensional collapse of the category of terms modulo the equational theory, as commented in

Remark 4.4.4. That is, we have that computations  $M \sim N: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  if for every computation  $S: \Rightarrow \vec{\sigma}_A$ ,  $S;M = S;N: \Rightarrow \vec{\tau}_A$ . This follows immediately from the definition of machine equivalence and the observation that  $S;M \sim S;N: \Rightarrow \vec{\tau}_A$  implies  $S;M = S;N: \Rightarrow \vec{\tau}_A$ .

## Chapter 5

# Categorical Semantics of the Functional Machine Calculus

This chapter contains the first main contribution of this thesis: the result that the category of !FMC-terms is equivalent to the *free* Cartesian closed category. This equivalence is naturally factorized into two parts: the equivalence between the free CCC (that is, the STLC) and the !SLC, and that between the !SLC and the !FMC. We give self-contained proofs for both.

We begin with the first equivalence, by giving an explicit construction of the free functor from the category of STLC-terms to the category of !SLC-terms. This is immediate from Theorem 4.3.7 of the previous chapter. Its inverse functor must send the category of *closed* !SLC-terms, composed by sequencing, into the category of *open*  $\lambda$ -terms, composed by substitution. How to do this is explained in the upcoming section.

We expect there to be a denotational equivalence between these two categories, because, although the !SLC gives us a sequencing mechanism on top of the  $\lambda$ -calculus, we expect its expressivity to remain the same: that is, to be able to express all the same higher-order functions. Further, we have chosen the equational theory so as to identify exactly what is needed to make the !SLC a Cartesian closed category (and no more).

The intuition behind the denotational equivalence of the !SLC and the !FMC is that of the isomorphism between an indexed product (*i.e.*, a memory) and a plain product (*i.e.*, a stack), given by fixing an ordering of indices.

The functor from !SLC-terms to !FMC-terms is given by a simple *embedding*, *i.e.*, an inclusion of terms. This necessitates a choice of location on which to embed. For the inverse, we collapse the memory of the !FMC into single large stack (depending on a chosen ordering on locations). This collapsing of types (memories) is then lifted to terms, giving the desired functor. The technical difficulty of this proof comes from necessarily working up-to (natural) isomorphism, and from the higher-order nature of the !FMC, which means that we likewise will need appropriate higher-

order isomorphisms.<sup>1</sup>

One perspective on this interpretation is that the !FMC gives *multiple* copies of the !SLC, one for each location. This is confirmed by the observation that we have multiple isomorphic arrow types in the category of !FMC-terms: one for each location. The interpretation of the !FMC into the !SLC can then be viewed as collapsing these multiple copies into one. This perspective highlights how this result relies on the assumption that *every location is equivalent*, and is reliant on the fact the type system accounts for *everything* happening to the memory, not just actions on the main location.

To summarize, to achieve the main result, we factorize the equivalence of the !FMC into the free CCC into two steps, as below:

$$\begin{array}{ccccc}
 & \xrightarrow{\llbracket - \rrbracket^<} & & \xrightarrow{\llbracket - \rrbracket'} & \\
 \text{FMC}(\Sigma_A) & & \text{SLC}(\lfloor \Sigma_A \rfloor) & & \text{CCC}(\lfloor \Sigma_A \rfloor) \\
 & \xleftarrow{\{-\}_a} & & \xleftarrow{\{-\}'} & 
 \end{array}$$

where  $\lfloor - \rfloor$  is the collapsing of the types of an !FMC signature, giving an !SLC signature (and, isomorphically, an STLC signature).<sup>2</sup>

Given these results, an obvious question is: if the simply-typed !FMC is equivalent to the simply typed  $\lambda$ -calculus, what is to be gained from study of the !FMC? We respond to this question in detail in the conclusion of the chapter. For now, we mention that, as expected, the denotational perspective collapses the all operational refinements made to the STLC by the !FMC.

## 5.1 The Sequential $\lambda$ -Calculus is Equivalent to the STLC

We construct a map from the !SLC to the STLC (rather than the free CCC) in order to compare calculi. We will, however, sometimes confuse the two when the difference is irrelevant. Note, this map will also bear a close resemblance to the quantitative interpretation of terms given in Chapter 6. The inverse map will be presented and discussed in the subsequent subsection.

We begin with the translation from the simply-typed  $\lambda$ -calculus to the !SLC. Indeed, knowing that the category of !FMC-terms is Cartesian closed, we have in hand a functor from the free CCC (*i.e.*, the simply-typed  $\lambda$ -calculus) to the category of !FMC-terms.

### 5.1.1 The Free Functor: STLC to !SLC

We first define an interpretation of !SLC types into the types of the  $\lambda$ -calculus. We then relate an !SLC-signature and a signature with values used to generate the  $\lambda$ -calculus.

---

<sup>1</sup>Note, the embedding functor, which maps stacks to memories which make use of only one location, is essentially surjective on objects, via the isomorphism exhibited in that section.

<sup>2</sup>Rightly, this should be parameterize in  $<$  also. Note also that  $\lfloor - \rfloor$  is surjective on signatures.

**Definition 5.1.1.** The *interpretation* of value types and stack types  $\llbracket \vec{\tau} \rrbracket$  and contexts are respectively defined by induction as:

$$\llbracket \alpha \rrbracket = \alpha \quad \llbracket \vec{\sigma} \Rightarrow \vec{\tau} \rrbracket = \llbracket \vec{\sigma} \rrbracket \rightarrow \llbracket \vec{\tau} \rrbracket \quad \llbracket \tau_1 \dots \tau_n \rrbracket = \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket$$

**Definition 5.1.2.** Given an !SLC-signature  $\Sigma = (\Sigma_0, \Sigma_c, \Sigma_v, \text{dom}, \text{cod}, \text{val})$ , define its *interpretation*  $\llbracket \Sigma \rrbracket$  as the  $\lambda$ -signature  $(\Sigma_0, \Sigma_c, \Sigma_v, \llbracket \text{dom} \rrbracket, \llbracket \text{cod} \rrbracket, \llbracket \text{val} \rrbracket)$ , where  $\llbracket \text{dom} \rrbracket(c) = \llbracket \text{dom}(c) \rrbracket$ ,  $\llbracket \text{cod} \rrbracket(c) = \llbracket \text{cod}(c) \rrbracket$  and  $\llbracket \text{val} \rrbracket(v) = \llbracket \text{val}(v) \rrbracket$ . Since  $\llbracket \Sigma \rrbracket$  and  $\Sigma$  are clearly isomorphic, and it will be clear from context if we are in the setting of the !SLC or the  $\lambda$ -calculus, we confuse the two and write  $\llbracket \Sigma \rrbracket$  as  $\Sigma$ .

By the freeness of  $\text{CCC}(\Sigma)$ , knowing that  $\text{!SLC}(\Sigma)$  is a CCC (Theorem 4.3.7), we can construct, for every function  $j : \Sigma \rightarrow U(\text{!SLC}(\Sigma))$ , a unique CCC-functor  $\{-\}_j : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$ . Clearly, there is a canonical choice of  $j$  given by the inclusion of the signature. Indeed, we will proceed show that the functor thus induced is in fact an equivalence of categories.

**Definition 5.1.3.** Define  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$  to be the unique CCC-functor making the following diagram commute:

$$\begin{array}{ccc} \Sigma & \xrightarrow{i} & U(\text{CCC}(\Sigma)) \\ & \searrow j & \downarrow U(\{-\}) \\ & & U(\text{!SLC}(\Sigma)) \end{array}$$

where  $i$  and  $j$  are the obvious inclusions of signatures.

We can present this functor as a map from the STLC to closed, typed !SLC-terms, defined inductively on type derivations. Note that, because we send *open*  $\lambda$ -terms to *closed* !SLC-terms, we must map  $\vdash$  into  $\Rightarrow$ , turning the free variables of a  $\lambda$ -term into the input stack of the corresponding !SLC-term.

**Lemma 5.1.4** (The Free Functor). *The functor  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$  can equivalently be defined on  $\lambda$ -terms as follows. On types and contexts, respectively, define inductively:*

$$\begin{aligned} \{\alpha\} &= \alpha \\ \{A \rightarrow B\} &= \{A\}^r \Rightarrow \{B\} & \{A_1, \dots, A_n\} &= \{A_1\}, \dots, \{A_n\} \\ \{A \times B\} &= \{A\}\{B\} \\ \{\top\} &= \epsilon_A \end{aligned}$$

where  $\{-\}^r$  is defined as  $\{-\}$ , except that  $\{A \times B\}^r = \{B\}^r \{A\}^r$  and  $\{A_1, \dots, A_n\}^r = \{A_n\}^r \dots \{A_1\}^r$ . Define a closed !SLC-term

$$\{\Gamma \vdash M : A\} : \{\Gamma\}^r \Rightarrow \{A\}$$

inductively on the type derivation of  $M$ :

$$\begin{aligned}
\{\Pi', y : A, \Pi \vdash y : A\} &= \langle \vec{x} \rangle. \langle \vec{y} \rangle. \langle \vec{z} \rangle. [\vec{y}] \\
\{\Pi \vdash v : A\} &= \langle \vec{x} \rangle. [v] \\
\{\Pi \vdash c @ M : B\} &= \{\Pi \vdash M : A\}; c \\
\{(p, q) : A \times B, \Pi \vdash M : C\} &= \{p : A, q : B, \Pi \vdash M : C\} \\
\{() : \top, \Pi \vdash M : C\} &= \{\Pi \vdash M : C\} \\
\{\Pi \vdash (M, N) : A \times B\} &= \langle \vec{x} \rangle. ([\vec{x}]. \{\Pi \vdash M : A\}; [\vec{x}]. \{\Pi \vdash N : B\}) \\
\{\Pi \vdash () : \top\} &= \langle \vec{x} \rangle \\
\{\Pi \vdash M @ N : B\} &= \langle \vec{x} \rangle. ([\vec{x}]. \{\Pi \vdash N : A\}; [\vec{x}]. \{\Pi \vdash M : A \rightarrow B\}); \langle f \rangle. ?f \\
\{\Pi \vdash \lambda p. M : A \rightarrow B\} &= \langle \vec{x} \rangle. [![\vec{x}]; \{p : A, \Pi \vdash M : B\}]
\end{aligned}$$

where in each case  $\vec{x} : \{\Gamma\}$ ,  $\vec{y} : \{A\}$ ,  $\vec{z} : \{\Gamma'\}$ ,  $f : \{A\}^r \Rightarrow \{B\}$ . Note that coherences are translated as identities, and we present the binary and nullary cases of  $n$ -ary tuples, and take the remaining cases to be clear.

*Proof.* Recall the usual equivalence between the category of simply typed  $\lambda$ -terms and the free Cartesian closed category (with both generated by the signature  $\Sigma$ ), given in Theorem 2.5.3. Composing this with the definitions of the equipment of  $\text{!FMC}(\Sigma)$  from Definition 4.3.1 thus gives the functor described above (modulo a small amount of simplification via beta reduction).  $\square$

**Remark 5.1.5.** The translation of the (admissible) cut, exchange and weakening rules are, respectively:

$$\begin{aligned}
\{\Pi', \Pi \vdash M\{N/x\} : B\} &= \langle \vec{xy} \rangle. [\vec{y} \vec{y} \vec{x}]. \{\Pi', \Pi \vdash N : A\}; [\vec{x}]. \{\Pi', p : A, \Pi \vdash M : B\} \\
\{\Pi', q : B, p : A, \Pi \vdash M : C\} &= \langle \vec{xy} \vec{z} \rangle. [\vec{y} \vec{z} \vec{x}]. \{\Pi', p : A, q : B, \Pi \vdash M : C\} \\
\{\Pi, p : A \vdash M : B\} &= \langle \vec{y} \rangle. \{\Pi \vdash M : B\},
\end{aligned}$$

where, in the first case,  $\vec{x} : \{\Pi\}$ ,  $\vec{y} : \{\Pi'\}$ , in the following two cases case  $\vec{x} : \{\Pi\}$ ,  $\vec{y} : \{A\}$  and  $\vec{z} : \{B\}$ , and in the last case,  $p \notin \text{fv}(M)$ .

### 5.1.2 The Interpretation: $\text{!SLC}$ to $\text{STLC}$

We construct an interpretation functor  $\llbracket - \rrbracket : \text{!SLC}(\Sigma) \rightarrow \text{CCC}(\Sigma)$ , which we will show is inverse to  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$ .

$$\llbracket \Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau} \rrbracket : \llbracket \vec{\sigma} \rrbracket \times \llbracket \Gamma \rrbracket \vdash \llbracket \vec{\tau} \rrbracket$$

The top-level arrow  $\Rightarrow$  becomes sequent entailment  $\vdash$  with the type of the output stack becoming the type of the  $\lambda$ -term. The corresponding  $\lambda$ -term then has two “inputs”, given by its context. The first corresponds to the input stack of the  $\text{!SLC}$ , the second to the free variables of the  $\text{!SLC}$ .



The interpretation can be considered via the operational intuition of a stack machine *with closures*. The interpretation of a term can be considered as running the machine on its first set of inputs, with the closure dealing with the free variables (the second input) as usual. The application of the !SLC is interpreted as pushing a new element to the first input, while the abstraction of the !SLC is interpreted as shifting from the first input to the second. The  $\lambda$ -term itself thus corresponds to the state of the stack at the end of a machine run, as a function of the (interpretation of the) input stack and the free variables of the !SLC.

The reason for the distinction between computations and values is to give the most natural correspondence with the  $\lambda$ -calculus. This now becomes clear: we see in the following definition that the force and thunk constructs of the !SLC correspond to application and abstraction in the  $\lambda$ -calculus. The soundness proof to follow shows clearly how the force and thunk *equations* correspond to the beta and eta equations of the  $\lambda$ -calculus. Thereby, the ability to *force* a value to act as a computation, and to *thunk* a computation, considering it a value, is exactly what makes the !SLC higher-order.

**Definition 5.1.6.** The *interpretation* of value types and stack types  $\llbracket \vec{\tau} \rrbracket$  and contexts are respectively defined by induction as:

$$\begin{aligned} \llbracket \alpha \rrbracket &= \alpha & \llbracket \tau_1 \dots \tau_n \rrbracket &= \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket \\ \llbracket \tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket &= \llbracket \vec{\sigma} \rrbracket \rightarrow \llbracket \vec{\tau} \rrbracket & \llbracket \tau_1, \dots, \tau_n \rrbracket &= \llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket \end{aligned}$$

where  $\alpha \in \Sigma_0$ . Note, the separating ( $|$ ) symbol in the following is just another notation for the standard product, or comma. Define the *interpretation* of !SLC-terms as open  $\lambda$ -terms

$$\llbracket \Gamma \vdash_v V : \tau \rrbracket : \llbracket \Gamma \rrbracket \vdash \llbracket \tau \rrbracket \quad \llbracket \Gamma \vdash_c M : \tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket : \llbracket \vec{\sigma} \rrbracket \mid \llbracket \Gamma \rrbracket \vdash \llbracket \vec{\tau} \rrbracket$$

by mutual induction on type derivations of computations and values, as follows.

$$\begin{aligned} \llbracket \Gamma, t : \tau, \Delta \vdash_v t : \tau \rrbracket(v, t, w) &= t \\ \llbracket \Gamma \vdash_v v : \tau \rrbracket(w) &= v \\ \llbracket \Gamma \vdash_v !M : \tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket(v) &= \lambda s. \llbracket \Gamma \vdash_c M : \tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket(s \mid v) \\ \llbracket \Gamma \vdash_c \star : \tilde{\sigma} \Rightarrow \vec{\sigma} \rrbracket(s \mid v) &= s \\ \llbracket \Gamma \vdash_c c.M : \tilde{\rho}\tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket(s, r \mid v) &= \llbracket \Gamma \vdash_c M : \tilde{\rho}\tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket(s, c @ r \mid v) \\ \llbracket \Gamma \vdash_c \langle x \rangle.M : \tilde{\rho}\tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(s, r \mid v) &= \llbracket x : \rho, \Gamma \vdash_c M : \tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket(s \mid r, v) \\ \llbracket \Gamma \vdash_c [V].M : \tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket(s \mid v) &= \llbracket \Gamma \vdash_c M : \tilde{\rho}\tilde{\sigma} \Rightarrow \vec{\tau} \rrbracket(s, \llbracket \Gamma \vdash_v V : \rho \rrbracket(v) \mid v) \\ \llbracket \Gamma \vdash_c ?V.M : \tilde{\rho}\tilde{\tau} \Rightarrow \vec{v} \rrbracket(t, r \mid v) &= \llbracket \Gamma \vdash_c M : \tilde{\sigma}\tilde{\tau} \Rightarrow \vec{v} \rrbracket(t, \llbracket \Gamma \vdash_v V : \tilde{\rho} \Rightarrow \vec{\sigma} \rrbracket(v) @ r \mid v) \end{aligned}$$

In each case, we have  $r : \llbracket \rho \rrbracket$  or  $r : \llbracket \vec{\rho} \rrbracket$ ,  $s : \llbracket \vec{\sigma} \rrbracket$ ,  $t : \llbracket \tau \rrbracket$ ,  $v : \llbracket \Gamma \rrbracket$  and  $w : \llbracket \Delta \rrbracket$ . Note that the inputs to  $\llbracket - \rrbracket$  are open  $\lambda$ -terms to be substituted for free variables, and in particular having inputs corresponding to the context is another way to express a valuation. We omit the context and/or types of terms inside the interpretation function when it is clear.

### 5.1.3 Soundness of the Interpretation

We proceed to show that the interpretation functor is well-defined. We first give the derived interpretations of our defined notation on  $\vec{x}$ .

**Lemma 5.1.7** (Application and Abstraction). *We have for every computation:*

$$\begin{aligned} \llbracket \vec{x} : \vec{\rho}, \Gamma \vdash_c [\vec{x}]. N : \vec{\sigma} \Rightarrow \vec{\tau} \rrbracket (s \mid r, v) &= \llbracket \vec{x} : \vec{\rho}, \Gamma \vdash_c N : \vec{\rho\sigma} \Rightarrow \vec{\tau} \rrbracket (s, r \mid r, v) \\ \llbracket \Gamma \vdash_c \langle \vec{x} \rangle. M : \vec{\rho\sigma} \Rightarrow \vec{\tau} \rrbracket (s, r \mid v) &= \llbracket \vec{x} : \vec{\rho}, \Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau} \rrbracket (s \mid r, v). \end{aligned}$$

*Proof.* For each statement, proceed by induction on the size of  $\vec{x}$ .  $\square$

**Remark 5.1.8.** Throughout the rest of this chapter, we will freely use the abstraction lemma without mention; however, we will still call attention to the use of the application lemma.

The following lemma gives the interpretation of the (admissible) exchange and weakening rules.

**Lemma 5.1.9** (Exchange and Weakening). *We have, for every computation:*

$$\begin{aligned} \llbracket \Gamma, y : \pi, x : \rho, \Gamma' \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau} \rrbracket (s \mid v, p, r, w) &= \\ \llbracket \Gamma, x : \rho, y : \pi, \Gamma' \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau} \rrbracket (s \mid v, r, p, w) \\ \llbracket \Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau} \rrbracket (s \mid v) &= \llbracket x : \rho, \Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau} \rrbracket (s \mid r, v) \end{aligned}$$

where, in the latter case,  $r \notin \text{fv}(M)$ .

*Proof.* Induction on the type derivation of  $M$ .  $\square$

The following lemma give the interpretation of the admissible sequencing rule, which is necessary for proving that  $\llbracket - \rrbracket$  is a well-defined functor.

**Lemma 5.1.10** (Sequencing). *For computations  $\Gamma \vdash_c N : \vec{\rho} \Rightarrow \vec{v}$  and  $\Gamma \vdash_c M : \vec{v\sigma} \Rightarrow \vec{\tau}$ , we have that:*

$$\llbracket N ; M \rrbracket (t, r \mid v) = \llbracket M \rrbracket (t, \llbracket N \rrbracket (r \mid v) \mid v)$$

*Proof.* We proceed by induction on the type derivation of  $N$ . In each case, we begin by unfolding the definition of sequencing.

- Identity:

$$\begin{aligned} \llbracket \star ; M \rrbracket (t, r \mid v) &= \llbracket M \rrbracket (t, \underline{r} \mid v) \\ (\text{defn. } \llbracket - \rrbracket) &= \llbracket M \rrbracket (t, \llbracket \star \rrbracket (r) \mid v) \end{aligned}$$

- Sequential Constant:

$$\begin{aligned}
\llbracket \textcolor{red}{c}. N ; M \rrbracket(t, s, r \mid v) &= \llbracket \textcolor{red}{c}. (N ; M) \rrbracket(t, s, r \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket N ; M \rrbracket(t, s, c @ r, \mid v) \\
(\text{l.H.}) &= \llbracket M \rrbracket(t, \llbracket N \rrbracket(s, c @ r \mid v) \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket M \rrbracket(t, \llbracket \textcolor{red}{c}. N \rrbracket(s, r \mid v) \mid v)
\end{aligned}$$

- Application:

$$\begin{aligned}
\llbracket \llbracket V \rrbracket. N ; M \rrbracket(t, r \mid v) &= \llbracket \llbracket V \rrbracket. (M ; N) \rrbracket(t, r \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket M ; N \rrbracket(t, r, \llbracket V \rrbracket(v) \mid v) \\
(\text{l.H.}) &= \llbracket M \rrbracket(t, \llbracket N \rrbracket(r, \llbracket V \rrbracket(v) \mid v) \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket M \rrbracket(t, \llbracket \llbracket V \rrbracket. N \rrbracket(r \mid v) \mid v)
\end{aligned}$$

- Abstraction:

$$\begin{aligned}
\llbracket \langle x \rangle. N ; M \rrbracket(t, r, s \mid v) &= \llbracket \langle x \rangle. (N ; M) \rrbracket(t, r, s \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket N ; M \rrbracket(t, r \mid s, v) \\
(\text{l.H.}) &= \llbracket M \rrbracket(t, \llbracket N \rrbracket(r \mid s, v) \mid s, v) \\
(\text{Weak.}) &= \llbracket M \rrbracket(t, \llbracket N \rrbracket(r \mid s, v) \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket M \rrbracket(t, \llbracket \langle s \rangle. N \rrbracket(r, s \mid v) \mid v)
\end{aligned}$$

- Sequential Execution:

$$\begin{aligned}
\llbracket \llbracket ?V \rrbracket. N ; M \rrbracket(t, r, s \mid v) &= \llbracket \llbracket ?V \rrbracket. (N ; M) \rrbracket(t, r, s \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket N ; M \rrbracket(t, r, \llbracket V \rrbracket(v) @ s \mid v) \\
(\text{l.H.}) &= \llbracket M \rrbracket(t, \llbracket N \rrbracket(r, \llbracket V \rrbracket(v) @ s \mid v) \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket M \rrbracket(t, \llbracket \llbracket ?V \rrbracket. N \rrbracket(r, s \mid v) \mid v)
\end{aligned}$$

□

We show the interpretation function is well-defined, *i.e.*, that it preserves equality of terms.

**Proposition 5.1.11** (Soundness). *For any computations  $\Gamma \vdash_c M, N : \vec{\sigma} \Rightarrow \vec{\tau}$ , we have*

$$M =_{eqn} N \text{ implies } \llbracket M \rrbracket = \llbracket N \rrbracket.$$

*Proof.* We prove the statement in the case of each equation. We begin each case by unfolding the definition of  $\llbracket - \rrbracket$ , unless otherwise specified.

- Identity (id):

$$\begin{aligned}
\llbracket \langle \vec{x} \rangle. [\vec{x}] \rrbracket(s \mid v) &= \llbracket [\vec{x}] \rrbracket(\epsilon \mid s, v) \\
(\text{App.}) &= \llbracket \star \rrbracket(s \mid s, v) \\
(\text{Weak.}) &= \llbracket \star \rrbracket(s \mid v)
\end{aligned}$$

- Beta ( $\beta$ ):

$$\begin{aligned}
\llbracket [V]. \langle x \rangle. ?x \rrbracket (s \mid v) &= \llbracket \langle x \rangle. ?x \rrbracket (s, \llbracket V \rrbracket (v) \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket ?x \rrbracket (s \mid \llbracket V \rrbracket (v), v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket x \rrbracket (\llbracket V \rrbracket (v), v) @ s \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket V \rrbracket (v) @ s \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket ?V \rrbracket (s \mid v)
\end{aligned}$$

- Eta ( $\eta$ ):

$$\begin{aligned}
\llbracket S ; \langle \tilde{x} \rangle. [! \tilde{x}]. M \rrbracket (\epsilon \mid v) \text{ (Seq.)} &= \llbracket \langle \tilde{x} \rangle. [! \tilde{x}]. M \rrbracket (\llbracket S \rrbracket (\epsilon \mid v), v) \\
(\text{Abs.}) &= \llbracket [! \tilde{x}]. M \rrbracket (\epsilon \mid \llbracket S \rrbracket (\epsilon \mid v), v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [! \tilde{x}]. M \rrbracket (\llbracket S \rrbracket (\epsilon \mid v), v) \\
(\text{defn. } \llbracket - \rrbracket) &= \lambda s. \llbracket [! \tilde{x}]. M \rrbracket (s \mid \llbracket S \rrbracket (\epsilon \mid v), v) \\
(\text{App. } \llbracket - \rrbracket) &= \lambda s. \llbracket M \rrbracket (s, \llbracket S \rrbracket (\epsilon \mid v) \mid v) \\
(\text{Seq.}) &= \lambda s. \llbracket S ; M \rrbracket (s \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [! S ; M] \rrbracket (v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [! S ; M] \rrbracket (\epsilon \mid v)
\end{aligned}$$

- Force ( $\phi$ ):

$$\begin{aligned}
\llbracket [! V] \rrbracket (s \mid v) &= \llbracket [! V] \rrbracket (v) @ s \\
(\text{defn. } \llbracket - \rrbracket) &= (\lambda s'. \llbracket V \rrbracket (s' \mid v)) @ s \\
&=_{\beta} \llbracket V \rrbracket (s \mid v)
\end{aligned}$$

- Thunk ( $\tau$ ):

$$\begin{aligned}
\llbracket [! ?x] \rrbracket (v) &= \lambda s. \llbracket ?x \rrbracket (s \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \lambda s. \llbracket x \rrbracket (v) @ s \\
&=_{\eta} \llbracket x \rrbracket (v)
\end{aligned}$$

- Terminal (!):

$$\begin{aligned}
\llbracket S ; \langle \tilde{x} \rangle \rrbracket (\epsilon \mid v) \text{ (Seq.)} &= \llbracket \langle \tilde{x} \rangle \rrbracket (\llbracket S \rrbracket (\epsilon \mid v) \mid v) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket \star \rrbracket (\epsilon \mid \llbracket S \rrbracket (\epsilon \mid v), v) \\
(\text{Weak.}) &= \llbracket \star \rrbracket (\epsilon \mid v)
\end{aligned}$$

- Diagonal ( $\Delta$ ):

$$\begin{aligned}
\llbracket S; \langle \tilde{x} \rangle. [\tilde{x}]. [\tilde{x}] \rrbracket(\epsilon \mid v) \text{ (Seq.)} &= \llbracket [\tilde{s}]. [\tilde{s}] \rrbracket(\llbracket S \rrbracket(\epsilon \mid v) \mid v) \\
&\text{(defn. } \llbracket - \rrbracket) = \llbracket [\tilde{x}]. [\tilde{x}] \rrbracket(\epsilon \mid \llbracket S \rrbracket(\epsilon \mid v), v) \\
&\text{(App.)} = \llbracket [\tilde{x}] \rrbracket(\llbracket S \rrbracket(\epsilon \mid v) \mid \llbracket S \rrbracket(\epsilon \mid v), v) \\
&\text{(App.)} = \llbracket \star \rrbracket(\llbracket S \rrbracket(\epsilon \mid v), \llbracket S \rrbracket(\epsilon \mid v) \mid \llbracket S \rrbracket(\epsilon \mid v), v) \\
&\text{(Weak.)} = \llbracket \star \rrbracket(\llbracket S \rrbracket(\epsilon \mid v), \llbracket S \rrbracket(\epsilon \mid v) \mid v) \\
&\text{(Seq.)} = \llbracket S; \star \rrbracket(\llbracket S \rrbracket(\epsilon \mid v) \mid v) \\
&\text{(Seq.)} = \llbracket S; S \rrbracket(\epsilon \mid v)
\end{aligned}$$

- Interchange ( $\iota$ ):

$$\begin{aligned}
\llbracket M; \langle \tilde{x} \rangle. (N; [\tilde{x}]) \rrbracket(s \mid v) &= \llbracket N; [\tilde{x}] \rrbracket(s \mid \llbracket M \rrbracket(\epsilon \mid v), v) \\
&\text{(Seq.)} = \llbracket [\tilde{x}] \rrbracket(\llbracket N \rrbracket(s \mid \llbracket M \rrbracket(\epsilon \mid v), v) \mid \llbracket M \rrbracket(\epsilon \mid v), v) \\
&\text{(App.)} = \llbracket \star \rrbracket(\llbracket N \rrbracket(s \mid v), \llbracket M \rrbracket(\epsilon \mid v) \mid \llbracket M \rrbracket(\epsilon \mid v), v) \\
&\text{(defn. } \llbracket - \rrbracket) = (\llbracket N \rrbracket(s \mid v), \llbracket M \rrbracket(\epsilon \mid v)) \\
&\text{(defn. } \llbracket - \rrbracket) = \llbracket \star \rrbracket(\llbracket N \rrbracket(s \mid v), \llbracket M \rrbracket(\epsilon \mid v) \mid v) \\
&\text{(Seq.)} = \llbracket M \rrbracket(\llbracket N \rrbracket(s \mid v), v) \\
&\text{(Seq.)} = \llbracket N; M \rrbracket(s \mid v)
\end{aligned}$$

□

**Corollary 5.1.12.** *The interpretation  $\llbracket - \rrbracket : \text{!SLC}(\rightarrow) \text{CCC}(\Sigma)$  is a well-defined functor.*

*Proof.* The interpretation preserves equality (i.e., is well-defined) by Proposition 5.1.11, preserves identity by definition and preserves composition by the Sequencing Lemma 5.1.10. □

### 5.1.4 Equivalence

In this section we demonstrate completeness of the free functor  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$  by proving that

$$\llbracket \{\Gamma \vdash M : A\} \rrbracket =_{\beta\eta\pi} \Gamma \vdash M : A,$$

or, equivalently stated, that the free functor is faithful. Since the free functor sends equipment of the free CCC to the corresponding equipment in !SLC( $\Sigma$ ), the proof of this essentially amounts to showing that the interpretation  $\llbracket - \rrbracket : \text{!SLC}(\Sigma) \rightarrow \text{CCC}(\Sigma)$  maps the equipment of !SLC( $\Sigma$ ) back to the corresponding equipment in CCC( $\Sigma$ ).

Further, it is then proved that  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$  in fact gives an equivalence of categories, by showing that

$$\{\llbracket M : \tilde{\sigma} \Rightarrow \tilde{\tau} \rrbracket\} =_{\text{ccc}} M : \tilde{\sigma} \Rightarrow \tilde{\tau},$$

for closed computations  $M$ , or, equivalently stated, that the free functor also full.

A reference for the two translations is presented in Figure 5.1.

$M$	$\{M\}$	$M$	$\llbracket M \rrbracket$
$x, y, z \vdash y$	$\langle \hat{x} \rangle. \langle \hat{y} \rangle. \langle \hat{z} \rangle. [\hat{y}]$	$\star$	$s \mid v \vdash s$
$\Pi \vdash c @ M$	$\{M\}; c$	$c. M$	$s, r \mid v \vdash \llbracket M \rrbracket(s, c @ r \mid v)$
$\Pi \vdash (M, N)$	$\langle \hat{x} \rangle. ([\hat{x}]. \{M\}; [\hat{x}]. \{N\})$	$[V]. M$	$s \mid v \vdash \llbracket M \rrbracket(s, \llbracket V \rrbracket(v) \mid v)$
$\Pi \vdash ()$	$\langle \hat{x} \rangle$	$\langle x \rangle. M$	$s, r \mid v \vdash \llbracket M \rrbracket(s \mid r, v)$
$\Pi \vdash M @ N$	$\langle \hat{x} \rangle. ([\hat{x}]. \{M\}; [\hat{x}]. \{N\}); \langle f \rangle. ?f$	$?V. M$	$t, r \mid v \vdash \llbracket M \rrbracket(t, \llbracket V \rrbracket(v) @ r \mid v)$
$\Pi \vdash \lambda p. M$	$\langle \hat{x} \rangle. [![\hat{x}]. \{M\}]$	$!M$	$v \vdash \lambda s. \llbracket M \rrbracket(s \mid v)$
$\Pi \vdash v$	$\langle \hat{x} \rangle. [v]$	$v$	$w \vdash v$

Figure 5.1: Translations between the Sequential  $\lambda$ -calculus and the  $\lambda$ -calculus

**Theorem 5.1.13** (Completeness of the Free Functor). *The functor  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$  is faithful.*

*Proof.* We show that  $\llbracket \{\Gamma \vdash M : A\} \rrbracket =_{\beta\eta\pi} \Gamma \vdash M : A$ , or equivalently that

$$\llbracket \{s \vdash M\} \rrbracket(s' \mid \epsilon) =_{\beta\eta\pi} M\{s'/s\}.$$

We proceed by induction on the type derivation of  $\Gamma \vdash M : A$ . In each case (excepting for abstraction), we slightly abuse notation and choose for the fresh free variables  $s' = s$  to avoid proliferation of  $\alpha$ -conversions. Although the  $\lambda$ -calculus has  $n$ -ary products, we show that binary and nullary cases and take the rest to be clear. In each case, we begin by unfolding the definition of  $\{-\}$ .

- Variable:

$$\begin{aligned}
\llbracket \{z, y, x \vdash y : A\} \rrbracket(z, y, x \mid \epsilon) &= \llbracket \langle \hat{x} \rangle. \langle \hat{y} \rangle. \langle \hat{z} \rangle. [\hat{y}] \rrbracket(z, y, x \mid \epsilon) \\
&\text{(defn. } \llbracket - \rrbracket) = \llbracket \langle \hat{y} \rangle. \langle \hat{z} \rangle. [\hat{y}] \rrbracket(z, y, x \mid x) \\
&\text{(defn. } \llbracket - \rrbracket) = \llbracket \langle \hat{z} \rangle. [\hat{y}] \rrbracket(z \mid y, x) \\
&\text{(defn. } \llbracket - \rrbracket) = \llbracket [\hat{y}] \rrbracket(\epsilon \mid z, y, x) \\
&\text{(Exch. + App.)} = y
\end{aligned}$$

- Value Constant:

$$\begin{aligned}
\llbracket \{s \vdash v\} \rrbracket(s \mid \epsilon) &= \llbracket \langle \hat{x} \rangle. [v] \rrbracket(s \mid \epsilon) \\
&\text{(defn. } \llbracket - \rrbracket) = \llbracket [v] \rrbracket(\epsilon \mid s) \\
&\text{(defn. } \llbracket - \rrbracket) = \llbracket v \rrbracket(s) \\
&\text{(defn. } \llbracket - \rrbracket) = v
\end{aligned}$$

- Sequential Constant:

$$\begin{aligned}
\llbracket \{s \vdash c @ M\} \rrbracket (s \mid \epsilon) &= \llbracket \{M\} ; \textcolor{red}{c} \rrbracket (s \mid \epsilon) \\
(\text{Seq.}) &= \llbracket \textcolor{red}{c} \rrbracket (\llbracket \{M\} \rrbracket (s \mid \epsilon) \mid \epsilon) \\
(\text{l.H.}) &= \llbracket \textcolor{red}{c} \rrbracket (M \mid \epsilon) \\
(\text{defn. } \llbracket - \rrbracket) &= c @ M
\end{aligned}$$

- Pattern:

$$\begin{aligned}
\llbracket \{s, (p, q) \vdash M\} \rrbracket (s, (p, q) \mid \epsilon) &= \llbracket \{s, p, q \vdash M\} \rrbracket (s, (p, q) \mid \epsilon) \\
&=_{\pi\eta} \llbracket \{s, p, q \vdash M\} \rrbracket (s, \pi_1(p, q), \pi_2(p, q) \mid \epsilon) \\
&=_{\beta} \llbracket \{s, p, q \vdash M\} \rrbracket (s, p, q \mid \epsilon) \\
(\text{l.H.}) &= M
\end{aligned}$$

- Binary Product:

$$\begin{aligned}
\llbracket \{s \vdash (M, N)\} \rrbracket (s \mid \epsilon) &= \llbracket \langle \textcolor{red}{x} \rangle . ([\textcolor{red}{x}]. \{M\} ; [\textcolor{red}{x}]. \{N\}) \rrbracket (s \mid \epsilon) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [\textcolor{red}{x}]. \{M\} ; [\textcolor{red}{x}]. \{N\} \rrbracket (\epsilon \mid s) \\
(\text{App.}) &= \llbracket \{M\} ; [\textcolor{red}{x}]. \{N\} \rrbracket (s \mid s) \\
(\text{Seq.}) &= \llbracket [\textcolor{red}{x}]. \{N\} \rrbracket (\llbracket \{M\} \rrbracket (s \mid s) \mid s) \\
(\text{Weak.}) &= \llbracket [\textcolor{red}{x}]. \{N\} \rrbracket (\llbracket \{M\} \rrbracket (s \mid \epsilon) \mid s) \\
(\text{l.H.}) &= \llbracket [\textcolor{red}{x}]. \{N\} \rrbracket (M \mid s) \\
(\text{App.}) &= \llbracket \{N\} \rrbracket (M, s \mid s) \\
(\text{Seq.}) &= \llbracket \star \rrbracket (M, \llbracket \{N\} \rrbracket (s \mid s) \mid s) \\
(\text{defn. } \llbracket - \rrbracket) &= (M, \llbracket \{N\} \rrbracket (s \mid s)) \\
(\text{Weak.}) &= (M, \llbracket \{N\} \rrbracket (s \mid \epsilon)) \\
(\text{l.H.}) &= (M, N)
\end{aligned}$$

- Nullary Product:

$$\begin{aligned}
\llbracket \{s \vdash ()\} \rrbracket (s \mid \epsilon) &= \llbracket \langle \textcolor{red}{x} \rangle \rrbracket (s \mid \epsilon) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket \star \rrbracket (\epsilon \mid s) \\
(\text{defn. } \llbracket - \rrbracket) &= ()
\end{aligned}$$

- Application:

$$\begin{aligned}
\llbracket \{s : \Gamma \vdash M @ N : B\} \rrbracket (s \mid \epsilon) &= \llbracket \langle \textcolor{red}{x} \rangle . ([\textcolor{red}{x}]. \{N\} ; [\textcolor{red}{x}]. \{M\}) ; \langle f \rangle . ?f \rrbracket (s \mid \epsilon) \\
(\text{Seq.}) &= \llbracket \langle f \rangle . ?f \rrbracket (\llbracket \langle \textcolor{red}{x} \rangle . ([\textcolor{red}{x}]. \{N\} ; [\textcolor{red}{x}]. \{M\}) \rrbracket (s \mid \epsilon) \mid \epsilon) \\
(\text{See Bin. Prod. Case}) &= \llbracket \langle f \rangle . ?f \rrbracket (N, M \mid \epsilon) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket ?f \rrbracket (N \mid M) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket f \rrbracket (M) @ N \\
(\text{defn. } \llbracket - \rrbracket) &= M @ N
\end{aligned}$$

- Abstraction:

$$\begin{aligned}
\llbracket \{s \vdash \lambda p.M\} \rrbracket (s \mid \epsilon) &= \llbracket \langle \vec{x} \rangle. [\![\vec{x}]\!]. \{M\} \rrbracket (s \mid \epsilon) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [\![\vec{x}]\!]. \{M\} \rrbracket (\epsilon \mid s) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [\![\vec{x}]\!]. \{M\} \rrbracket (s) \\
(\text{defn. } \llbracket - \rrbracket) &= \lambda p. \llbracket [\![\vec{x}]\!]. \{M\} \rrbracket (p \mid s) \\
(\text{App.}) &= \lambda p. \llbracket \{M\} \rrbracket (p, s \mid s) \\
(\text{Weak.} + \text{I.H.}) &= \lambda p.M
\end{aligned}$$

□

**Theorem 5.1.14** (Fullness of the Free Functor). *The functor  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$  is full.*

*Proof.* We require that for closed terms  $M : \vec{\sigma} \Rightarrow \vec{\tau}$ , that  $\{\llbracket M \rrbracket\} =_{\text{eqn}} M$ . We actually prove a stronger statement: for all (open) terms  $\Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}$

$$\begin{aligned}
\{s : \llbracket \vec{\sigma} \rrbracket \mid v : \llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket (s \mid v) : \llbracket \vec{\tau} \rrbracket\} &=_{\text{eqn}} \langle \vec{v} \rangle. M : \vec{v\sigma} \Rightarrow \vec{\tau} \\
\{v : \llbracket \Gamma \rrbracket \vdash \llbracket V \rrbracket (v) : \llbracket \tau \rrbracket\} &=_{\text{eqn}} \langle \vec{v} \rangle. [V] : \vec{v} \Rightarrow \tau.
\end{aligned}$$

where  $\vec{v} : \Gamma$ . We proceed by mutual induction on the type derivations of  $M$  and  $V$ . In each case, we begin by unfolding the definition of  $\llbracket - \rrbracket$ . We will make use of the interpretation of the cut rule from Remark 5.1.5.

- Variable:

$$\begin{aligned}
\{z, y, x \vdash \llbracket y \rrbracket (z, y, x)\} &= \{z, y, x \vdash y\} \\
(\text{defn. } \{-\}) &= \langle \vec{x} \rangle. \langle y \rangle. \langle \vec{z} \rangle. [y]
\end{aligned}$$

- Thunk:

$$\begin{aligned}
\{v \vdash \llbracket !M \rrbracket (v)\} &= \{v \vdash \lambda s. \llbracket M \rrbracket (s \mid v)\} \\
(\text{defn. } \{-\}) &= \langle \vec{v} \rangle. [\![\vec{v}]\!]. \{s, v \vdash \llbracket M \rrbracket (s \mid v)\} \\
(\text{I.H.}) &= \langle \vec{v} \rangle. [\![\vec{v}]\!]. \langle \vec{w} \rangle. \{\vec{w}/\vec{v}\} M \\
&=_{\beta} \langle \vec{v} \rangle. [\![M]\!]
\end{aligned}$$

□

- Value Constant:

$$\begin{aligned}
\{w \vdash_v \llbracket v \rrbracket (w)\} &= \{w \vdash_v v\} \\
&= \langle \vec{w} \rangle. [v]
\end{aligned}$$

- Identity:

$$\begin{aligned}
\{s, v \vdash \llbracket \star \rrbracket (s \mid v)\} &= \{s, v \vdash s\} \\
(\text{defn. } \{-\}) &= \langle \vec{v} \rangle. \langle \vec{x} \rangle. [\![\vec{x}]\!] \\
&=_{\text{id}} \langle \vec{v} \rangle. \star
\end{aligned}$$



- Sequential Constant:

$$\begin{aligned}
\{\llbracket \mathbf{c}.M \rrbracket(s, r \mid v)\} &= \{\llbracket M \rrbracket(s, c@r \mid v)\} \\
(\text{Weak.} + \text{Cut.}) &= \langle \vec{v}\vec{w}\vec{x} \rangle. [\vec{x}]. [\vec{x}\vec{w}\vec{v}]. \{s, r, v \vdash c@r\}. [\vec{w}\vec{v}]. \llbracket M \rrbracket(s, t \mid r, v) \\
(\text{I.H.}) &= \langle \vec{v}\vec{w}\vec{x} \rangle. [\vec{x}]. [\vec{x}\vec{w}\vec{v}]. \{s, r, v \vdash c@r\}. [\vec{r}\vec{v}]. \langle \vec{v}'\vec{w}' \rangle. \{\vec{v}'/\vec{v}\}M \\
&=_{\beta} \langle \vec{v}\vec{w}\vec{x} \rangle. [\vec{x}]. [\vec{x}\vec{w}\vec{v}]. \{s, r, v \vdash c@r\}. M \\
(\text{defn. } \{-\}) &= \langle \vec{v}\vec{x} \rangle. [\vec{x}]. [\vec{x}\vec{w}\vec{v}]. \{s, r, v \vdash r\}. c. M \\
(\text{defn. } \{-\}) &= \langle \vec{v}\vec{x} \rangle. [\vec{x}]. [\vec{x}\vec{w}\vec{v}]. \langle \vec{v}'\vec{w}'\vec{x}' \rangle. [\vec{w}']. c. M \\
&=_{\beta} \langle \vec{v}\vec{w}\vec{x} \rangle. [\vec{x}\vec{w}]. c. M \\
&=_{\text{id}} \langle \vec{v} \rangle. c. M
\end{aligned}$$

- Application:

$$\begin{aligned}
\{\llbracket [\mathbf{V}].M \rrbracket(s \mid v)\} &= \{\llbracket M \rrbracket(s, \llbracket V \rrbracket(v) \mid v)\} \\
(\text{Weak.} + \text{Cut.}) &= \langle \vec{v}\vec{x} \rangle. [\vec{x}]. [\vec{x}\vec{v}]. \{\llbracket V \rrbracket(s, v)\}. [\vec{v}]. \llbracket M \rrbracket(s, t \mid v) \\
(\text{I.H.}) &= \langle \vec{v}\vec{x} \rangle. [\vec{x}]. [\vec{x}\vec{v}]. \langle \vec{v}'\vec{x}' \rangle. \{\{v'/v\}V\}. [\vec{v}]. \llbracket M \rrbracket(s, t \mid v) \\
&=_{\beta} \langle \vec{v}\vec{x} \rangle. [\vec{x}]. [V]. [\vec{v}]. \llbracket M \rrbracket(s, t \mid v) \\
(\text{I.H.}) &= \langle \vec{v}\vec{x} \rangle. [\vec{x}]. [V]. [\vec{v}]. \langle \vec{v}' \rangle. \{\vec{v}'/\vec{v}\}M \\
&=_{\beta} \langle \vec{v}\vec{x} \rangle. [\vec{x}]. [V]. M \\
&=_{\text{id}} \langle \vec{v} \rangle. [V]. M
\end{aligned}$$

- Abstraction:

$$\begin{aligned}
\{s, r, v \vdash \llbracket \langle \mathbf{x} \rangle. M \rrbracket(s, r \mid v)\} &= \{s, r, v \vdash \llbracket M \rrbracket(s \mid r, v)\} \\
(\text{I.H.}) &= \langle \vec{v} \rangle. \langle \mathbf{x} \rangle. M
\end{aligned}$$

- Sequential Execution:

$$\begin{aligned}
\{\llbracket ?V.M \rrbracket(s, r \mid v)\} &= \{\llbracket M \rrbracket(s, \llbracket V \rrbracket(v) @ r \mid v)\} \\
(\text{Weak.} + \text{Cut.}) &= \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{x}\overrightarrow{wv}]. \{s, r, v \vdash \llbracket V \rrbracket(v) @ r\}. [\vec{wv}]. \llbracket M \rrbracket(s, t \mid r, v) \\
(\text{l.H.}) &= \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{x}\overrightarrow{wv}]. \{s, r, v \vdash \llbracket V \rrbracket(v) @ r\}. [\vec{wv}]. \langle \overrightarrow{v'}\overrightarrow{r'} \rangle. \{\overrightarrow{v'}/\overrightarrow{v}\}M \\
&=_{\beta} \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{x}\overrightarrow{wv}]. \{s, r, v \vdash \llbracket V \rrbracket(v) @ r\}M \\
(\text{defn. } \{-\}) &= \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{x}\overrightarrow{wv}]. \langle \overrightarrow{v'}\overrightarrow{w'}\overrightarrow{x'} \rangle. \\
&\quad [\vec{x'}\overrightarrow{w'}\overrightarrow{v'}]. \{s, r, v \vdash r\}. [\vec{x'}\overrightarrow{w'}\overrightarrow{v'}]. \{s, r, v \vdash \llbracket V \rrbracket(v)\}M \\
&=_{\beta} \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{x}\overrightarrow{wv}]. \{s, r, v \vdash r\}. [\vec{x}\overrightarrow{wv}]. \{s, r, v \vdash \llbracket V \rrbracket(v)\}M \\
(\text{defn. } \{-\}) &= \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{x}\overrightarrow{wv}]. \langle \overrightarrow{v'}\overrightarrow{w'}\overrightarrow{x'} \rangle. [\vec{w'}']. [\vec{x}\overrightarrow{wv}]. \\
&\quad \{s, r, v \vdash \llbracket V \rrbracket(v)\}M \\
&=_{\beta} \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{w}]. [\vec{x}\overrightarrow{wv}]. \{s, r, v \vdash \llbracket V \rrbracket(v)\}M \\
(\text{l.H.}) &= \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{w}]. [\vec{x}\overrightarrow{wv}]. \langle \overrightarrow{v'}\overrightarrow{w'}\overrightarrow{x'} \rangle. [\{v'/v\}V]. M \\
&=_{\beta} \langle \overrightarrow{vw}x \rangle. [\vec{x}]. [\vec{w}]. [V]. M \\
&=_{\text{id}} \langle \overrightarrow{v} \rangle. [V]. M
\end{aligned}$$

**Corollary 5.1.15.** *The categories  $\text{!SLC}(\Sigma)$  and  $\text{CCC}(\Sigma)$  are equivalent.*

*Proof.* We have shown that the free Cartesian closed functor  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$  is fully faithful (Theorems 5.1.13 and 5.1.14), and it is clearly surjective on objects, thus we have an equivalence of categories.  $\square$

Indeed, we have that  $\llbracket - \rrbracket : \text{!SLC}(\Sigma) \rightarrow \text{CCC}(\Sigma)$  is an inverse functor to  $\{-\} : \text{CCC}(\Sigma) \rightarrow \text{!SLC}(\Sigma)$ . Note that it also follows that the interpretation  $\llbracket - \rrbracket$  is in fact a CCC-functor. Additionally, we note that, while the pattern  $\lambda$ -calculus is non-strict, we can consider instead the strict CCC [71], as discussed in Chapter 2. The two functors described are indeed strict, in the sense that they map associators and unitors to the identity, and strictly preserve products and exponentials.

The result above immediately gives us that the first-order !SLC, with an equational theory generated by the first-order equations, is a sound and fully complete language for *string diagrams* [54], in the way illustrated in Section 4.1 of this chapter. To be precise, the category of first-order !SLC-terms can be defined similarly to !SLC( $\Sigma$ ), and the same argument as made in this section (omitting the cases with higher-order constructs), then gives us that this is in fact the free Cartesian category. Being cautious of Remark 4.2.4, the above results should also extend easily to the *linear* (higher-order) !SLC and the free symmetric monoidal (closed) category.

## 5.2 The !FMC is Equivalent to the !SLC

Here, it is proved that the category of !SLC-terms, which we now know to be the free CCC, is equivalent to the category of !FMC-terms. The structure of this section is as follows.

We will give an *embedding*  $\{-\}_a$  of the !SLC into the !FMC, parameterized by the location  $a \in A$  on which it is embedded. The inverse interpretation  $\llbracket - \rrbracket^<$  will be given by an extension to terms of the *collapse* of a memory into a single stack. This is done by fixing an order  $<$  on locations, and then concatenating the various stacks of the !FMC into one large stack, in that order. This action on types must then be lifted to terms in an appropriate way, and the two functors arising from embedding and collapsing must be shown to be inverse. Although the roundtrip given by embedding stacks into memories and then collapsing memories into stacks is an obvious identity, the reverse composition is only equivalent to the identity functor up-to natural isomorphism. Thus, we require to prove that there exists a natural isomorphism  $\kappa$  satisfying the following commutative diagram.

$$\begin{array}{ccc} \vec{\sigma}_A & \xrightarrow{M} & \vec{\tau}_A \\ \kappa \downarrow & & \downarrow \kappa \\ \{\llbracket \vec{\sigma}_A \rrbracket\} & \xrightarrow{\{\llbracket M \rrbracket\}} & \{\llbracket \vec{\tau}_A \rrbracket\} \end{array}$$

This isomorphism will be exhibited in the following subsection. It is given on base types by the relocation isomorphism, but requires an appropriate generalization for higher-order types.

### 5.2.1 Embedding: !SLC to !FMC

Recall that an !SLC-signature is an !FMC-signature with just this one *main* location. Note, there is nothing special about the main location  $\lambda$ , except that we single it out by convention. Recall also that we will freely confuse stack types and memory types generated over a single location, which are clearly isomorphic, *i.e.*, the !SLC is considered to have no memory and just a single stack.

**Definition 5.2.1.** Given an !SLC-signature  $\Sigma$ , define the category !SLC( $\Sigma$ ) to be !FMC( $\Sigma_A$ ), where  $A = \{\lambda\}$ .

In order to relate the category of !FMC-terms and !SLC-terms, we first deal with the fact they are generated over different signatures. We will collapse the types of constants in the !FMC-signature, which is generated over a set of locations  $A$ , into types of the !SLC-signature, which is generated over a single location  $\lambda$ . Throughout this section, we fix  $A = \{a_1, \dots, a_m\}$ .

Consider that an indexed product (*i.e.*, a memory) is isomorphic to a standard product (*i.e.*, a stack) by fixing a choice of ordering on its indices. Following this idea, we now define the *collapse* of !FMC-types into !SLC-types.<sup>3</sup> We will later extend this definition to act on terms, which will then forms the desired equivalence.

---

<sup>3</sup>Note that a significant amount of extra work goes into some of the definitions and proofs of this section in order to deal with arbitrary !FMC-signatures. A simpler presentation would be possible if we were to restrict the !FMC of this section to be generated by a signature with constants acting exclusively on the main location.

**Definition 5.2.2.** Given a strict order  $<$  on the set of locations  $A = \{a_1, a_2, a_3, \dots\}$ , such that  $a_i < a_j$  when  $i < j$ , define the *collapse interpretation*  $\llbracket - \rrbracket^<$  of value, stack and memory !FMC-types generated over  $A$  into !SLC-types (generated over  $\{\lambda\}$ ), respectively, by induction.

$$\begin{aligned} \llbracket \alpha \rrbracket^< &= \alpha & \llbracket \tau_1 \dots \tau_n \rrbracket^< &= \llbracket \tau_1 \rrbracket^< \dots \llbracket \tau_n \rrbracket^< \\ \llbracket \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket^< &= \llbracket \tilde{\sigma}_A \rrbracket^< \Rightarrow \llbracket \vec{\tau}_A \rrbracket^< & \llbracket \vec{\tau}_A \rrbracket^< &= \llbracket \vec{\tau}_{a_1} \rrbracket^< \dots \llbracket \vec{\tau}_{a_m} \rrbracket^< \end{aligned}$$

We will fix the order  $<$  throughout this section, and omit it on  $\llbracket - \rrbracket^<$ , writing just  $\llbracket - \rrbracket$ .

**Definition 5.2.3.** Given an !FMC-signature  $\Sigma_A = (\Sigma_0, \Sigma_c, \Sigma_v, \text{dom}, \text{cod}, \text{val})$ , define its *collapse*  $[\Sigma]$  as  $(\Sigma_0, \Sigma_c, \Sigma_v, [\text{dom}], [\text{cod}], [\text{val}])$ , where  $[\text{dom}](c) = \llbracket \text{dom}(c) \rrbracket$ ,  $[\text{cod}](c) = \llbracket \text{cod}(c) \rrbracket$  and  $[\text{val}](v) = \llbracket \text{val}(v) \rrbracket$ . We will write  $[c]$  and  $[v]$  to distinguish the constant symbols of  $[\Sigma_A]$  and the constant symbols  $c$  and  $v$  of  $\Sigma_A$ .

The aim, then, is to prove that  $\text{!FMC}(\Sigma_A) \simeq \text{!SLC}([\Sigma_A])$ . The desired equivalence of categories will indeed be up-to-isomorphism, and we exhibit (what will be proved to be) the relevant natural isomorphism  $\kappa$  in the following definition. This will be given at base types by simple relocation, but must be appropriately lifted to higher-order types: we cannot simply return the variable since it has type  $\tilde{\sigma}_A \Rightarrow \vec{\tau}_A$  and not type  $[\tilde{\sigma}_A \Rightarrow \vec{\tau}_A]$  (in the case of  $\kappa$ , and vice-versa for  $\kappa^{-1}$ ), hence the conjugation by appropriate isomorphisms of smaller type. This then generalizes from singleton types to memories of any size in the obvious way.

**Definition 5.2.4.** For each memory type  $\vec{\tau}_A$ , we define terms

$$\kappa(\vec{\tau}_A) : \tilde{\tau}_A \Rightarrow \llbracket \vec{\tau}_A \rrbracket \quad \kappa^{-1}(\vec{\tau}_A) : \llbracket \vec{\tau}_A \rrbracket \Rightarrow \vec{\tau}_A,$$

and *iterated collapse* and *iterated embed* functions  $[-]^*$  and  $\lceil - \rceil^*$ , on (families of) variables, for each memory type, by mutual induction on types<sup>4</sup>.

$$\begin{aligned} \kappa(\vec{\tau}_A) &= \langle \tilde{x}_A \rangle. [\llbracket \vec{x}_A \rrbracket^*] : \tilde{\tau}_A \Rightarrow \llbracket \vec{\tau}_A \rrbracket \\ [\vec{x}_A : \tau_A]^* &= [\vec{x}_{a_m}]_p^* \dots [\vec{x}_{a_1}]_p^* : \llbracket \vec{\tau}_A \rrbracket \\ [x : \tilde{\rho}_A \Rightarrow \vec{v}_A]^* &= !\kappa^{-1}(\tilde{\rho}_A) ; ?x. \kappa(\vec{v}_A) : \llbracket \tilde{\sigma}'_A \Rightarrow \vec{\tau}'_A \rrbracket \\ [x : \alpha]^* &= x : \llbracket \alpha \rrbracket \end{aligned}$$

$$\begin{aligned} \kappa^{-1}(\vec{\tau}_A) &= \langle \tilde{x} \rangle. [\lceil \vec{x} \rceil^*] : \llbracket \vec{\tau}_A \rrbracket \Rightarrow \vec{\tau}_A \\ \lceil \vec{x}_{a_m} \dots \vec{x}_{a_1} : \llbracket \vec{\tau}_A \rrbracket \rceil^* &= \{ \lceil \vec{x}_b \rceil_p^* \mid b \in A \} : \vec{\tau}_A \\ \lceil x : \llbracket \tilde{\rho}_A \Rightarrow \vec{v}_A \rrbracket \rceil^* &= !\kappa(\tilde{\rho}_A) ; ?x. \kappa^{-1}(\vec{v}_A) : \tilde{\sigma}'_A \Rightarrow \vec{\tau}'_A \\ \lceil x : \llbracket \alpha \rrbracket \rceil^* &= x : \alpha \end{aligned}$$

where  $[-]_p^*$  and  $\lceil - \rceil_p^*$  are the pointwise application of  $[-]^*$  and  $\lceil - \rceil^*$  on stacks, respectively, and we take  $\tilde{\sigma}'_A \Rightarrow \vec{\tau}'_A$  to be of smaller type than  $\vec{\tau}_A$ . We will omit the type associated with  $\kappa(\vec{\tau}_A)$  and  $\kappa^{-1}(\vec{\tau}_A)$  when it is clear. Note, the action of  $\lceil - \rceil^*$  on stacks is determined by  $\vec{\tau}_A$ .

<sup>4</sup>Note that a type has the same size as its collapse.

**Lemma 5.2.5** (Isomorphism). *For each type,  $\kappa$  and  $\kappa^{-1}$  are inverse to each other, as are  $\lceil - \rceil^*$  and  $\lfloor - \rfloor^*$ .*

*Proof.* Proceed by mutual induction on types. Note, it is clear that  $\lceil - \rceil^*$  and  $\lfloor - \rfloor^*$  are inverses. We can then calculate the following.

$$\begin{aligned}\kappa; \kappa^{-1} &= \langle \vec{x}_A \rangle. [\langle \vec{x}_A \rangle^*]. \langle \vec{x} \rangle. [\langle \vec{x} \rangle^*] = \langle \vec{x}_A \rangle. [[\langle \vec{x}_A \rangle^*]^*] = \langle \vec{x}_A \rangle. [\vec{x}_A] = \star \\ \kappa^{-1}; \kappa &= \langle \vec{x} \rangle. [\langle \vec{x} \rangle^*]. \langle \vec{x}_A \rangle. [\langle \vec{x}_A \rangle^*] = \langle \vec{x} \rangle. [[\langle \vec{x} \rangle^*]^*] = \langle \vec{x} \rangle. [\vec{x}] = \star\end{aligned}\quad \square$$

Note that the definition of  $\kappa$  relies on the strength of simple types: we need to know the size of the stacks in memory.

The embedding of the !SLC into the !FMC is given by the obvious inclusion of terms, except for the case of sequential constants. Recalling the aim is to prove  $!SLC(\lfloor \Sigma_A \rfloor) \simeq !FMC(\Sigma_A)$ , constants from the collapsed signature  $\lfloor \Sigma_A \rfloor$  must thus be mapped to those of the !FMC signature  $\Sigma_A$ , and this is dealt with using the isomorphisms defined above.

Note that, similar to the previous section, the inputs to the following function are terms to be substituted for free variables, *i.e.*, this is another way to express a valuation. The commas appearing in the inputs just represent concatenation, and could be omitted. This convention will appear throughout this section.

**Definition 5.2.6.** We define the *embedding*  $\{-\}_a$  of !SLC-types into !FMC-types as

$$\begin{aligned}\{\alpha\}_a &= \alpha & \{\tau_1 \dots \tau_n\}_a &= a(\{\tau_1\}_a \dots \{\tau_n\}_a) \\ \{\vec{\sigma} \Rightarrow \vec{\tau}\}_a &= \{\vec{\sigma}\}_a \Rightarrow \{\vec{\tau}\}_a & \{\tau_1, \dots, \tau_n\}_a &= \{\tau_1\}_a, \dots, \{\tau_n\}_a\end{aligned}$$

and the embedding of !SLC-terms over  $\lfloor \Sigma_A \rfloor$  into !FMC-terms over  $\Sigma_A$ ,

$$\begin{aligned}\{\Gamma\}_a \vdash_c \{\Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}\}_a &: \{\vec{\sigma}\}_a \Rightarrow \{\vec{\tau}\}_a \\ \{\Gamma\}_a \vdash_v \{\Gamma \vdash_c V : \tau\}_a &: \{\tau\}_a\end{aligned}$$

is given by mutual induction on type derivations of computations and values as follows.

$$\begin{aligned}\{\Gamma, x : \tau, \Delta \vdash_v x : \tau\}_a(\vec{v}, y, \vec{w}) &= y \\ \{\Gamma \vdash_v \lfloor v \rfloor : \alpha\}_a(\vec{v}') &= v \\ \{\Gamma \vdash_v \lfloor v \rfloor : \vec{\sigma} \Rightarrow \vec{\tau}\}_a(\vec{v}') &= !\kappa^{-1}; ?v. \kappa \\ \{\Gamma \vdash_v !M : \vec{\sigma} \Rightarrow \vec{\tau}\}_a(\vec{v}) &= !\{\Gamma \vdash_v M : \vec{\sigma} \Rightarrow \vec{\tau}\}_a(\vec{v}) \\ \{\Gamma \vdash_c \star : \vec{\tau} \Rightarrow \vec{\tau}\}_a(\vec{v}) &= \star \\ \{\Gamma \vdash_c ?V. M : \vec{\rho} \Rightarrow \vec{\tau}\}_a(\vec{v}) &= ?\{\Gamma \vdash_v V : \vec{\rho} \Rightarrow \vec{\sigma}\}_a(\vec{v}). \{\Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}\}_a(\vec{v}) \\ \{\Gamma \vdash_c \langle x \rangle. M : \vec{\rho} \Rightarrow \vec{\tau}\}_a(\vec{v}) &= a\langle x' \rangle. \{x : \rho, \Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}\}_a(x', \vec{v}) \\ \{\Gamma \vdash_c [V]. M : \vec{\sigma} \Rightarrow \vec{\tau}\}_a(\vec{v}) &= a[\{\Gamma \vdash_v V : \rho\}(\vec{v})]. \{\Gamma \vdash_c M : \vec{\rho} \Rightarrow \vec{\tau}\}_a(\vec{v}) \\ \{\Gamma \vdash_c \lfloor c \rfloor. M : \vec{\rho} \Rightarrow \vec{\tau}\}_a(\vec{v}) &= \kappa^{-1}(\vec{\rho}_A). c; \kappa(\vec{\sigma}_A); \{\Gamma \vdash_c M : \vec{\sigma} \Rightarrow \vec{\tau}\}_a(\vec{v})\end{aligned}$$

In each case, we have  $\vec{v}, \vec{v'} : \{\Gamma\}$ ,  $\vec{w} : \{\Delta\}$ ,  $y : \{\tau\}$  and  $x' : \{\rho\}$ . We will often omit the input, or part of the input, when the meaning is unambiguous, especially when it is unchanging throughout a proof. We omit the context and/or types of terms inside the interpretation function when it is clear. From now on, without loss of generality, we will choose to embed into the *main* location, *i.e.*, let  $a = \lambda$  of the !FMC, which will be omitted from terms. Similarly, we will omit the location  $a$  from the embedding  $\{-\}_a$ , which we write as simply  $\{-\}$ .

**Theorem 5.2.7.** *The category !SLC( $[\Sigma_A]$ ) is a Cartesian closed category and the embedding  $\{-\}_a : \text{!SLC}([\Sigma_A]) \rightarrow \text{!FMC}(\Sigma_A)$  is a CCC-functor.*

*Proof.* That !SLC( $\Sigma_A$ ) is a CCC is immediate from Theorem 4.3.7, applied in the case of  $A = \{\lambda\}$ . It is also immediate from the definition of !SLC( $\Sigma_A$ ) that the unit type, identity, products and exponentials are all preserved by embedding. Further, equality of terms is immediately preserved, as constants are not mentioned in any of the unit equations.

We must further show that sequencing (composition) is preserved, *i.e.*, that  $\{M; N\} = \{M\}; \{N\}$ . We proceed by induction on the type derivation of  $N$ , in the manner of Lemma 5.1.10, from the previous subsection. All cases are immediate here, though, except for the case of the sequential constant, since  $\{-\}$  is simply inclusion of terms. The case of the sequential constant is detailed below.

$$\begin{aligned} \{(c.M); N\} &= \{c.(M; N)\} \\ (\text{defn. } \{-\}) &= \kappa^{-1}; c.\kappa.\{M; N\} \\ (\text{I.H.}) &= \kappa^{-1}; c.\kappa.\{M\}; \{N\} \\ (\text{defn. } \{-\}) &= \{c.M\}; \{N\} \end{aligned}$$

Overall, this gives that the embedding is a well-defined CCC-functor.  $\square$

**Corollary 5.2.8.** *The embedding  $\{-\}_a$  maps the exponent  $(\rightarrow, \epsilon)$  of the !SLC( $[\Sigma_A]$ ) to  $(\xrightarrow{a}, \epsilon^a)$  of !FMC( $\Sigma_A$ ).*

In the next section, we construct what will prove to be the inverse functor.

### 5.2.2 Collapsing the Memory: !FMC to !SLC

The definition of  $\llbracket - \rrbracket$ , which is defined on !FMC-types, is now extended to !FMC-terms. We first extend this to our notation for families of variables.

**Notation 5.2.9.** Given a memory  $S_A = \{S_a \mid a \in A\}$ , family of variables  $\vec{x}_A = \{\vec{x}_{a_i} \mid a_i \in A\}$  and memory type  $\vec{\tau}_A = \{\vec{\tau}_{a_i} \mid a_i \in A\}$ , we use the following notation.

$$\llbracket S_A \rrbracket = S_{a_1} \cdots S_{a_m} \quad \llbracket \vec{x}_A \rrbracket = \vec{x}_{a_1} \cdots \vec{x}_{a_m} \quad \llbracket \vec{\tau}_A \rrbracket = \vec{\tau}_{a_1} \cdots \vec{\tau}_{a_m}$$

We will then use the following notation for applications and abstractions.

$$\llbracket [S_A] \rrbracket.M = [S_{a_1} \cdots S_{a_m}].M \quad \langle \llbracket \vec{x}_A \rrbracket \rangle.M = \langle \vec{x}_{a_m} \cdots \vec{x}_{a_1} \rangle.M.$$

Note that, in the latter case, we reverse the vector  $\lfloor \vec{x}_A \rfloor$ , rather than reversing  $\vec{x}_A$  and then applying  $\lfloor - \rfloor$ .

The following *collapse interpretation* can be understood as follows. The interpretation of application (and abstraction) must be given by an application (abstraction), except instead of acting on the head of one of the many stacks, they must now act on the same value, but which may now be somewhere deep in the single stack. We interpret application then as a sequence of abstractions which pick up the entire stack, followed by a sequence of applications, which replaces the stack, but with an extra application inserting the new value in the appropriate in the stack. Similarly, an abstraction is interpreted as a sequence of abstractions which pick up the whole stack, followed by a sequence of applications, which replaces all but the variable to be bound. This variable is then the only one which remains free in the remaining term.

The interpretation of sequencing  $N; M$  of  $N: \tilde{\rho}_A \Rightarrow \vec{v}_A$  and  $M: \tilde{v}_A \tilde{\sigma}_A \Rightarrow \vec{\tau}_A$  is given below. This also illustrates how the cases of sequential constant and sequential execution are interpreted.

Note that, unlike with the nominal string diagrams we used earlier, here coloured wires indicate what *used to be* different locations, which have now been collapsed to a single stack. Note that the input  $\lfloor \vec{\sigma}_A \vec{\rho}_A \rfloor = \vec{\sigma}_{a_m} \vec{\rho}_{a_m} \dots \vec{\sigma}_{a_1} \vec{\rho}_{a_1}$ . We will make the interpretation of sequencing formal in the next subsection, but give it here for illustrative purposes.

**Definition 5.2.10.** Define the *collapse interpretation*  $\llbracket - \rrbracket$  on contexts as  $\llbracket \tau_1, \dots, \tau_n \rrbracket = \llbracket \tau_1 \rrbracket, \dots, \llbracket \tau_n \rrbracket$  and the *collapse* of !FMC-terms over  $\Sigma_A$  into !SLC-terms over  $\lfloor \Sigma_A \rfloor$

$$\begin{aligned} \llbracket \Gamma \rrbracket \vdash_c \llbracket \Gamma \vdash_c M: \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket : \llbracket \tilde{\sigma}_A \rrbracket \Rightarrow \llbracket \vec{\tau}_A \rrbracket \\ \llbracket \Gamma \rrbracket \vdash_v \llbracket \Gamma \vdash_c V: \tau \rrbracket : \llbracket \tau \rrbracket \end{aligned}$$

by mutual induction over type derivations of computatons and values, as follows:

$$\begin{aligned} \llbracket \Gamma, x: \tau, \Delta \vdash_v x: \tau \rrbracket(\vec{u}, y, \vec{v}) &= y \\ \llbracket \Gamma \vdash_v v: \tau \rrbracket(\vec{w}) &= \lfloor v \rfloor \\ \llbracket \Gamma \vdash_v !M: \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(\vec{v}) &= !\llbracket \Gamma \vdash_c M: \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(\vec{v}) \\ \llbracket \Gamma \vdash_c \star: \tilde{\sigma}_A \Rightarrow \vec{\sigma}_A \rrbracket(\vec{v}) &= \star \\ \llbracket \Gamma \vdash_c c.M: \tilde{\rho}_A \tilde{\sigma}_A \Rightarrow \vec{\sigma}_A \rrbracket(\vec{v}) &= \langle \lfloor \tilde{x}_A \tilde{y}_A \rfloor \rangle. \lfloor \tilde{x}_A \rfloor. \lfloor c \rfloor. \langle \lfloor \tilde{z}_A \rfloor \rangle. \lfloor \tilde{y}_A \tilde{z}_A \rfloor. \\ &\quad \llbracket \Gamma \vdash_c M: \tilde{\rho}_A \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(\vec{v}) \\ \llbracket \Gamma \vdash_c [V]a_n.M: \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(\vec{v}) &= \langle \lfloor \tilde{x}_A \rfloor \rangle. \lfloor \tilde{x}_A a_n(y) \rfloor. \llbracket \Gamma \vdash_c M: a_n(\rho) \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(\vec{v}) \\ \llbracket \Gamma \vdash_c a_n(x).M: a_n(\rho) \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(\vec{v}) &= \langle \lfloor a_n(x') \tilde{y}_A \rfloor \rangle. \lfloor \tilde{y}_A \rfloor. \llbracket x, \Gamma \vdash_c M: \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(x', \vec{v}) \\ \llbracket \Gamma \vdash_c ?V.M: \tilde{\rho}_A \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(\vec{v}) &= \langle \lfloor \tilde{x}_A \tilde{y}_A \rfloor \rangle. \lfloor \tilde{x}_A \rfloor. ?\llbracket \Gamma \vdash_v V: \tilde{\rho}_A \Rightarrow \vec{v}_A \rrbracket(\vec{v}). \\ &\quad \langle \lfloor \tilde{z}_A \rfloor \rangle. \lfloor \tilde{y}_A \tilde{z}_A \rfloor. \llbracket \Gamma \vdash_c M: \tilde{v}_A \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket(\vec{v}) \end{aligned}$$

where, in the application case,  $y = \llbracket \Gamma \vdash_v V: \rho \rrbracket(\vec{v})$ , and throughout  $\vec{v}: \Gamma$  and  $\vec{w}: \Delta$ . We further have  $\lfloor \tilde{y}_A \tilde{z}_A \rfloor: \llbracket \tilde{\sigma}_A \tilde{\rho}_A \rrbracket$ ,  $\lfloor \tilde{x}_A \rfloor: \llbracket \tilde{\sigma}_A \rrbracket$ ,  $\lfloor \tilde{y}_A \rfloor: \llbracket \tilde{\sigma}_A \rrbracket$  and  $\lfloor \tilde{y}_A \tilde{z}_A \rfloor: \llbracket \tilde{\sigma}_A \tilde{\rho}_A \rrbracket$

in, respectively, the sequential constant, application, abstraction and sequential execution cases. We will sometimes omit the input, or part of the input, when the meaning is unambiguous, especially when it is unchanging throughout a proof. We omit the context and/or types of terms inside the interpretation function when it is clear. Note, this map is implicitly parameterized in terms of an ordering  $<$  on locations.

We proceed to show that this interpretation preserves equality of terms.

### 5.2.3 Soundness of the Collapse of the Memory

The purpose of this subsection is to show that the collapse interpretation is a well-defined functor. We first give the derived interpretations of our defined notation  $\llbracket \vec{x} \rrbracket$  for application and abstraction.

**Lemma 5.2.11** (Application and Abstraction). *We have for every computation:*

$$\begin{aligned} & \llbracket [\vec{y}_A : \vec{\rho}_A], \Gamma \vdash_c [\vec{y}_A]. M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket ([\vec{y}'_A], \vec{v}) \\ &= \langle [\vec{x}_A] \rangle. \llbracket [\vec{x}_A \vec{y}'_A] \rrbracket. \llbracket [\vec{y}_A : \vec{\rho}_A], \Gamma \vdash_c M : \vec{\rho}_A \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket ([\vec{y}'_A], \vec{v}) \\ & \llbracket \Gamma \vdash_c \langle \vec{x}_A \rangle. M : \vec{\rho}_A \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket (\vec{v}) \\ &= \langle [\vec{x}'_A \vec{y}_A] \rangle. \llbracket [\vec{y}_A] \rrbracket. \llbracket [\vec{x}_A : \vec{\rho}_A], \Gamma \vdash_c M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket ([\vec{x}'_A], \vec{v}) \end{aligned}$$

*Proof.* In each case, we proceed by induction on the size of  $x_A$ .

- Abstraction: For the base case, where  $x_A = a_n(x)$  is a singleton, we have by definition

$$\llbracket a_n \langle x \rangle. M \rrbracket (\vec{v}) = \langle [a_n(x') \vec{y}_A] \rangle. \llbracket [\vec{y}_A] \rrbracket. \llbracket M \rrbracket (x', \vec{v}) .$$

For the inductive case, we calculate:

$$\begin{aligned} & \llbracket a_n \langle w \rangle. \langle \vec{x}_A \rangle. M \rrbracket (\vec{v}) \\ (\text{defn. } \llbracket - \rrbracket) &= \langle [a_n(w) \vec{x}'_A \vec{y}_A] \rangle. \llbracket [\vec{y}_A \vec{x}'_A] \rrbracket. \llbracket \langle \vec{x}_A \rangle. M \rrbracket (w, \vec{v}) \\ (\text{I.H}) &= \langle [a_n(w) \vec{x}'_A \vec{y}_A] \rangle. \llbracket [\vec{y}_A \vec{x}'_A] \rrbracket. \langle [\vec{x}'_A \vec{y}'_A] \rangle. \llbracket [\vec{y}'_A] \rrbracket. \llbracket M \rrbracket ([\vec{x}'_A], w, \vec{v}) \\ &=_{\beta} \langle [a_n(w) \vec{x}'_A \vec{y}_A] \rangle. \llbracket [\vec{y}_A] \rrbracket. \llbracket M \rrbracket ([\vec{x}'_A], w, \vec{v}) \\ (\text{Exch.}) &= \langle [a_n(w) \vec{x}'_A \vec{y}_A] \rangle. \llbracket [\vec{y}_A] \rrbracket. \llbracket M \rrbracket ([\vec{x}'_A a_n(w)], \vec{v}) . \end{aligned}$$

- Application: For the base case, where  $x_A = a_n(x)$  is a singleton, we have by definition

$$\llbracket [y] a_n. M \rrbracket (x', \vec{v}) = \langle [\vec{x}_A] \rangle. \llbracket [\vec{x}_A a_n(y)] \rrbracket. \llbracket M \rrbracket (y' \vec{v}) .$$



For the inductive case, we calculate:

$$\begin{aligned}
& \llbracket [x]a_n. [\vec{y}_A]. M \rrbracket ([\vec{y}'_A a_n(x')], \vec{v}) \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [\vec{w}_A] \rangle. [\llbracket \vec{w}_A a_n(x') \rrbracket]. \llbracket [\vec{y}_A]. M \rrbracket ([\vec{y}'_A a(x')], \vec{v}) \\
(\text{I.H}) &= \langle [\vec{w}_A] \rangle. [\llbracket \vec{w}_A a_n(x') \rrbracket]. \langle [a_n(x')\vec{w}'_A] \rangle. [\llbracket \vec{w}'_A a_n(x')\vec{y}'_A \rrbracket]. \\
& \quad \llbracket M \rrbracket ([\vec{y}'_A a_n(x')], \vec{v}) \\
&=_{\beta} \langle [\vec{w}_A] \rangle. [\llbracket \vec{w}_A \vec{a}_n(x')\vec{y}'_A \rrbracket]. \llbracket M \rrbracket ([\vec{y}'_A a_n(x')], \vec{v}). \quad \square
\end{aligned}$$

The following lemma gives the interpretation of the (admissible) exchange and weakening rules.

**Lemma 5.2.12** (Exchange and Weakening). *For all computations, we have*

$$\begin{aligned}
& \llbracket \Gamma, x: \pi, y: \rho, \Gamma' \vdash_c M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket (\vec{u}, x', y', \vec{v}) = \\
& \llbracket \Gamma, y: \rho, x: \pi, \Gamma' \vdash_c M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket (\vec{u}, y', x', \vec{v})
\end{aligned}$$

$$\llbracket \Gamma \vdash_c M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket (\vec{v}) = \llbracket x: \rho, \Gamma \vdash_c M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket (x', \vec{v}),$$

where, in the latter case,  $r \notin \text{fv}(M)$

*Proof.* Induction on the type derivation of  $M$ .  $\square$

The following lemma will give the interpretation of the admissible sequencing rule, which is necessary for proving that  $\llbracket - \rrbracket$  is a well-defined functor.

**Lemma 5.2.13** (Sequencing). *For computations  $N: \vec{\rho}_A \Rightarrow \vec{v}_A$  and  $M: \vec{v}_A \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , we have*

$$\llbracket N; M \rrbracket (\vec{v}) = \langle [\vec{x}_A \vec{y}_A] \rangle. [\llbracket \vec{x}_A \rrbracket]. \llbracket N \rrbracket (\vec{v}); \langle [\vec{z}_A] \rangle. [\llbracket \vec{y}_A \vec{z}_A \rrbracket]. \llbracket M \rrbracket (\vec{v}).$$

*Proof.* We proceed by induction on the type derivation of  $N$ . We make use of the convention to omit inputs to  $\llbracket - \rrbracket$  where possible. We begin each case by unfolding the definition of sequencing.

- Identity:

$$\begin{aligned}
& \llbracket \star; M \rrbracket = \llbracket M \rrbracket \\
&=_{\text{id}} \langle [\vec{x}_A \vec{y}_A] \rangle. [\llbracket \vec{y}_A \vec{x}_A \rrbracket]. \llbracket M \rrbracket \\
&=_{\beta} \langle [\vec{x}_A \vec{y}_A] \rangle. [\llbracket \vec{x}_A \rrbracket]. \langle [\vec{x}'_A] \rangle. [\llbracket \vec{y}_A \vec{x}'_A \rrbracket]. \llbracket M \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [\vec{x}_A \vec{y}_A] \rangle. [\llbracket \vec{x}_A \rrbracket]. \llbracket \star \rrbracket; \langle [\vec{x}'_A] \rangle. [\llbracket \vec{y}_A \vec{x}'_A \rrbracket]. \llbracket M \rrbracket
\end{aligned}$$

- Sequential Constant:

$$\begin{aligned}
\llbracket c. N ; M \rrbracket &= \llbracket c. (N ; M) \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [\tilde{w}_A \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{w}_A]. [c]. \langle [\tilde{u}_A] \rangle. [\tilde{y}_A \tilde{x}_A \tilde{u}_A]. \llbracket N ; M \rrbracket \\
(\text{I.H.}) &= \langle [\tilde{w}_A \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{w}_A]. [c]. \langle [\tilde{u}_A] \rangle. [\tilde{y}_A \tilde{x}_A \tilde{u}_A]. \underline{\langle [\tilde{u}'_A \tilde{x}'_A \tilde{y}'_A] \rangle}. \\
&\quad [\tilde{x}'_A \tilde{u}'_A]. \llbracket N \rrbracket ; \langle [\tilde{z}_A] \rangle. [\tilde{y}'_A \tilde{z}_A]. \llbracket M \rrbracket \\
&=_{\beta} \langle [\tilde{w}_A \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{w}_A]. [c]. \langle [\tilde{u}_A] \rangle. [\tilde{x}_A \tilde{u}_A]. \llbracket N \rrbracket ; \\
&\quad \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket \\
&=_{\beta} \langle [\tilde{w}_A \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A \tilde{w}_A]. \underline{\langle [\tilde{w}'_A \tilde{x}'_A] \rangle}. [\tilde{w}'_A]. [c]. \langle [\tilde{u}_A] \rangle. \\
&\quad [\tilde{x}'_A \tilde{u}_A]. \llbracket N \rrbracket ; \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [\tilde{w}_A \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A \tilde{w}_A]. \llbracket c. N \rrbracket ; \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket
\end{aligned}$$

- Application: let  $w = \llbracket V \rrbracket$  in

$$\begin{aligned}
\llbracket [V]a_n. N ; M \rrbracket &= \llbracket [V]a_n. (N ; M) \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [\tilde{x}_A \tilde{y}_A] \rangle. [\tilde{y}_A \tilde{x}_A a_n(w)]. \llbracket N ; M \rrbracket \\
(\text{I.H.}) &= \langle [\tilde{x}_A \tilde{y}_A] \rangle. [\tilde{y}_A \tilde{x}_A a_n(w)]. \underline{\langle [a_n(w') \tilde{x}'_A \tilde{y}'_A] \rangle}. [\tilde{x}'_A a_n(w')]. \llbracket N \rrbracket ; \\
&\quad \langle [\tilde{z}_A] \rangle. [\tilde{y}'_A \tilde{z}_A]. \llbracket M \rrbracket \\
&=_{\beta} \langle [\tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A a_n(w)]. \llbracket N \rrbracket ; \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket \\
&=_{\beta} \langle [\tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A]. \underline{\langle [\tilde{x}'_A] \rangle}. [\tilde{x}'_A a_n(w)]. \llbracket N \rrbracket ; \\
&\quad \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [\tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A]. \llbracket [V]a_n. N \rrbracket ; \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket
\end{aligned}$$

- Abstraction:

$$\begin{aligned}
\llbracket a_n \langle u \rangle. N ; M \rrbracket &= \llbracket a_n \langle w \rangle. (N ; M) \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [a_n(w') \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{y}_A \tilde{x}_A]. \llbracket N ; M \rrbracket (w') \\
(\text{I.H.}) &= \langle [a_n(w') \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{y}_A \tilde{x}_A]. \underline{\langle [\tilde{x}'_A \tilde{y}'_A] \rangle}. [\tilde{x}'_A]. \llbracket N \rrbracket (w') ; \\
&\quad \langle [\tilde{z}_A] \rangle. [\tilde{y}'_A \tilde{z}_A]. \llbracket M \rrbracket (w') \\
&=_{\beta} \langle [a_n(w') \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A]. \llbracket N \rrbracket (w') ; \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket (w') \\
(\text{Weak.}) &= \langle [a_n(w') \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A]. \llbracket N \rrbracket (w') ; \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket \\
&=_{\beta} \langle [a_n(w') \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A a_n(w')]. \underline{\langle [a_n(w') \tilde{x}'_A] \rangle}. [\tilde{x}'_A]. \\
&\quad \llbracket N \rrbracket (w') ; \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [a_n(w') \tilde{x}_A \tilde{y}_A] \rangle. [\tilde{x}_A a_n(w')]. \llbracket a_n \langle w \rangle. N \rrbracket ; \\
&\quad \langle [\tilde{z}_A] \rangle. [\tilde{y}_A \tilde{z}_A]. \llbracket M \rrbracket
\end{aligned}$$

- Sequential Execution:

$$\begin{aligned}
\llbracket ?V. N ; M \rrbracket &= \llbracket ?V. (N ; M) \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [\vec{w}_A \vec{x}_A \vec{y}_A] \rangle. [\vec{w}_A]. \llbracket ?V \rrbracket. \langle [\vec{u}_A] \rangle. [\vec{y}_A \vec{x}_A \vec{u}_A]. \llbracket N ; M \rrbracket \\
(\text{I.H.}) &= \langle [\vec{w}_A \vec{x}_A \vec{y}_A] \rangle. [\vec{w}_A]. \llbracket ?V \rrbracket. \langle [\vec{u}_A] \rangle. [\vec{y}_A \vec{x}_A \vec{u}_A]. \langle [\vec{u}'_A \vec{x}'_A \vec{y}'_A] \rangle. \\
&\quad [\vec{x}'_A \vec{u}'_A]. \llbracket N \rrbracket ; \langle [\vec{z}_A] \rangle. [\vec{y}'_A \vec{z}_A]. \llbracket M \rrbracket \\
&=_{\beta} \langle [\vec{w}_A \vec{x}_A \vec{y}_A] \rangle. [\vec{w}_A]. \llbracket ?V \rrbracket. \langle [\vec{u}_A] \rangle. [\vec{x}_A \vec{u}_A]. \llbracket N \rrbracket ; \\
&\quad \langle [\vec{z}_A] \rangle. [\vec{y}_A \vec{z}_A]. \llbracket M \rrbracket \\
&=_{\beta} \langle [\vec{w}_A \vec{x}_A \vec{y}_A] \rangle. [\vec{x}_A \vec{w}_A]. \langle [\vec{w}'_A \vec{x}'_A] \rangle. [\vec{w}'_A]. \llbracket ?V \rrbracket. \langle [\vec{u}_A] \rangle. \\
&\quad [\vec{x}'_A \vec{u}_A]. \llbracket N \rrbracket ; \langle [\vec{z}_A] \rangle. [\vec{y}_A \vec{z}_A]. \llbracket M \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle [\vec{w}_A \vec{x}_A \vec{y}_A] \rangle. [\vec{x}_A \vec{w}_A]. \llbracket ?V. N \rrbracket ; \langle [\vec{z}_A] \rangle. [\vec{y}_A \vec{z}_A]. \llbracket M \rrbracket \quad \square
\end{aligned}$$

**Remark 5.2.14.** Note that it immediately follows from the above Lemma that  $\llbracket - \rrbracket$  respects *strict* sequencing, *i.e.*, where  $\vec{\sigma}_A = \epsilon_A$ , we have as below, left. Similarly, in the case of *strict* sequential execution we have as below, right.

$$\llbracket M ; N \rrbracket = \llbracket M \rrbracket ; \llbracket N \rrbracket \quad \llbracket ?V. M \rrbracket = ?\llbracket V \rrbracket ; \llbracket M \rrbracket$$

To see this, applying the result above with  $\vec{\sigma}_A = \epsilon_A$  and calculate

$$\llbracket M ; N \rrbracket = \langle [\vec{x}_A] \rangle. [\vec{x}_A]. \llbracket N \rrbracket ; \langle [\vec{w}_A] \rangle. [\vec{w}_A]. \llbracket M \rrbracket =_{\text{id}} \llbracket N \rrbracket ; \llbracket M \rrbracket.$$

with the result for strict sequential execution similar. We will use these results in future proofs, commenting only that they are applications of the Sequencing Lemma or the definition of  $\llbracket - \rrbracket$ , respectively.

This section is concluded by proving that the collapse interpretation is a well-defined functor from  $\text{!FMC}(\Sigma_A)$  to  $\text{!SLC}(\Sigma_A)$ .

**Proposition 5.2.15** (Soundness). *For any computations  $\Gamma \vdash_c M, N : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , we have*

$$M = N \quad \text{implies} \quad \llbracket M \rrbracket = \llbracket N \rrbracket.$$

*Proof.* We prove the statement in the case of each equation. In each case, we begin by unfolding the definition of  $\llbracket - \rrbracket$ , except where otherwise specified.

- Identity (id):

$$\begin{aligned}
\llbracket \langle [\vec{x}_A] \rangle. [\vec{x}_A] \rrbracket (\text{Abs.}) &= \langle [\vec{x}_A] \rangle. \llbracket \vec{x}_A \rrbracket (\llbracket \vec{x}_A \rrbracket) \\
(\text{App.}) &= \langle [\vec{x}_A] \rangle. [\vec{x}_A]. \llbracket \star \rrbracket (\llbracket \vec{x}_A \rrbracket) \\
(\text{Weak.}) &= \langle [\vec{x}_A] \rangle. [\vec{x}_A]. \llbracket \star \rrbracket \\
&=_{\text{id}} \llbracket \star \rrbracket
\end{aligned}$$

- Local Beta ( $\beta$ ): let  $x' = \llbracket V \rrbracket$  in

$$\begin{aligned}
\llbracket [V]a_n.a_n\langle x \rangle. ?x \rrbracket &= \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}_A a_n(x') \rrbracket]. \llbracket \langle x \rangle a_n. ?x \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}_A a_n(x') \rrbracket]. \langle \llbracket a_n(x') \tilde{y}'_A \rrbracket \rangle. [\llbracket \vec{y}'_A \rrbracket]. \llbracket ?x \rrbracket(x') \\
&=_{\beta} \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}'_A \rrbracket]. \llbracket ?x \rrbracket(x') \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}'_A \rrbracket]. ?\llbracket x \rrbracket(x') \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}'_A \rrbracket]. ?x' \\
(\text{defn. } x') &= ?\llbracket V \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket ?V \rrbracket
\end{aligned}$$

- Eta ( $\eta'$ ): (Remark 4.3.6)

$$\begin{aligned}
&\llbracket [!M; a\langle f \rangle. ?f]a \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= [!M; a\langle f \rangle. ?f] \\
(\text{Seq.}) &= [!\langle \llbracket \tilde{s}_A \rrbracket \rangle. [M]; \langle g \rangle. [\llbracket \vec{s}_A a(g) \rrbracket]; [a_n\langle f \rangle. ?f]] \\
(\text{Abs.}) &= [!\langle \llbracket \tilde{s}_A \rrbracket \rangle. [M]; \langle g \rangle. [\llbracket \vec{s}_A a(g) \rrbracket]; \langle [a(g')\tilde{s}'_A] \rangle. [\llbracket \vec{s}_A \rrbracket]. \llbracket ?f \rrbracket(g')] \\
&=_{\beta} [!\langle \llbracket \tilde{s}_A \rrbracket \rangle. [M]; \langle g \rangle. [\llbracket \vec{s}_A \rrbracket]. \llbracket ?f \rrbracket(g)] \\
(\text{defn. } \llbracket - \rrbracket) &= [!\langle \llbracket \tilde{s}_A \rrbracket \rangle. [M]; \langle g \rangle. [\llbracket \vec{s}_A \rrbracket]. ?\llbracket f \rrbracket(g)] \\
(\text{defn. } \llbracket - \rrbracket) &= [!\langle \llbracket \tilde{s}_A \rrbracket \rangle. [M]; \langle g \rangle. [\llbracket \vec{s}_A \rrbracket]. ?g] \\
&=_{\beta} [!\langle \llbracket \tilde{s}_A \rrbracket \rangle. [M]; \langle g \rangle. [\llbracket \vec{s}_A \rrbracket]. [g]. \langle h \rangle. ?h] \\
&=_{\iota} [!\langle \llbracket \tilde{s}_A \rrbracket \rangle. [\llbracket \vec{s}_A \rrbracket]. [M]; \langle h \rangle. ?h] \\
&=_{\text{id}} [!M]. \langle h \rangle. ?h \\
&=_{\eta'}' [!M]
\end{aligned}$$

- Force ( $\phi$ ):

$$\begin{aligned}
\llbracket [!M] \rrbracket &= ?\llbracket [!M] \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= ?\llbracket [!M] \rrbracket \\
&=_{\phi} \llbracket [!M] \rrbracket
\end{aligned}$$

- Thunk ( $\tau$ ):

$$\begin{aligned}
\llbracket [!V] \rrbracket &= !\llbracket [!V] \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= !\llbracket [!V] \rrbracket \\
&=_{\tau} \llbracket [!V] \rrbracket
\end{aligned}$$

- Terminal (!):

$$\begin{aligned}
\llbracket S; \langle \tilde{x}_A \rangle \rrbracket (\text{Seq.}) &= \llbracket S \rrbracket; \llbracket \langle \tilde{x}_A \rangle \rrbracket \\
(\text{Abs.}) &= \llbracket S \rrbracket; \langle \llbracket \tilde{x}'_A \rrbracket \rangle. \llbracket \star \rrbracket(\llbracket \tilde{x}'_A \rrbracket) \\
(\text{Weak.}) &= \llbracket S \rrbracket; \langle \llbracket \tilde{x}_A \rrbracket \rangle. \llbracket \star \rrbracket \\
&= ! \llbracket \star \rrbracket
\end{aligned}$$

- Diagonal ( $\Delta$ ):

$$\begin{aligned}
\llbracket S; \langle \tilde{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A] \rrbracket (\text{Seq.}) &= \llbracket S \rrbracket; \llbracket \langle \tilde{x}_A \rangle. [\vec{x}_A]. [\vec{x}_A] \rrbracket \\
(\text{Abs.}) &= \llbracket S \rrbracket; \langle \llbracket \tilde{x}'_A \rrbracket \rangle. \llbracket [\vec{x}_A]. [\vec{x}_A] \rrbracket(\llbracket \tilde{x}'_A \rrbracket) \\
(\text{App.}) &= \llbracket S \rrbracket; \langle \llbracket \tilde{x}'_A \rrbracket \rangle. [\llbracket \vec{x}_A \rrbracket]. \llbracket [\vec{x}_A] \rrbracket(\llbracket \tilde{x}'_A \rrbracket) \\
(\text{App.}) &= \llbracket S \rrbracket; \langle \llbracket \tilde{x}'_A \rrbracket \rangle. [\llbracket \vec{x}_A \rrbracket]. [\llbracket \vec{x}_A \rrbracket] \\
&=_{\Delta} \llbracket S \rrbracket; \llbracket S \rrbracket \\
(\text{Seq.}) &= \llbracket S; S \rrbracket
\end{aligned}$$

- Interchange ( $\iota$ ):

$$\begin{aligned}
&\llbracket P; \langle \tilde{y}_A \rangle. (Q; [\vec{y}_A]) \rrbracket \\
(\text{Seq.}) &= \langle \llbracket \tilde{x}_A \rrbracket \rangle. \llbracket P \rrbracket; \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{x}_A \vec{y}_A \rrbracket]. \llbracket \langle \tilde{y}_A \rangle. (Q; [\vec{y}_A]) \rrbracket \\
&= \langle \llbracket \tilde{x}_A \rrbracket \rangle. \llbracket P \rrbracket; \langle \llbracket \tilde{y}_A \rrbracket \rangle. \llbracket [\vec{x}_A \vec{y}_A] \rrbracket. \langle \llbracket \tilde{y}'_A \tilde{x}'_A \rrbracket \rangle. [\llbracket \vec{x}'_A \rrbracket]. \llbracket Q; [\vec{y}_A] \rrbracket(\llbracket \vec{y}'_A \rrbracket) \\
&=_{\beta} \langle \llbracket \tilde{x}_A \rrbracket \rangle. \llbracket P \rrbracket; \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{x}_A \rrbracket]. \llbracket Q; [\vec{y}_A] \rrbracket(\llbracket \vec{y}_A \rrbracket) \\
(\text{Seq.}) &= \langle \llbracket \tilde{x}_A \rrbracket \rangle. \llbracket P \rrbracket; \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{x}_A \rrbracket]. \llbracket Q \rrbracket(\llbracket \vec{y}_A \rrbracket); [\llbracket \vec{y}_A \rrbracket](\llbracket \vec{y}_A \rrbracket) \\
(\text{Weak.}) &= \langle \llbracket \tilde{x}_A \rrbracket \rangle. \llbracket P \rrbracket; \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{x}_A \rrbracket]. \llbracket Q \rrbracket; [\llbracket \vec{y}_A \rrbracket](\llbracket \vec{y}_A \rrbracket) \\
&= \langle \llbracket \tilde{x}_A \rrbracket \rangle. \llbracket P \rrbracket; \langle \llbracket \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{x}_A \rrbracket]. \llbracket Q \rrbracket; [\llbracket \vec{y}_A \rrbracket] \\
&=_{\iota} \langle \llbracket \tilde{x}_A \rrbracket \rangle. [\llbracket \vec{x}_A \rrbracket]. \llbracket Q \rrbracket; \llbracket P \rrbracket \\
&=_{\text{id}} \llbracket Q \rrbracket; \llbracket P \rrbracket \\
(\text{Seq.}) &= \llbracket Q; P \rrbracket
\end{aligned}$$

- Permutation ( $\pi$ ), where  $a_n < a_k$ , with the  $a_k < a_n$  case similar:

$$\begin{aligned}
\llbracket [V]a_k. a_n \langle x \rangle. M \rrbracket &= \langle \llbracket a_n(x') \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}_A a_n(x') a_k(w) \rrbracket]. \llbracket a_n \langle x \rangle. M \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \llbracket a_n(x') \tilde{y}_A \rrbracket \rangle. \llbracket [\vec{y}_A a_n(x') a_k(w)] \rrbracket. \langle \llbracket a_k(w') a_n(x') \tilde{y}'_A \rrbracket \rangle. \\
&\quad [\llbracket \vec{y}'_A a_k(w') \rrbracket]. \llbracket M \rrbracket(x') \\
&=_{\beta} \langle \llbracket a_n(x') \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}_A a_k(w) \rrbracket]. \llbracket M \rrbracket(x') \\
&=_{\beta} \langle \llbracket a_n(x') \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}_A \rrbracket]. \langle \llbracket \tilde{y}'_A \rrbracket \rangle. [\llbracket \vec{y}'_A a_k(w) \rrbracket]. \llbracket M \rrbracket(x') \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \llbracket a_n(x') \tilde{y}_A \rrbracket \rangle. [\llbracket \vec{y}_A \rrbracket]. \llbracket [V]a_k. M \rrbracket(x') \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket a_n \langle x \rangle. [V]a_k. M \rrbracket
\end{aligned}$$

where  $w = \llbracket V \rrbracket = \llbracket V \rrbracket(r)$  (by the Weakening Lemma).

- Relocation ( $\rho$ ):

$$\begin{aligned}
\llbracket [V]a. a\langle x \rangle. [x]b \rrbracket &= \llbracket [V] \rrbracket. \llbracket a\langle x \rangle. [x]b \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [V] \rrbracket. \langle x' \rangle. \llbracket [x]b \rrbracket(x') \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [V] \rrbracket. \langle x' \rangle. [x] \\
&=_{\beta} \llbracket [V] \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket [V]b \rrbracket \quad \square
\end{aligned}$$

**Corollary 5.2.16.** *The interpretation  $\llbracket - \rrbracket : \text{!FMC}(\Sigma_A) \rightarrow \text{!SLC}(\llbracket \Sigma_A \rrbracket)$  is a well-defined functor.*

*Proof.* The interpretation preserves equality of terms by Proposition 5.2.15, preserves identity by definition and preserves composition by the Sequencing Lemma 5.2.13.  $\square$

We proceed to show that this functor is the inverse of  $\{-\} : \text{!SLC}(\llbracket \Sigma_A \rrbracket) \rightarrow \text{!FMC}(\Sigma_A)$

#### 5.2.4 Equivalence

In this section, we prove the equivalence of  $\text{!FMC}(\Sigma_A)$  and  $\text{!SLC}(\llbracket \Sigma_A \rrbracket)$ . That is, we show that the embedding functor  $\{-\}_a : \text{!SLC}(\llbracket \Sigma_A \rrbracket) \rightarrow \text{!FMC}(\Sigma_A)$  is full and faithful. That is, we prove

$$\llbracket \{M\}_a \rrbracket = M \quad \{\llbracket - \rrbracket\}_a \simeq \text{id}.$$

We begin with the faithfulness, which is nearly trivial, since the embedding is essentially just the inclusion of terms: except for the case of (computation) constants. For fullness, we require working up-to a natural isomorphism, which will be given by  $\kappa$ .

The following lemma will allow us to deal with the case of constants in the proof of faithfulness. To see this, recalling the definition of the embedding functor on constants applies the isomorphisms  $\kappa$ . We show that the collapse interpretation maps these back to the identity. Note, it is easy to see the following type-checks, by noting that collapsing a type is an idempotent operation.

**Lemma 5.2.17.** *For every  $\vec{\tau}_A$ , we have  $\llbracket \kappa(\vec{\tau}_A) \rrbracket(\vec{v}) = \star$  and  $\llbracket \kappa^{-1}(\vec{\tau}_A) \rrbracket(\vec{v}) = \star$ .*

*Proof.* For each statement, we proceed by induction on the size of the (collapse of the) memory type  $\vec{\tau}_A$ . For each statement, the base case, the empty memory, is trivial. Note that the following statement holds.

$$(*) \quad \llbracket [x]^* \rrbracket(x', \vec{v}) = x' \quad \llbracket [x]^* \rrbracket(x', \vec{v}) = x'$$

This is true by definition in the case that  $\sigma = \alpha$ . In the case that  $\rho = \bar{\pi}_A \Rightarrow \bar{v}_A$  (where, in particular,  $\bar{\pi}_A$  and  $\bar{v}_A$  are smaller than  $\bar{\tau}_A$ ), see that the first statement of  $(*)$  is given by the following calculation.

$$\begin{aligned}
\llbracket [x]^* \rrbracket(x', \bar{v}) &= \llbracket !\kappa^{-1}(\bar{\pi}_A); ?x; \kappa(\bar{v}_A) \rrbracket(x', \bar{v}) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket !\kappa^{-1}(\bar{\pi}_A); ?x; \kappa(\bar{v}_A) \rrbracket(x', \bar{v}) \\
(\text{Seq.}) &= \llbracket !\kappa^{-1}(\bar{\pi}_A) \rrbracket(x', \bar{v}); \llbracket ?x \rrbracket(x', \bar{v}); \llbracket \kappa(\bar{v}_A) \rrbracket(x', \bar{v}) \\
(\text{l.H.}) &= \llbracket ?x \rrbracket(x', \bar{v}) \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket !x \rrbracket(x', \bar{v}) \\
&=_{\phi} \llbracket x \rrbracket(x', \bar{v}) \\
(\text{defn. } \llbracket - \rrbracket) &= x'
\end{aligned}$$

The second statement of  $(*)$  follows from a similar argument. With this setup, we now prove the two statements of the lemma.

For the inductive case of the first statement of the lemma, let  $x$  be the left-most element of  $\llbracket \bar{z}_A \rrbracket$ , so that for some  $\bar{y}_A$ , we have

$$\llbracket \bar{z}_A \rrbracket = x \cdot \llbracket \bar{y}_A \rrbracket.$$

Note that this implies  $\llbracket \bar{z}_A \rrbracket^* = \llbracket \llbracket \bar{z}_A \rrbracket \rrbracket_p^* = \llbracket x \cdot \llbracket \bar{y}_A \rrbracket \rrbracket_p^* = \llbracket x \rrbracket^* \cdot \llbracket \llbracket \bar{y}_A \rrbracket \rrbracket_p^* = \llbracket x \rrbracket^* \cdot \llbracket \bar{y}_A \rrbracket^*$ .

$$\begin{aligned}
\llbracket \kappa(\bar{\tau}_A) \rrbracket &= \llbracket \langle \bar{z}_A \rangle \cdot \llbracket \bar{z}_A \rrbracket^* \rrbracket \\
(\text{Abs.}) &= \langle \llbracket \bar{z}'_A \rrbracket \rangle \cdot \llbracket \llbracket \bar{z}_A \rrbracket^* \rrbracket(\llbracket \bar{z}'_A \rrbracket) \\
(\text{defn. } \llbracket \bar{z}_A \rrbracket) &= \langle \llbracket \bar{y}'_A \rrbracket \rangle \cdot \langle x' \rangle \cdot \llbracket \llbracket x \rrbracket^* \cdot \llbracket \llbracket \bar{y}_A \rrbracket^* \rrbracket(x', \llbracket \bar{y}'_A \rrbracket) \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \llbracket \bar{y}'_A \rrbracket \rangle \cdot \langle x' \rangle \cdot \llbracket \llbracket x \rrbracket^* \rrbracket(x', \llbracket \bar{y}'_A \rrbracket) \cdot \llbracket \llbracket \bar{y}_A \rrbracket^* \rrbracket(x', \llbracket \bar{y}'_A \rrbracket) \\
(*) &= \langle \llbracket \bar{y}'_A \rrbracket \rangle \cdot \langle x' \rangle \cdot \llbracket x' \rrbracket \cdot \llbracket \llbracket \bar{y}_A \rrbracket^* \rrbracket(x', \llbracket \bar{y}'_A \rrbracket) \\
&=_{\text{id}} \langle \llbracket \bar{y}'_A \rrbracket \rangle \cdot \llbracket \llbracket \bar{y}_A \rrbracket^* \rrbracket(x', \llbracket \bar{y}'_A \rrbracket) \\
(\text{Weak.}) &= \langle \llbracket \bar{y}'_A \rrbracket \rangle \cdot \llbracket \llbracket \bar{y}_A \rrbracket^* \rrbracket(\llbracket \bar{y}'_A \rrbracket) \\
(\text{Abs.}) &= \llbracket \langle \bar{y}'_A \rangle \cdot \llbracket \bar{y}_A \rrbracket^* \rrbracket \\
(\text{defn. } \kappa) &= \llbracket \kappa(\bar{\sigma}_A) \rrbracket \\
(\text{l.H.}) &= \star
\end{aligned}$$

For the inductive case of the second statement of the lemma, let  $x: \llbracket \rho \rrbracket$  be the leftmost element of  $\bar{z}: \llbracket \bar{\tau}_A \rrbracket$ , so that for some  $\bar{y}: \llbracket \bar{\sigma}_A \rrbracket$ , we have

$$\bar{z}: \llbracket \bar{\tau}_A \rrbracket = x: \llbracket \rho \rrbracket \cdot \bar{y}: \llbracket \bar{\sigma}_A \rrbracket.$$

Note that this implies  $\llbracket \bar{z} \rrbracket^* = \llbracket x\bar{y} \rrbracket^* = b(\llbracket x \rrbracket^*) \cdot \llbracket \bar{y} \rrbracket^*$ , where  $b$  is the first (with respect

to the order  $<$  on locations) non-empty location in  $\vec{\tau}_A$ .

$$\begin{aligned}
\llbracket \kappa^{-1}(\vec{\tau}_A) \rrbracket &= \llbracket \langle \vec{z} \rangle . \llbracket [\vec{z}]^* \rrbracket \\
(\text{Abs.}) &= \langle \vec{z} \rangle . \llbracket \llbracket [\vec{z}]^* \rrbracket (\vec{z}') \\
(\text{defn. } \llbracket \vec{z}_A \rrbracket) &= \langle \vec{y}' \rangle . \langle x' \rangle . \llbracket \llbracket [x]^* \rrbracket b . \llbracket [\vec{y}]^* \rrbracket (x', \vec{y}') \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \vec{y}' \rangle . \langle x' \rangle . \llbracket \llbracket [x]^* \rrbracket (x', \vec{y}') \rrbracket . \llbracket \llbracket [\vec{y}]^* \rrbracket (x', \vec{y}') \\
(*) &= \langle \vec{y}' \rangle . \langle x' \rangle . [x'] . \llbracket \llbracket [\vec{y}]^* \rrbracket (x', \vec{y}') \\
&=_{\text{id}} \langle \vec{y}' \rangle . \llbracket \llbracket [\vec{y}]^* \rrbracket (x', \vec{y}') \\
(\text{Weak.}) &= \langle \vec{y}' \rangle . \llbracket \llbracket [\vec{y}]^* \rrbracket (\vec{y}') \\
(\text{Abs.}) &= \llbracket \langle \vec{y}' \rangle . \llbracket [\vec{y}]^* \rrbracket \\
(\text{defn. } \kappa^{-1}) &= \llbracket \kappa^{-1}(\vec{\sigma}_A) \rrbracket \\
(\text{l.H.}) &= \star
\end{aligned}$$

□

**Proposition 5.2.18** (Completeness of Embedding). *The embedding functor  $\{-\}_a : \text{!SLC}(\llbracket \Sigma_A \rrbracket) \rightarrow \text{!FMC}(\Sigma_A)$  is faithful.*

*Proof.* We have for any closed !SLC-term  $M : \vec{\sigma} \Rightarrow \vec{\tau}$ , that

$$\llbracket \{M\}_a \rrbracket = M.$$

Proceed by induction on the type derivation of  $M$ . The only non-trivial cases are for higher-order value constants, sequential constants, and sequential execution, which we detail below, taking  $a = \lambda$  without loss of generality.

- Value constant ( $v : \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ ):

$$\begin{aligned}
\llbracket \{v\} \rrbracket &= \llbracket \kappa^{-1} ; v . \kappa \rrbracket \\
(\text{Seq.}) &= \llbracket \kappa^{-1} \rrbracket ; \llbracket v \rrbracket ; \llbracket \kappa \rrbracket \\
(\text{Lem. 5.2.17}) &= \llbracket v \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \llbracket v \rrbracket
\end{aligned}$$

- Sequential constant:

$$\begin{aligned}
\llbracket \{c . M\} \rrbracket &= \llbracket \kappa^{-1} ; [c] . \kappa ; \{M\} \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \vec{x}' \rangle . [\vec{x}] . \llbracket \kappa^{-1} ; [c] . \kappa \rrbracket . \langle \vec{z} \rangle . [\vec{y} \vec{z}] . \llbracket \{M\} \rrbracket \\
(\text{Seq.}) &= \langle \vec{x}' \rangle . [\vec{x}] . \llbracket \kappa^{-1} \rrbracket ; \llbracket [c] \rrbracket . \llbracket \kappa \rrbracket . \langle \vec{z} \rangle . [\vec{y} \vec{z}] . \llbracket \{M\} \rrbracket \\
(\text{Lem. 5.2.17}) &= \langle \vec{x}' \rangle . [\vec{x}] . \llbracket [c] \rrbracket . \langle \vec{z} \rangle . [\vec{y} \vec{z}] . \llbracket \{M\} \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \vec{x}' \rangle . [\vec{x}] . c . \langle \vec{z} \rangle . [\vec{y} \vec{z}] . \llbracket \{M\} \rrbracket \\
&=_{\iota} \langle \vec{x}' \rangle . [\vec{x} \vec{y}] . c . \llbracket \{M\} \rrbracket \\
(\text{l.H.}) &= c . M
\end{aligned}$$



- Sequential Execution:

$$\begin{aligned}
\llbracket \{?V.M\} \rrbracket &= \llbracket \{?V\}. \{M\} \rrbracket \\
(\text{defn. } \llbracket - \rrbracket) &= \langle \overrightarrow{xy}. [\overrightarrow{x}]. ?\llbracket \{V\} \rrbracket. \langle \overrightarrow{z} \rangle. [\overrightarrow{yz}]. \llbracket \{M\} \rrbracket \rangle \\
&=_{\iota} \langle \overrightarrow{xy}. [\overrightarrow{xy}]. ?\llbracket \{V\} \rrbracket. \llbracket \{M\} \rrbracket \rangle \\
&=_{\text{id}} ?\llbracket \{V\} \rrbracket. \llbracket \{M\} \rrbracket \\
(\text{I.H.}) &= ?V.M
\end{aligned}$$

□

**Theorem 5.2.19** (Fullness of Embedding). *The embedding functor  $\llbracket - \rrbracket_a : \text{!SLC}(\llbracket \Sigma_A \rrbracket) \rightarrow \text{!FMC}(\Sigma_A)$  is full.*

*Proof.* We aim to show that  $\{\llbracket - \rrbracket\}_a \simeq \text{id} : \text{!FMC}(\Sigma_A) \rightarrow \text{!FMC}(\Sigma_A)$ , with the natural isomorphism given by  $\kappa$  and its inverse  $\kappa^{-1}$  at each memory type. In other words, we must show that the following diagram commutes, for every closed !FMC-term  $M : \overrightarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A$ . We will omit the location on  $\llbracket - \rrbracket_a$ , taking, without loss of generality,  $a = \lambda$ , so we can also omit the location on terms.

$$\begin{array}{ccc}
\overrightarrow{\sigma}_A & \xrightarrow{M} & \overrightarrow{\tau}_A \\
\kappa \downarrow & & \downarrow \kappa \\
\{\llbracket \overrightarrow{\sigma}_A \rrbracket\} & \xrightarrow{\llbracket M \rrbracket} & \{\llbracket \overrightarrow{\tau}_A \rrbracket\}
\end{array}$$

We will prove a stronger statement: for *open* !FMC-terms, where  $\overrightarrow{u}, \overrightarrow{w} : \overrightarrow{\omega}$  and  $\overrightarrow{v} : \llbracket \overrightarrow{\omega} \rrbracket$  and  $\sigma = \{\llbracket \overrightarrow{u} \rrbracket_p^* / \overrightarrow{w}\}$ ,

$$\begin{aligned}
&\kappa ; \{\overrightarrow{v} \vdash_c \llbracket \overrightarrow{w} \vdash_c M : \overrightarrow{\rho}_A \Rightarrow \overrightarrow{\sigma}_A \rrbracket(\overrightarrow{v})\}(\overrightarrow{u}) ; \kappa^{-1} = \sigma M \\
&\{\overrightarrow{v} \vdash_v \llbracket \overrightarrow{w} \vdash_v V : \overrightarrow{\tau}_A \Rightarrow \overrightarrow{\upsilon}_A \rrbracket(\overrightarrow{v})\}(\overrightarrow{u}) = !\kappa^{-1}(\overrightarrow{\tau}_A) ; ?\sigma V ; \kappa(\overrightarrow{\tau}_A) \\
&\{\overrightarrow{v} \vdash_v \llbracket \overrightarrow{w} \vdash_v V : \alpha \rrbracket(\overrightarrow{v})\}(\overrightarrow{u}) = \sigma V
\end{aligned}$$

and in particular, we take the valuation (input) to hold only variables. Note that when applied to closed computations, this gives the required result. We proceed by induction on type derivations. We begin each case by unfolding the definitions of  $\llbracket - \rrbracket$ , and we immediately apply  $\{-\}$  whenever possible (when its action is just the obvious inclusion of terms). Note that we make use of the fact that  $[\overrightarrow{x}_A \overrightarrow{y}_A]^* = \llbracket [\overrightarrow{x}_A]_p^* [\overrightarrow{y}_A]_p^* \rrbracket$ , where  $\llbracket - \rrbracket_p^*$  is defined to be the pointwise application of  $\llbracket - \rrbracket^*$  to a family of variables.

- Variable ( $z : \alpha$ ):

$$\begin{aligned}
\{\overrightarrow{v}, y, \overrightarrow{v}' \vdash_v \llbracket z \rrbracket(\overrightarrow{v}, y, \overrightarrow{v}')\}(\overrightarrow{u}, x, \overrightarrow{u}') &= \{\overrightarrow{v}, y, \overrightarrow{v}' \vdash_v y\}(\overrightarrow{u}, x, \overrightarrow{u}') \\
(\text{defn. } \{-\}) &= x \\
(\text{defn. subst.}) &= \{\llbracket \overrightarrow{u}, x, \overrightarrow{u}' \rrbracket_p^* / \overrightarrow{w}, z, \overrightarrow{w}'\}z
\end{aligned}$$

- Variable ( $z: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ ):

$$\begin{aligned}
\{\vec{v}, y, \vec{v}' \vdash_v \llbracket z \rrbracket(\vec{v}, y, \vec{v}')\}(\vec{u}, x, \vec{u}') &= \{\vec{v}, y, \vec{v}' \vdash_v y\}(\vec{u}, x, \vec{u}') \\
(\text{defn. } \{-\}) &= x \\
&=_{\tau} !?!?x \\
(\text{Iso.}) &= !\kappa^{-1}?!; \kappa; ?x. \kappa^{-1}; \kappa \\
(\text{defn. subst}) &= !\kappa^{-1}; ?\{!\kappa; ?x. \kappa^{-1}/z\}z.; \kappa \\
(\text{defn. } \lfloor - \rfloor^*) &= !\kappa^{-1}; ?\{[x]^*/z\}z; \kappa \\
(\text{defn. subst}) &= !\kappa^{-1}; ?\{[\vec{u}, x, \vec{u}']_p^*/\vec{w}, z, \vec{w}'\}z; \kappa
\end{aligned}$$

- Value Constant ( $v: \alpha$ ):

$$\begin{aligned}
\{\vec{v}' \vdash_v \llbracket v \rrbracket(\vec{v}')\}(\vec{u}) &= \{\vec{v}' \vdash_v v\}(\vec{u}) \\
(\text{defn. } \{-\}) &= v \\
(\text{defn. subst.}) &= \{[\vec{u}]_p^*/\vec{w}\}v
\end{aligned}$$

- Value Constant ( $v: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$ ):

$$\begin{aligned}
\{\vec{v}' \vdash_v \llbracket v \rrbracket(\vec{v}')\}(\vec{u}) &= \{\vec{w} \vdash_v \lfloor v \rfloor\}(\vec{u}) \\
(\text{defn. } \{-\}) &= !\kappa^{-1}; ?v; \kappa \\
(\text{defn. subst.}) &= !\kappa^{-1}; ?\{[\vec{w}]^*/\vec{w}\}v; \kappa
\end{aligned}$$

- Thunk:

$$\begin{aligned}
\{\vec{v} \vdash_v \llbracket !M \rrbracket(\vec{v})\}(\vec{u}) &= \{\vec{v} \vdash_v !\llbracket M \rrbracket(\vec{v})\}(\vec{u}) \\
(\text{defn. } \{-\}) &= !\{\vec{v} \vdash_v \llbracket M \rrbracket(\vec{v})\}(\vec{u}) \\
\text{I.H.} &= !\kappa^{-1}; \{[\vec{v}]_p^*/\vec{w}\}M; \kappa \\
&=_{\phi} !\kappa^{-1}; ?!\{[\vec{v}]_p^*/\vec{w}\}M; \kappa
\end{aligned}$$

- Identity:

$$\begin{aligned}
\kappa; \{\llbracket \star \rrbracket(\vec{v})\}(\vec{u}); \kappa^{-1} &= \kappa; \{\star\}(\vec{u}); \kappa^{-1} \\
(\text{defn. } \{-\}) &= \kappa; \star; \kappa^{-1} \\
(\text{defn. subst.}) &= \kappa^{-1}; \{[\vec{v}]_p^*/\vec{w}\}\star; \kappa
\end{aligned}$$

- Sequential Constant: let  $\sigma = \{\lceil \vec{v} \rceil_p^* / \vec{w} \}$  in

$$\begin{aligned}
& \kappa ; \{ \llbracket c.M \rrbracket(\vec{v}) \}(\vec{u}) ; \kappa^{-1} \\
(\text{defn. } \llbracket - \rrbracket, \{-\}) &= \kappa ; \langle \llbracket \vec{x}_A \vec{y}_A \rrbracket \rangle . \llbracket \vec{x}_A \rrbracket . ? \{ \llbracket c \rrbracket(\vec{v}) \}(\vec{u}) . \langle \llbracket \vec{z}_A \rrbracket \rangle . \\
& \quad \llbracket \vec{y}_A \vec{z}_A \rrbracket . \{ \llbracket M \rrbracket(\vec{v}) \}(\vec{u}) ; \kappa^{-1} \\
(\text{defn. } \llbracket - \rrbracket, \{-\}) &= \kappa . \langle \llbracket \vec{x}_A \vec{y}_A \rrbracket \rangle . \llbracket \vec{x}_A \rrbracket . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \\
& \quad \llbracket \vec{y}_A \vec{z}_A \rrbracket . \kappa^{-1} ; \sigma M ; \kappa ; \kappa^{-1} \\
(\text{Iso.}) &= \kappa . \langle \llbracket \vec{x}_A \vec{y}_A \rrbracket \rangle . \llbracket \vec{x}_A \rrbracket . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \\
& \quad \llbracket \vec{y}_A \vec{z}_A \rrbracket . \kappa^{-1} ; \sigma M \\
(\text{defn. } \kappa^{-1}) &= \kappa . \langle \llbracket \vec{x}_A \vec{y}_A \rrbracket \rangle . \llbracket \vec{x}_A \rrbracket . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \\
& \quad \llbracket \vec{y}_A \vec{z}_A \rrbracket . \langle \vec{s} \rangle . \llbracket \vec{s} \rrbracket^* . \sigma M \\
&=_{\beta} \kappa . \langle \llbracket \vec{x}_A \vec{y}_A \rrbracket \rangle . \llbracket \vec{x}_A \rrbracket . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}_A \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
&=_{\alpha} \kappa . \langle \llbracket \vec{x}'_A \vec{y}'_A \rrbracket \rangle . \llbracket \vec{x}'_A \rrbracket . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}'_A \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
(\text{defn. } \kappa) &= \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{y}_A \vec{x}_A \rrbracket^* . \langle \llbracket \vec{x}'_A \vec{y}'_A \rrbracket \rangle . \llbracket \vec{x}'_A \rrbracket . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \\
& \quad \llbracket \llbracket \vec{y}'_A \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
&=_{\beta} \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{x}_A \rrbracket^* . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
(\text{I.H.}) &= \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{x}_A \rrbracket^* . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
&=_{\phi} \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{x}_A \rrbracket^* ; . \kappa^{-1} ; c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
(\text{defn. } \kappa^{-1}) &= \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{x}_A \rrbracket^* . \langle \llbracket \vec{x}'_A \rrbracket \rangle . \llbracket \vec{x}'_A \rrbracket^* . c . \kappa ; \\
& \quad \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
&=_{\beta} \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \llbracket \vec{x}_A \rrbracket^* \rrbracket^* . c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
(\text{Iso.}) &= \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{x}_A \rrbracket . c . \kappa ; \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
(\text{defn. } \kappa) &= \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{x}_A \rrbracket . c . \langle \vec{z}'_A \rangle . \llbracket \vec{z}'_A \rrbracket^* . \langle \llbracket \vec{z}_A \rrbracket \rangle . \llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^* \rrbracket . \sigma M \\
&=_{\beta} \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{x}_A \rrbracket . c . \langle \vec{z}'_A \rangle . \llbracket \llbracket \vec{y}_A \rrbracket_p^* \llbracket \vec{z}'_A \rrbracket_p^* \rrbracket^* \rrbracket . \sigma M \\
(\text{Iso.}) &= \langle \vec{x}_A \vec{y}_A \rangle . \llbracket \vec{x}_A \rrbracket . c . \langle \vec{z}'_A \rangle . \llbracket \vec{y}_A \vec{z}'_A \rrbracket . \sigma M \\
(\text{Perm.}) &=_{\pi_{\text{id}}} \langle \vec{y}_A \rangle . c . \llbracket \vec{y}_A \rrbracket . \sigma M \\
&=_{\iota} c . \sigma M \\
(\text{defn. subst.}) &= \sigma c . M
\end{aligned}$$

- Application: let  $\sigma = \{\lceil \vec{v} \rceil^* / \vec{w} \}$  in

$$\begin{aligned}
& \kappa ; \{ \llbracket [V]a_n.M \rrbracket(\vec{v}) \}(\vec{u}) ; \kappa^{-1} \\
(\text{defn. } \llbracket - \rrbracket, \{-\}) &= \kappa ; \langle \lfloor \vec{x}_A \rfloor \rangle. \llbracket \vec{x}_A a(y) \rrbracket. \{ \llbracket M \rrbracket(\vec{v}) \}(\vec{u}) ; \kappa^{-1} \\
(\text{defn. } \kappa) &= \langle \vec{x}_A \rangle. \llbracket \vec{x}_A \rrbracket^*. \langle \lfloor \vec{x}_A \rfloor \rangle. \llbracket \vec{x}_A a(y) \rrbracket. \{ \llbracket M \rrbracket(\vec{v}) \}(\vec{u}) ; \kappa^{-1} \\
&=_{\beta} \langle \vec{x}_A \rangle. \llbracket \lfloor \vec{x}_A \rfloor_p^* a_n(y) \rrbracket. \{ \llbracket M \rrbracket(\vec{v}) \}(\vec{u}) ; \kappa^{-1} \\
(\text{I.H.}) &= \langle \vec{x}_A \rangle. \llbracket \lfloor \vec{x}_A \rfloor_p^* a_n(y) \rrbracket. \kappa^{-1}. \sigma M ; \kappa ; \kappa^{-1} \\
(\text{Iso.}) &= \langle \vec{x}_A \rangle. \llbracket \lfloor \vec{x}_A \rfloor_p^* a_n(y) \rrbracket. \kappa^{-1}. \sigma M \\
(\text{defn. } \kappa^{-1}) &= \langle \vec{x}_A \rangle. \llbracket \lfloor \vec{x}_A \rfloor_p^* a_n(y) \rrbracket. \langle \vec{z} \rangle. \llbracket \vec{z} \rrbracket^*. \sigma M \\
(*) &=_{\beta} \langle \vec{x}_A \rangle. \llbracket \{\sigma V / y'\} \lceil \vec{x}_A a_n(y') \rceil^* \rrbracket. \sigma M \\
(\text{Iso.}) &= \langle \vec{x}_A \rangle. \llbracket \vec{x}_A a_n(\sigma V) \rrbracket. \sigma M \\
&=_{\text{id}} \llbracket \sigma V \rrbracket a_n. \sigma M \\
(\text{defn. subst.}) &= \sigma[V]a_n.M
\end{aligned}$$

where  $y = \{ \llbracket V \rrbracket \}(\vec{u})$  and  $(*)$  requires to note that, by inductive hypothesis, we have  $y = \{\sigma V / y'\} \lfloor y' \rfloor^*$ . To see this, consider the two cases of the IH on values, and the two cases of the definition of  $\lfloor y' \rfloor$ .

- Abstraction: let  $\sigma = \{\lceil \vec{v} \rceil_p^* / \vec{w} \}$  in

$$\begin{aligned}
& \kappa ; \{ \llbracket a_n \langle x \rangle. M \rrbracket(\vec{v}) \}(\vec{u}) ; \kappa^{-1} \\
(\text{defn. } \llbracket - \rrbracket, \{-\}) &= \kappa ; \langle \lfloor a_n(x') \hat{y}_A \rfloor \rangle. \llbracket \vec{y}_A \rrbracket. \{ \llbracket M \rrbracket(y', \vec{v}) \}(x', \vec{u}) ; \kappa^{-1} \\
(\text{defn. } \kappa) &= \langle a_n(x') \hat{y}_A \rangle. \llbracket \vec{y}_A a_n(x') \rrbracket^*. \langle \lfloor a_n(x') \hat{y}'_A \rfloor \rangle. \llbracket \vec{y}'_A \rrbracket. \\
& \quad \{ \llbracket M \rrbracket(y, \vec{v}) \}(x', \vec{u}) ; \kappa^{-1} \\
&=_{\beta} \langle a_n(x') \hat{y}_A \rangle. \llbracket \vec{y}_A \rrbracket^*. \{ \llbracket M \rrbracket(y', \vec{v}) \}(\lfloor x' \rfloor^*, \vec{u}) ; \kappa^{-1} \\
(\text{I.H.}) &= \langle a_n(x') \hat{y}_A \rangle. \llbracket \vec{y}_A \rrbracket^* ; \kappa^{-1} ; \{ \lceil \lfloor x' \rfloor^* \rceil_p^* / x, \vec{w} \} M ; \kappa ; \kappa^{-1} \\
(\text{Iso.}) &= \langle a_n(x') \hat{y}_A \rangle. \llbracket \vec{y}_A \rrbracket^* ; \kappa^{-1} ; \{ x', \lceil \vec{v} \rceil_p^* / x, \vec{w} \} M ; \kappa ; \kappa^{-1} \\
&=_{\alpha} \langle a_n(x) \hat{y}_A \rangle. \llbracket \vec{y}_A \rrbracket^* ; \kappa^{-1} ; \sigma M ; \kappa ; \kappa^{-1} \\
(\text{Iso.}) &= \langle a_n(x) \hat{y}_A \rangle. \llbracket \vec{y}_A \rrbracket^* ; \kappa^{-1} ; \sigma M \\
(\text{defn. } \kappa^{-1}) &= \langle a_n(x) \hat{y}_A \rangle. \llbracket \vec{y}_A \rrbracket^*. \langle \vec{z} \rangle. \llbracket \vec{z} \rrbracket^*. \sigma M \\
&=_{\beta} \langle a_n(x) \hat{y}_A \rangle. \llbracket \lceil \vec{y}_A \rceil^* \rrbracket. \sigma M \\
(\text{Iso.}) &= \langle a_n(x) \hat{y}_A \rangle. \llbracket \vec{y}_A \rrbracket. \sigma M \\
&=_{\pi', \text{id}} a_n \langle x \rangle. \sigma M \\
(\text{defn. } \sigma, \text{ subst.}) &= \{ \lceil \vec{v} \rceil_p^* / \vec{w} \} a_n \langle x \rangle. M
\end{aligned}$$

- Sequential Execution: let  $\sigma = \{\vec{v}\}_p^* / \vec{w}$  in

$$\begin{aligned}
& \kappa; \{\llbracket !V.M \rrbracket(\vec{v})\}(\vec{u}); \kappa^{-1} \\
(\text{defn. } \llbracket - \rrbracket, \{-\}) &= \kappa; \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\{\llbracket V \rrbracket(\vec{v})\}(\vec{u}). \langle \llbracket \tilde{z}_A \rrbracket \rangle. \\
& \quad \llbracket \llbracket \vec{y}_A \vec{z}_A \rrbracket \rrbracket. \{\llbracket M \rrbracket(\vec{v})\}(\vec{u}); \kappa^{-1} \\
(\text{I.H.}) &= \kappa. \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\{\llbracket V \rrbracket(\vec{v})\}(\vec{u}). \langle \llbracket \tilde{z}_A \rrbracket \rangle. \\
& \quad \llbracket \llbracket \vec{y}_A \vec{z}_A \rrbracket \rrbracket. \kappa^{-1}; \sigma M; \kappa; \kappa^{-1} \\
(\text{Iso.}) &= \kappa. \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\{\llbracket V \rrbracket(\vec{v})\}(\vec{u}). \langle \llbracket \tilde{z}_A \rrbracket \rangle. \\
& \quad \llbracket \llbracket \vec{y}_A \vec{z}_A \rrbracket \rrbracket. \kappa^{-1}; \sigma M \\
(\text{defn. } \kappa^{-1}) &= \kappa. \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\{\llbracket V \rrbracket(\vec{v})\}(\vec{u}). \langle \llbracket \tilde{z}_A \rrbracket \rangle. \\
& \quad \llbracket \llbracket \vec{y}_A \vec{z}_A \rrbracket \rrbracket. \langle \tilde{s} \rangle. [\tilde{s}]^*. \sigma M \\
&=_{\beta} \kappa. \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\{\llbracket V \rrbracket(\vec{v})\}(\vec{u}). \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \vec{z}_A \rrbracket \rrbracket^*]. \sigma M \\
&=_{\alpha} \kappa. \langle \llbracket \tilde{x}'_A \tilde{y}'_A \rrbracket \rangle. \llbracket \tilde{x}'_A \rrbracket. ?\{\llbracket V \rrbracket(\vec{v})\}(\vec{u}). \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}'_A \vec{z}_A \rrbracket \rrbracket^*]. \sigma M \\
(\text{defn. } \kappa) &= \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \llbracket \vec{y}_A \vec{z}_A \rrbracket \rrbracket^*. \langle \llbracket \tilde{x}'_A \tilde{y}'_A \rrbracket \rangle. \llbracket \tilde{x}'_A \rrbracket. ?\{\llbracket V \rrbracket(\vec{v})\}(\vec{u}). \langle \llbracket \tilde{z}_A \rrbracket \rangle. \\
& \quad [\llbracket \llbracket \vec{y}'_A \vec{z}_A \rrbracket \rrbracket^*]. \sigma M \\
&=_{\beta} \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket^*. ?\{\llbracket V \rrbracket(\vec{v})\}(\vec{u}). \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^*]. \sigma M \\
(\text{I.H.}) &= \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket^*. ?\{\llbracket \sigma V \rrbracket\}^*. \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^*]. \sigma M \\
(\text{defn. } \llbracket - \rrbracket^*) &= \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket^*. ?!\kappa^{-1}; ?\sigma V; \kappa. \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^*]. \sigma M \\
&=_{\phi} \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket^*; \kappa^{-1}; ?\sigma V; \kappa. \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^*]. \sigma M \\
(\text{defn. } \kappa^{-1}) &= \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket^*. \langle \llbracket \tilde{x}'_A \rrbracket \rangle. [\llbracket \tilde{x}'_A \rrbracket^*]. ?\sigma V; \kappa. \langle \llbracket \tilde{z}_A \rrbracket \rangle. \\
& \quad [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^*]. \sigma M \\
&=_{\beta} \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. [\llbracket \tilde{x}_A \rrbracket^*]^*. ?\sigma V; \kappa. \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^*]. \sigma M \\
(\text{Iso.}) &= \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\sigma V; \kappa. \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^*]. \sigma M \\
(\text{defn. } \kappa) &= \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\sigma V. \langle \llbracket \tilde{z}'_A \rrbracket \rangle. [\llbracket \llbracket \vec{z}'_A \rrbracket^* \rrbracket^*]. \langle \llbracket \tilde{z}_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \vec{z}_A \rrbracket^*]. \sigma M \\
&=_{\beta} \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\sigma V. \langle \llbracket \tilde{z}'_A \rrbracket \rangle. [\llbracket \llbracket \vec{y}_A \rrbracket_p^* \llbracket \vec{z}'_A \rrbracket_p^* \rrbracket^*]. \sigma M \\
(\text{Iso.}) &= \langle \llbracket \tilde{x}_A \tilde{y}_A \rrbracket \rangle. \llbracket \tilde{x}_A \rrbracket. ?\sigma V. \langle \llbracket \tilde{z}'_A \rrbracket \rangle. \llbracket \vec{y}_A \vec{z}'_A \rrbracket. \sigma M \\
(\text{Perm.}) &=_{\pi' \text{id}} \langle \llbracket \tilde{y}_A \rrbracket \rangle. ; ?\sigma V. \llbracket \vec{y}_A \rrbracket. \sigma M \\
&=_{\iota} ?\sigma V. \sigma M \\
(\text{defn. subst.}) &= \sigma ?V.M \quad \square
\end{aligned}$$

**Theorem 5.2.20.** *The embedding functor  $\llbracket - \rrbracket_a : \text{!SLC}(\llbracket \Sigma_A \rrbracket) \rightarrow \text{!FMC}(\Sigma_A)$  gives an equivalence of categories.*

*Proof.* We have shown that embedding functor  $\llbracket - \rrbracket : \text{!SLC}(\llbracket \Sigma_A \rrbracket) \rightarrow \text{!FMC}(\Sigma_A)$  is Cartesian closed, fully faithful (Proposition 5.2.18 and Theorem 5.2.19), and is essentially surjective on objects via the isomorphisms given in Definition 5.2.4, thus we have an equivalence of categories.  $\square$

**Corollary 5.2.21.** *The categories  $!FMC(\Sigma_A)$  and  $CCC(\lfloor \Sigma_A \rfloor)$  are equivalent.*

*Proof.* Compose the equivalences:  $\{-\} : CCC(\Sigma_A) \rightarrow !SLC(\lfloor \Sigma_A \rfloor)$  and  $\lceil - \rceil_a : !SLC(\lfloor \Sigma_A \rfloor) \rightarrow !FMC(\lfloor \Sigma_A \rfloor)$  from Corollary 5.1.15 and Corollary 5.2.20.  $\square$

Given these results, an obvious question is: if the simply-typed  $!FMC$  is equivalent to the simply typed  $\lambda$ -calculus, what is to be gained from study of the  $!FMC$ ? We respond to this in two ways.

First, the results of this chapter tell us that the  $!FMC$  is *denotationally* equivalent to the  $\lambda$ -calculus: however, it is a useful *operational* refinement. The difference between the calculi lies in the fact we have gained the ability to express *how* computation takes place: how computation is sequenced, whether inputs are passed as function arguments or as mutable state, when the random generator is consulted, *etc.*

The achievement of these results is evidence of the strength of the  $!FMC$ 's type system. Simple types record all information about execution: all input and output, all changes to state, and all consultation of oracles. Indeed, strong type systems for effects is a major research objective, and the strength of the  $!FMC$ 's type system is what allows for the result. Note how, unlike other approaches to sequencing and effects, and to decompositions of CBN and CBV  $\lambda$ -calculi, no new type constructors are involved: we remain in the realm of pure intuitionistic logic.

However, as emphasized in the introduction to Chapter 4, it is emphasized that what is presented here is not in fact a semantics *of effects*, but of a calculus which can encode effects naturally. Relatedly, the type system presented here is *too strong* for dealing with effects in reality: for example, when reading from a probabilistic generator, the programmer does not typically want to know *how many times* the generator was consulted, which is currently information recorded in the type system. Indeed, such information is *intensional* and not even observable in general.<sup>5</sup>

Note that, although the *current* type system is too strong, and the current semantics does not account for effects, we believe the  $!FMC$  provides an excellent basis for future variants which address properly this major research objective. In particular, the  $!FMC$  provides valuable intuitions about how to usefully think about effectful programs, and how (certain) effects relate to higher-order mechanisms which are well-understood. The ultimate research aim is to be able to extend reasoning techniques familiar from the  $\lambda$ -calculus to real-world (especially, imperative) languages. The type system allows us to “see” side-effects, and this makes an excellent basis for future research.

---

<sup>5</sup>For example, reading from a random generator and discarding the result is observationally equivalent to doing nothing at all!

## Chapter 6

# Strong Normalisation

As a second main contribution of this thesis, we show that simply typed terms are strongly normalising with respect to beta and permutation reduction by extending Gandy’s proof for the  $\lambda$ -calculus [31]. This proof interprets terms in a domain of monotone functionals, equipped with a partial order. One may further collapse a functional to a natural number to give an overestimate of the longest reduction sequence of a term. The literature has several variants on Gandy’s proof, including one by De Vrijer that calculates longest reduction sequences exactly [22]; see also [106, 94, 93]

Our proof is a variant of Gandy’s, which emphasizes its latent operational intuition and makes use of simpler structures – rather than interpreting terms in domains of *strictly* ordered, *strict* monotone functionals, as Gandy does, we interpret terms in the more standard domain of *non-strict* monotone functionals. The author is not aware of other proofs in the literature that are formulated in this way. We discuss later the small technical difference that makes this possible.

Now, we will give an overview of this section; we will forward-reference various constructs, but will complete the formal definitions later. We will work with the variant of the FMC *without base types or values*, given in Definition 3.3.1, since base types and the distinction between computations and values is of no interest here. Note, this means that type  $\tau$  refers to a type of form  $\vec{\sigma}_A \Rightarrow \vec{\tau}_A$ , and  $(\Rightarrow)$  is the base case of an induction on types. We now recapitulate the notion of reduction we are concerned with is the following.

**Definition 6.0.1.** Beta and permutation reduction for the FMC are respectively defined by the following rules, applicable in all contexts:

$$[N]a.a\langle x \rangle.M \rightarrow_{\beta} \{N/x\}M \quad [N]a.b\langle x \rangle.M \rightarrow_{\pi} b\langle x \rangle.[N]a.M,$$

where, in the second case,  $x \notin \text{fv}(N)$  and  $a \neq b$ . We will often write simply  $\rightarrow$  for  $\rightarrow_{\beta}$ .

We will begin by showing beta reduction is strongly normalising, with strong normalisation for both beta and permutation reduction together following from a

further (simple) argument in Section 6.5. Note, by a similar argument to that section, we could also include the rewrite  $\langle x \rangle. [x] \rightarrow_{\text{id}} \star$  in our reduction relation, and prove it strongly normalising. We choose not to do this, as the property of confluence would be lost.<sup>1</sup> It is an easy exercise to transfer the results to the !FMC and its reduction relation, which includes reductions for *force* and *think*.

**Definition 6.0.2.** A term  $M$  is *strongly normalising* with respect to a reduction relation if there exists no infinite sequence of reductions beginning from  $M$ .

Similar to Gandy’s proof, we exhibit a measure function which strictly reduces with every reduction step  $\Gamma \vdash M \rightarrow_\beta M' : \tau$ . This is done by defining a quantitative semantic interpretation: we interpret the type  $\vec{\sigma}_A \Rightarrow \vec{\tau}_A$  as the set of monotone functions

$$\llbracket \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket = \llbracket \vec{\sigma}_A \rrbracket \rightarrow \mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket,$$

equipped with the extensional partial ordering (generated from  $\mathbb{N}$ , carrying its usual ordering). We note that each partial order has a least element, which we denote  $0_\tau$ .<sup>2</sup>

The interpretation then associates to each term a function taking the interpretation of its inputs to an overestimation of the length of its reduction (this is the numeric output) and the interpretation of its outputs. This interpretation of outputs is one small generalization needed from the standard Gandy proof in order to account for sequential composition of terms.

The measure on terms is extracted from the interpretation by a *collapse* function<sup>3</sup>  $\llbracket - \rrbracket_\tau : \llbracket \tau \rrbracket \rightarrow \mathbb{N}$ , defined at each type  $\tau$ , which provides a least element as input and discards the non-numeric output:

$$\llbracket f \rrbracket_\tau = \pi_1(f(0_{\vec{\rho}_A})),$$

where  $f : \tau = \vec{\rho}_A \Rightarrow \vec{\sigma}_A$  (with the definition extended element-wise for type vectors and families). We omit the subscript  $\tau$  whenever possible. The *measure function* will then be given by the composition of the interpretation and the collapse function,  $\llbracket \llbracket - \rrbracket \rrbracket : \llbracket \tau \rrbracket \rightarrow \mathbb{N}$ , and we aim to prove that

$$M \rightarrow_\beta M' : \tau \text{ implies } \llbracket \llbracket M \rrbracket \rrbracket >_{\mathbb{N}} \llbracket \llbracket M' \rrbracket \rrbracket,$$

from which strong normalisation immediately follows.

The problem of strong normalisation is more difficult than that of weak normalisation due to the possibility of reduction within discarded arguments. To illustrate

<sup>1</sup>We expect that the right way to include the identity equation in the reduction relation is to *expand* typed terms, similar to the case of  $\eta$ -expansion in the  $\lambda$ -calculus – however, consideration of this is beyond the scope of the current work.

<sup>2</sup>We prefer to avoid the use of  $\perp$  for these least elements because of its connotations of undefinedness and non-termination; the least elements of these posets are better regarded as indicating immediate termination in zero steps.

<sup>3</sup>No relation to the similarly named collapse functor of Chapter 5



this, consider a machine-based intuition for the quantitative interpretation, where the numeric output of  $\llbracket M \rrbracket(\llbracket S_A \rrbracket)$ , with  $\llbracket S_A \rrbracket$  defined pointwise on memories, measures the length of the machine run  $(S_A, M)$ . Then, given a naive interpretation function which measures only the length of the machine run (see Section 6.6), we run into the following problem: pushing an argument to the stack takes only one transition step, but that leaves redexes inside the argument unaccounted for. Indeed, if the argument is then discarded or left unused, reductions inside it will not reduce the resulting measure at all.

By adding the measure of the argument to the count as it is pushed to the stack, we can achieve a strict decrease in the measure. This “trick” of accounting for discarded terms in order to move from a proof of weak to strong normalisation is typical of Gandy-style proofs. This results in an overestimation of the number of machine steps remaining, but it means that reductions inside the argument result in reductions in the measure, as required.

The proof of strong normalisation for the FMC described above directly specializes to one for simply-typed lambda calculus, but the proof so obtained differs from previous formulations in the way in which it enforces strict monotonicity.

As stated, Gandy’s proof interprets terms in domains of *strictly* ordered, *strict* monotone functions. These domains are well-founded (because they are strict) and the interpretation of terms is such that it decreases on reduction, giving strong normalisation. This means that in the interpretation of  $\lambda x.M$ , we must then include the measure of the argument supplied to  $x$ , so that it always contributes to the overall interpretation, even if it’s discarded. This is necessary in order for the interpretation of the abstraction to be strictly monotone. In other words, the measure of an argument is added to the count not as it is pushed to the stack, but as it is popped from the stack. The literature has several further such constructions [34]. However, this approach somewhat obscures the operational intuition.

Our proof avoids the domain of *strict* functionals and instead interprets terms in the domain of (non-strict) monotone functionals. One benefit of this is that the usual domain of monotone functions is much easier to work with than the strict domains. Non-strict domains are not well-founded, but our interpretation of terms ensures that when functionals are collapsed to a natural number, this strictly decreases upon reduction, again giving strong normalisation.

Working with monotone functions is made possible because, in our proof, we account for the possibility of reduction within arguments *when they are supplied as arguments*, as opposed to when the arguments are consumed by an abstraction (as in Gandy’s proof). That is, in the interpretation of  $[N].M$ , we increment the overall measure with that for  $N$ , measuring potential reduction in  $N$  even if it will be discarded by  $M$ . An abstraction  $\langle x \rangle.M$  may then be given its natural interpretation as a standard monotone function, avoiding strictness. In fact, for the FMC, this variation is especially natural, because the problem of reduction inside arguments lies not just in arguments which are discarded, but also in certain terms not seen in the  $\lambda$ -calculus, such as  $[N].\star$ , where an argument is merely never consumed.

We now formally define the interpretation of typed terms described above. From now on, when we say ‘monotone function’, we mean *non-strict* monotone function.

## 6.1 A Quantitative Interpretation

In this section, we define the *interpretation* of the type  $\vec{\sigma}_A \Rightarrow \vec{\tau}_A$  as the set of monotonic functions  $[\![\vec{\sigma}_A]\!] \rightarrow \mathbb{N} \times [\![\vec{\tau}_A]\!]$ , equipped with the extensional partial ordering on functions and the interpretation of a term  $M : \tau$  as  $[\![M]\!] \in [\![\tau]\!]$ . We take the base domain  $\mathbb{N}$  to have the usual ordering  $\leq$ .

The following work culminates in Section 6.4, where we prove that if  $M \rightarrow N$  at type  $\tau$  then  $[\![M]\!] >_{\mathbb{N}} [\![N]\!]$ , to give strong normalisation. We will prove that top-level beta reduction reduces the measure, and that reduction inside any context reduces the measure.

Indeed, most of the following work is taken up by proving that top-level beta reduction – the base case – reduces the measure, which requires proving substitution commutes with the interpretation (Section 6.2) and that the interpretation respects the permutation of actions (Section 6.5). Proving that reduction in any context – the inductive case – reduces the measure is straightforward, excepting the one interesting case of reduction in an argument. This case necessitates proving that reduction of a term is monotonic, *i.e.*, that  $N \rightarrow N'$  implies  $[\![N]\!] \geq_{[\![\tau]\!]} [\![N']\!]$ , which is the content of Section 6.3.

**Notation 6.1.1.** We write elements of product domains as vectors  $(t_1, \dots, t_n)$ , and will elide the isomorphisms for associativity and unitality so that concatenation of  $s$  and  $t$  may be written  $(s, t)$ . If  $t \in \{T_a \mid a \in A\}$  denotes an  $A$ -indexed family of vectors, we denote by  $t_a \in T_a$  the  $a$ ’th projection of  $t$ . A *singleton*  $a(t)$ , where  $t$  is a vector and  $a \in A$ , is a family of vectors over  $A$ , such that  $a(t)_a = t$  and  $a(t)_b$  is empty for  $a \neq b$ . Concatenation lifts to indexed products pointwise:  $(s, t)_a = (s_a, t_a)$ .

**Definition 6.1.2.** The interpretation of each type  $\tau$  of the FMC is given by the set  $[\![\tau]\!]$ , equipped with the binary relation  $\leq_{[\![\tau]\!]}$ . We define these inductively on types by:

$$\begin{aligned} [\![\vec{\sigma}_A \Rightarrow \vec{\tau}_A]\!] &= \{f \in [\![\vec{\sigma}_A]\!] \rightarrow \mathbb{N} \times [\![\vec{\tau}_A]\!] \mid \forall s \leq_{[\![\vec{\sigma}_A]\!]} s'. f(s) \leq_{\mathbb{N} \times [\![\vec{\tau}_A]\!]} f(s') \} \\ f &\leq_{[\![\vec{\sigma}_A \Rightarrow \vec{\tau}_A]\!]} g \quad \text{iff} \quad \forall s \in [\![\vec{\sigma}_A]\!]. f(s) \leq_{\mathbb{N} \times [\![\vec{\tau}_A]\!]} g(s) \end{aligned}$$

with vectors and families interpreted as products and dependent products, respectively,

$$[\![\tau_1 \dots \tau_n]\!] = [\![\tau_1]\!] \times \dots \times [\![\tau_n]\!] \quad \text{and} \quad [\![\vec{\tau}_A]\!] = \prod_{a \in A} [\![\vec{\tau}_a]\!]$$

and given the element-wise ordering.

**Remark 6.1.3.** Note that, because we are working without base types, the base case of induction on types is  $\Rightarrow$ , whose interpretation is isomorphic to  $\mathbb{N}$ .

It is worth observing that for the simple types of the  $\lambda$ -calculus, as embedded in the FMC, these domains are the natural ones. Briefly, a simple type  $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow o$  embeds as the FMC-type  $\tau_1 \dots \tau_n \Rightarrow$  with the domain  $\llbracket \tau_1 \rrbracket \times \dots \times \llbracket \tau_n \rrbracket \rightarrow \mathbb{N}$ , which is the expected one modulo Curryng.

**Proposition 6.1.4.** *For all types  $\tau$ , the relation  $\leq_{\llbracket \tau \rrbracket}$  on  $\llbracket \tau \rrbracket$  is a partial order.*

*Proof.* Reflexivity, transitivity and anti-symmetry of  $\leq_{\llbracket \tau \rrbracket}$  follow by induction on types.  $\square$

For every type  $\tau$ , we give a least inhabiting element  $0_\tau$  in  $\llbracket \tau \rrbracket$ . These elements will be used in the definition of the measure, in order to extract the bound on the length of the machine run.

**Definition 6.1.5.** Define  $0_\tau \in \llbracket \tau \rrbracket$  inductively on types:

$$0_{\vec{\sigma}_A \Rightarrow \vec{\tau}_A} = s \in \llbracket \vec{\sigma}_A \rrbracket \mapsto (0, 0_{\vec{\tau}_A}),$$

where  $0_{\vec{\tau}_A}$  is defined element-wise for vectors and families.

**Remark 6.1.6.** We can see by induction on types that  $0_\tau$  is well-defined: it defines a constant function, which is thus monotonic. In fact, for all types  $\tau$ ,  $0_\tau$  is a least element of  $\llbracket \tau \rrbracket$ : the constant function which outputs a least element is itself a least element in the extensional ordering on functions.

We now define the interpretation function. First, we require to define the collapse function and a *valuation* to interpret the free variables of a term.

**Definition 6.1.7.** The *collapse function*  $\lfloor - \rfloor_\tau: \llbracket \tau \rrbracket \rightarrow \mathbb{N}$  at each type  $\tau$  is given by:

$$\lfloor f \rfloor_\tau = \pi_1(f(0_{\vec{\rho}_A})),$$

for  $f: \tau = \vec{\rho}_A \Rightarrow \vec{\sigma}_A$ . We omit the subscript  $\tau$  whenever the type is clear.

**Remark 6.1.8.** For all types  $\tau$ ,  $\lfloor - \rfloor$  is monotonic.

**Definition 6.1.9.** A *valuation*  $v$  is a function assigning to each variable  $x: \tau$  a value  $v(x) \in \llbracket \tau \rrbracket$ . Given a valuation  $v$ , let  $v\{x \leftarrow t\}$  denote the valuation on  $\Gamma \cup \{x: \tau\}$  which assigns  $t: \tau$  to  $x$  and otherwise behaves as  $v$ .

The interpretation is then defined as follows. For the interpretation to be well-defined, the construction for each term must be shown to preserve monotonicity. We will do so in a subsequent lemma (Lemma 6.1.13).

**Definition 6.1.10.** For each term  $\Gamma \vdash M: \vec{\sigma}_A \Rightarrow \vec{\tau}_A$  and valuation  $v$ , we inductively define the *interpretation* function

$$\llbracket \Gamma \vdash M: \tau \rrbracket_v \in \llbracket \tau \rrbracket.$$

We omit the context and/or types of terms inside an interpretation when it is clear.

$$\begin{aligned}
\llbracket \star : \tilde{\tau}_A \Rightarrow \vec{\tau}_A \rrbracket_v(t) &= (0, t) \\
\llbracket a(x).M : a(\rho) \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_v(s, a(r)) &= (1 + n, t) \\
&\text{where } (n, t) = \llbracket M : \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_{v\{x \leftarrow r\}}(s) \\
\llbracket [N]a.M : \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_v(s) &= (1 + n + \lfloor \llbracket N : \rho \rrbracket_v \rfloor, t) \\
&\text{where } (n, t) = \llbracket M : a(\rho) \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_v(s, a(\llbracket N : \rho \rrbracket_v)) \\
\llbracket x.M : \tilde{\rho}_A \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_v(s, r) &= (n + m, t) \\
&\text{where } (m, t) = \llbracket M : \tilde{v}_A \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_v(s, u) \\
&\text{and } (n, u) = v(x : \tilde{\rho}_A \Rightarrow \vec{v}_A)(r)
\end{aligned}$$

We define  $\llbracket \Gamma \vdash M : \tau \rrbracket$  to be  $\llbracket \Gamma \vdash M : \tau \rrbracket_w$ , for the *least valuation*  $w = \{x \leftarrow 0_\sigma \mid x : \sigma\}$  given by the minimal elements.

**Remark 6.1.11.** In this definition, the application case  $\llbracket [N]a.M \rrbracket_v(s) = (1 + m + \lfloor \llbracket N : \rho \rrbracket_v \rfloor, t)$  adds the measure  $\lfloor \llbracket N : \rho \rrbracket_v \rfloor$  to account for reduction inside the argument  $N$ . Further, both it and the abstraction case  $\llbracket a(x).M \rrbracket_v(s, a(r)) = (1 + m, t)$  add 1 to count redexes, so that a reduction step reduces the overall measure by (at least) 2. It would suffice to count only abstractions or only applications, but the choice to count both is so that we count steps of the stack machine. We observe the following: for the alternative interpretation that omits to count  $\lfloor \llbracket N : \rho \rrbracket_v \rfloor$ , and instead has  $\llbracket [N]a.M \rrbracket_v(s) = (1 + m, t)$ , then  $\llbracket M \rrbracket(\llbracket S_A \rrbracket)$  measures the exact length of the machine run on  $(S_A, M)$ , where  $\llbracket S_A \rrbracket$  is defined element-wise. This is formally proven later, in Section 6.6, and provides the proof with an operational intuition.

Note that in giving the above definition, technically, we must simultaneously prove its well-definedness at each inductive step. That is, each level relies on the previous level being well-defined. For example, in the application case, to define  $\llbracket [N].M : \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_v$ , we assume that  $\llbracket N \rrbracket_v \in \llbracket \rho \rrbracket$  and  $\llbracket M \rrbracket_v \in \llbracket a(\rho) \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket$ , (*i.e.* that they are monotonic) in order to construct  $\llbracket [N].M \rrbracket \in \llbracket \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket$ , and then prove that  $\llbracket [N].M \rrbracket$  is indeed monotonic.

We prove well-definedness in the lemma below. However, we need to strengthen the inductive hypothesis, as proving monotonicity in the abstraction case necessitates proving that “increasing” the valuation increases the interpretation. Thus, we add a second such clause to the statement of the lemma, which relies on the following definition.

**Definition 6.1.12.** Given two valuations  $v$  and  $w$ , write

$$v \leq w \quad \text{if} \quad \forall x : \tau \in \Gamma, \quad v(x) \leq_{\llbracket \tau \rrbracket} w(x) .$$

**Lemma 6.1.13.** For all terms  $\Gamma \vdash M : \tau$  and valuations  $v$  and  $w$ , we have that:

$$(i) \quad \llbracket M \rrbracket_v \in \llbracket \tau \rrbracket \quad \text{and} \quad (ii) \quad v \leq w \quad \text{implies} \quad \llbracket M \rrbracket_v \leq_{\llbracket \tau \rrbracket} \llbracket M \rrbracket_w .$$

*Proof.* We prove both statements simultaneously by induction on the type derivation of  $\Gamma \vdash M : \tau$ . Recall that the first item is equivalent to claiming  $\llbracket M : \tau \rrbracket_v$  is monotonic. When we write ‘increasing’ here, we mean non-strictly.

- For the base case

$$\Gamma \vdash M \equiv \star : \overleftarrow{\tau}_A \Rightarrow \overrightarrow{\tau}_A,$$

- (1) Observe that  $\llbracket \star \rrbracket_v(t) = (0, t)$  and so is monotonic.
- (2) Observe that  $\llbracket \star \rrbracket_v = \llbracket \star \rrbracket_w$  for every  $v$  and  $w$ .

- For the abstraction case, where  $x : \rho$  and

$$\Gamma \vdash M \equiv a\langle x \rangle. M' : a(\rho) \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A,$$

- (1) We must show the function

$$\begin{aligned} \llbracket a\langle x \rangle. M \rrbracket_v(s, a(r)) &= (1 + n, t) \\ \text{where } (n, t) &= \llbracket M \rrbracket_{v\{x \leftarrow r\}}(s) \end{aligned}$$

is monotonic. Indeed, we have that  $(s, a(r)) \leq \llbracket \overrightarrow{\sigma}_A a(\rho) \rrbracket (s', a(r'))$  implies, by inductive hypothesis (ii) on  $M'$ , that

$$\llbracket M' \rrbracket_{v\{x \leftarrow r\}} \leq \llbracket \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A \rrbracket \llbracket M' \rrbracket_{v\{x \leftarrow r'\}} ,$$

and then by inductive hypothesis (i) on  $M'$ , this implies

$$\llbracket M' \rrbracket_{v\{x \leftarrow r\}}(s) \leq \llbracket \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A \rrbracket \llbracket M' \rrbracket_{v\{x \leftarrow r'\}}(s') .$$

Thus, increasing the input of the function increases its output.

- (2) We wish to show, for arbitrary  $(s, a(r)) \in \llbracket \overrightarrow{\sigma}_A a(\rho) \rrbracket$ , that

$$v \leq w \quad \text{implies} \quad \llbracket a\langle x \rangle. M' \rrbracket_v(s, a(r)) \leq_{\mathbb{N} \times \llbracket \overrightarrow{\tau}_A \rrbracket} \llbracket a\langle x \rangle. M' \rrbracket_w(s, a(r)) .$$

Unfolding definitions, we see we must show that  $v \leq w$  implies

$$\begin{aligned} (1 + n, t) &\leq_{\mathbb{N} \times \llbracket \overrightarrow{\tau}_A \rrbracket} (1 + n', t') \\ \text{where } (n, t) &= \llbracket M \rrbracket_{v\{x \leftarrow r\}}(s) , \\ \text{and } (n', t') &= \llbracket M \rrbracket_{w\{x \leftarrow r\}}(s) \end{aligned}$$

By assumption, we have that, for all  $r \in \llbracket \rho \rrbracket$ ,  $v\{x \leftarrow r\} \leq w\{x \leftarrow r\}$ . Applying inductive hypothesis (ii) on  $M'$ , we thus have that

$$\llbracket M' \rrbracket_{v\{x \leftarrow r\}} \leq \llbracket \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A \rrbracket \llbracket M' \rrbracket_{w\{x \leftarrow r\}} .$$

and consequently  $(n, t) \leq_{\mathbb{N} \times \llbracket \overrightarrow{\tau}_A \rrbracket} (n', t')$ . Indeed, the required result immediately follows.

- For the application case, with  $\Gamma \vdash N : \rho$  and

$$\Gamma \vdash M \equiv [N]a.M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

- (1) We have to show that

$$\begin{aligned} \llbracket [N]a.M' \rrbracket_v(s) &= (1 + n + \lfloor \llbracket N \rrbracket_v \rfloor, t) \\ \text{where } (n, t) &= \llbracket M' \rrbracket_v(s, a(\llbracket N \rrbracket_v)) . \end{aligned}$$

is monotonic. Applying inductive hypothesis (i) on  $M'$ , we achieve that  $\llbracket M' \rrbracket_v$  is monotonic. Thus, increasing the input  $s$  increases  $(n, t)$ , which therefore increases the output of the entire function.

- (2) We have to show, for arbitrary  $s \in \llbracket \vec{\sigma}_A \rrbracket$ , that

$$v \leq w \quad \text{implies} \quad \llbracket [N]a.M' \rrbracket_v(s) \leq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket [N]a.M' \rrbracket_w(s)$$

Unfolding definitions, we see we must show that  $v \leq w$  implies

$$\begin{aligned} (1 + n + \lfloor \llbracket N \rrbracket_v \rfloor, t) &\leq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (1 + n' + \lfloor \llbracket N \rrbracket_w \rfloor, t') \\ \text{where } (n, t) &= \llbracket M' \rrbracket_w(s, a(\llbracket N \rrbracket_v)) . \\ \text{and } (n', t') &= \llbracket M' \rrbracket_w(s, a(\llbracket N \rrbracket_w)) . \end{aligned}$$

Applying inductive hypothesis (ii) on  $M'$  and  $N$ , we achieve

$$\llbracket M' \rrbracket_v \leq_{\llbracket a(\rho) \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket} \llbracket M' \rrbracket_w \quad \text{and} \quad \llbracket N \rrbracket_v \leq_{\llbracket \rho \rrbracket} \llbracket N \rrbracket_w.$$

The conjunction of both statements implies that  $(n, t) \leq_{\mathbb{N} \times \llbracket \vec{\tau} \rrbracket} (n', t')$ . Using Remark 6.1.8, we additionally observe  $\lfloor \llbracket N \rrbracket_v \rfloor \leq_{\mathbb{N}} \lfloor \llbracket N \rrbracket_w \rfloor$ , and the result follows.

- For the variable case,

$$\Gamma, x : \vec{\rho}_A \Rightarrow \vec{v}_A \vdash M \equiv x.M' : \vec{\rho}_A \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

- (1) We have to show that

$$\begin{aligned} \llbracket x.M' \rrbracket_v(s, r) &= (n + m, t) \\ \text{where } (m, t) &= \llbracket M' \rrbracket_v(s, u) \\ \text{and } (n, u) &= v(x)(r) . \end{aligned}$$

is monotonic. Observe that  $v(x) \in \llbracket \vec{\rho}_A \Rightarrow \vec{v}_A \rrbracket$  and so is monotonic. Thus, increasing the input  $(s, r)$  increases  $(n, u)$ . We have from inductive hypothesis (i) on  $M'$  that  $\llbracket M' \rrbracket_v$  is monotonic. Altogether, this results in an increase in  $(m, t)$ , which therefore increases the output of the entire function.

(2) We wish to show, for arbitrary  $(s, r) \in \vec{\sigma}_A \vec{\rho}_A$ , that

$$v \leq w \quad \text{implies} \quad \llbracket x.M' \rrbracket_v(s, r) \leq_{\mathbb{N} \times \llbracket \tau_A \rrbracket} \llbracket x.M' \rrbracket_w(s, r) .$$

Unfolding definitions, we see we must show that  $v \leq w$  implies

$$\begin{aligned} (n + m, t) &\leq_{\mathbb{N} \times \llbracket \tau_A \rrbracket} (n' + m', t') \\ \text{where } (m, t) &= \llbracket M' \rrbracket_v(s, u) \\ \text{and } (n, u) &= v(x)(r) \\ \text{and } (m', t') &= \llbracket M' \rrbracket_w(s, u') \\ \text{and } (n', u') &= w(x)(r) . \end{aligned}$$

By assumption, we have  $v(x) \leq_{\llbracket \vec{\rho}_A \Rightarrow \vec{v}_A \rrbracket} w(x)$ , which implies that  $(n, u) \leq_{\mathbb{N} \times \llbracket \vec{v}_A \rrbracket} (n', u')$ . Applying inductive hypothesis (ii) on  $M'$ , we have that

$$\llbracket M' \rrbracket_v \leq_{\llbracket \vec{v}_A \vec{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket} \llbracket M' \rrbracket_w .$$

Altogether, this implies  $(m, t) \leq_{\mathbb{N} \times \llbracket \tau_A \rrbracket} (m', t')$ . Thus, we achieve the required result.  $\square$

**Remark 6.1.14.** Note that monotonicity does not rely on the measure of the argument,  $\llbracket N \rrbracket_v$ , being added to the count in the application case.

We will see later that monotonicity alone suffices to prove that reduction inside a term non-strictly decreases the measure (Lemma 6.3.1). Adding  $\llbracket N \rrbracket_v$  to the count is, however, necessary for achieving a strict inequality.

## 6.2 The Substitution Lemma

To prove the base case of the main proposition (Proposition 6.4.1, that beta reduction reduces the measure), we need to prove that substitution commutes with the interpretation in an appropriate way. In order to prove that, we must first show that weakening and sequencing behave well with respect to the interpretation. Each of these lemmata is proved by a simple induction on the type derivation of terms.

**Lemma 6.2.1** (Weakening). *For terms  $\Gamma \vdash M : \tau$  and valuation  $v$  and  $s \in \llbracket \sigma \rrbracket$ , we have that*

$$\llbracket \Gamma \vdash M : \tau \rrbracket_v = \llbracket \Gamma, x : \sigma \vdash M : \tau \rrbracket_{v\{x \leftarrow s\}},$$

where  $x \notin \text{fv}(M)$ .

*Proof.* Induction on the type derivation of  $\Gamma \vdash M : \tau$ .  $\square$

As a consequence of this lemma, we may speak of  $\llbracket M : \tau \rrbracket$  independent of a valuation  $v$  when  $M$  is closed.

We must show that sequencing behaves well with respect to the interpretation because substitution for a variable results in a sequencing operation, and this is required in the variable case of the Substitution Lemma.

**Lemma 6.2.2** (Sequencing). *For terms  $\Gamma \vdash M : \tilde{\sigma}_A \tilde{\tau}_A \Rightarrow \vec{v}_A$  and  $\Gamma \vdash N : \tilde{\rho}_A \Rightarrow \vec{\sigma}_A$  and valuation  $v$ , we have:*

$$\begin{aligned} \llbracket N ; M \rrbracket_v(t, r) &= (i + j, u) \\ \text{where } (i, s) &= \llbracket N \rrbracket_v(r) \\ \text{and } (j, u) &= \llbracket M \rrbracket_v(t, s). \end{aligned}$$

*Proof.* We proceed by induction on the type derivation of  $\Gamma \vdash N : \tau$ .

- For the base case,

$$\Gamma \vdash N \equiv \star : \tilde{\sigma}_A \Rightarrow \vec{\sigma}_A,$$

let  $\llbracket M \rrbracket_v(t, s) = (m, u)$ . Since  $\llbracket \star \rrbracket_v(s) = (0, s)$ , we need to show that  $\llbracket \star ; M \rrbracket_v(t, s) = (0 + m, u)$ , but this is immediate since  $\star ; M = M$ .

- For the abstraction case,

$$\Gamma \vdash N \equiv a\langle x \rangle . N' : \alpha(\pi) \tilde{\rho}_A \Rightarrow \vec{\sigma}_A,$$

let

$$\llbracket N' \rrbracket_{v\{x \leftarrow p\}}(r) = (i, s), \quad \llbracket M \rrbracket_v(t, s) = (j, u),$$

so that  $\llbracket a\langle x \rangle . N' \rrbracket_v(r, p) = (1 + i, s)$ . Because  $x$  is not free in  $M$ , by Lemma 6.2.1 we have  $\llbracket M \rrbracket_{v\{x \leftarrow p\}}(t, s) = (j, u)$ . The inductive hypothesis gives  $\llbracket N' ; M \rrbracket_{v\{x \leftarrow p\}}(t, r) = (i + j, u)$ , so that  $\llbracket a\langle x \rangle . (N' ; M) \rrbracket_v(t, r, p) = (1 + i + j, u)$ . By definition,  $a\langle x \rangle . N' ; M = a\langle x \rangle . (N' ; M)$ , so this is the required result.

- For the application case, with  $P : \pi$ ,

$$\Gamma \vdash N \equiv [P]a . N' : \tilde{\rho}_A \Rightarrow \vec{\sigma}_A,$$

let

$$\llbracket N' \rrbracket_v(r, \llbracket P \rrbracket_v) = (i, s), \quad \llbracket M \rrbracket_v(t, s) = (j, u),$$

so that  $\llbracket [P]a . N' \rrbracket_v(r) = (\lfloor \llbracket P \rrbracket_v \rfloor + 1 + i, s)$ . The inductive hypothesis gives  $\llbracket N' ; M \rrbracket_v(t, r, \llbracket P \rrbracket_v) = (i + j, u)$ , so that  $\llbracket [P]a . (N' ; M) \rrbracket_v(t, r) = (\lfloor \llbracket P \rrbracket_v \rfloor + 1 + i + j, u)$ . By definition,  $[P]a . N' ; M = [P]a . (N' ; M)$ , so this is the required result.

- For the variable case,

$$\Gamma, x : \tilde{\nu}_A \Rightarrow \vec{\pi}_A \vdash N \equiv x . N' : \tilde{\nu}_A \tilde{\rho}_A \Rightarrow \vec{\sigma}_A,$$



let

$$v(x)(n) = (i, p) \quad \llbracket N' \rrbracket_v(r, p) = (j, s) \quad \llbracket M \rrbracket_v(t, s) = (k, u)$$

so that  $\llbracket x.N' \rrbracket_v(r, n) = (i + j, s)$ . The inductive hypothesis gives  $\llbracket N'; M \rrbracket_v(t, r, p) = (j + k, u)$ , so that  $\llbracket x.(N'; M) \rrbracket_v(t, r, n) = (i + j + k, u)$ . By definition,  $x.N'; M = x.(N'; M)$ , so this is the required result.  $\square$

We finally come to prove the Substitution Lemma. This is required in the base case of the main Proposition 6.4.1 to follow.

**Lemma 6.2.3** (Substitution). *For terms  $\Gamma, x: \omega \vdash M: \tau$  and  $\Gamma \vdash N: \omega$  and valuation  $v$ , we have*

$$\llbracket \{N/x\}M \rrbracket_v = \llbracket M \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}.$$

*Proof.* We proceed by induction on the type derivation of  $\Gamma, x: \omega \vdash M: \tau$ .

- For the base case,

$$\Gamma, x: \omega \vdash M \equiv \star: \tilde{\tau}_A \Rightarrow \vec{\tau}_A,$$

we have

$$\llbracket \Gamma \vdash \{N/x\}\star \rrbracket_v(t) = \llbracket \Gamma \vdash \star \rrbracket_v(t) = (0, t) = \llbracket \Gamma, x \vdash \star \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(t),$$

as required.

- For the abstraction case,

$$\Gamma, x: \omega \vdash M \equiv a\langle y \rangle. M': a(\rho)\tilde{\sigma}_A \Rightarrow \vec{\tau}_A,$$

we need to show that

$$\llbracket \{N/x\}a\langle y \rangle. M' \rrbracket_v(s, a(r)) = \llbracket a\langle y \rangle. M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, a(r)),$$

where  $x \neq y$ . For the left-hand side, we have

$$\begin{aligned} \llbracket \{N/x\}a\langle y \rangle. M' \rrbracket_v(s, a(r)) &= \\ \llbracket a\langle y \rangle. \{N/x\}M' \rrbracket_v(s, a(r)) &= (1 + n, t) \\ \text{where } (n, t) &= \llbracket \{N/x\}M' \rrbracket_{v\{y \leftarrow r\}}(s), \end{aligned}$$

and, for the right-hand side, we have

$$\begin{aligned} \llbracket a\langle y \rangle. M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, a(r)) &= (1 + n', t') \\ \text{where } (n', t') &= \llbracket M' \rrbracket_{u\{x \leftarrow \llbracket N \rrbracket_v\}}(s). \end{aligned}$$

Let  $u = v\{y \leftarrow r\}$ . Applying the inductive hypothesis on  $M'$  gives the first equality below.

$$\llbracket \{N/x\}M \rrbracket_{v\{y \leftarrow r\}} = \llbracket M' \rrbracket_{u\{x \leftarrow \llbracket N \rrbracket_u\}} = \llbracket M' \rrbracket_{u\{x \leftarrow \llbracket N \rrbracket_v\}}$$

By the Weakening Lemma 6.2.1, and since  $y \notin \text{fv}(N)$ , we have  $\llbracket N \rrbracket_u = \llbracket N \rrbracket_v$ . This gives the second equality. Thus, we have that  $(n, t) = (n', t')$  and the required result follows.

- For the application case, where  $\Gamma, x : \omega \vdash P : \rho$  and

$$\Gamma, x : \omega \vdash M \equiv [P]a. M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

we need to show that

$$\llbracket \{N/x\}[P]a. M' \rrbracket_v(s) = \llbracket [P]a. M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s) .$$

For the left-hand side, we have

$$\begin{aligned} \llbracket \{N/x\}[P]a. M' \rrbracket_v(s) &= \\ \llbracket [\{N/x\}P]a. \{N/x\}M' \rrbracket_v(s) &= (1 + n + \lfloor \llbracket \{N/x\}P \rrbracket_v \rfloor, t) \\ \text{where } (n, t) &= \llbracket \{N/x\}M' \rrbracket_v(s, a(\llbracket \{N/x\}P \rrbracket_v)) , \end{aligned}$$

and, for the right-hand side, we have

$$\begin{aligned} \llbracket [P]a. M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s) &= (1 + n' + \lfloor \llbracket P \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}} \rfloor, t') \\ \text{where } (n', t') &= \llbracket M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, a(\llbracket P \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}})) . \end{aligned}$$

Applying the inductive hypothesis on  $M'$  and  $P$  achieves

$$\begin{aligned} \llbracket \{N/x\}M' \rrbracket_v &= \llbracket M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}} \\ \llbracket \{N/x\}P \rrbracket_v &= \llbracket P \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}} . \end{aligned}$$

Thus, we have that  $(n, t) = (n', t')$  and indeed  $1 + n' + \lfloor \llbracket \{N/x\}P \rrbracket_v \rfloor = 1 + n' + \lfloor \llbracket P \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}} \rfloor$  as required.

- For the variable case, where  $y \neq x$ ,

$$\Gamma, x : \omega, y : \vec{\rho}_A \Rightarrow \vec{v}_A \vdash M \equiv y. M' : \vec{\rho}_A \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

we need to show that

$$\llbracket \{N/x\}y. M' \rrbracket_v(s, r) = \llbracket y. M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, r) .$$

For the left-hand side,

$$\begin{aligned} \llbracket \{N/x\}y. M' \rrbracket_v(s, r) &= \\ \llbracket y. \{N/x\}M' \rrbracket_v(s, r) &= (n + m, t) \\ \text{where } (m, t) &= \llbracket \{N/x\}M' \rrbracket_v(s, u) \\ \text{and } (n, u) &= v(y)(r) , \end{aligned}$$

and for the right-hand side,

$$\begin{aligned} \llbracket y. M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, r) &= (n' + m', t') \\ \text{where } (m', t') &= \llbracket M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, u') \\ \text{and } (n', u') &= v\{x \leftarrow \llbracket N \rrbracket_v\}(y)(r) . \end{aligned}$$

Observe that  $v\{x \leftarrow \llbracket N \rrbracket_v\}(y) = v(y)$ , which implies  $(n, u) = (n', u')$ . Application of the inductive hypothesis on  $M'$  achieves

$$\llbracket \{N/x\}M' \rrbracket_v = \llbracket M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}} .$$

Thus, we have  $(m, t) = (m', t')$  and the result follows.

- For the variable case,

$$\Gamma, x : \bar{\rho}_A \Rightarrow \bar{v}_A \vdash M \equiv x.M' : \bar{\rho}_A \bar{\sigma}_A \Rightarrow \bar{\tau}_A,$$

we need to show that

$$\llbracket \{N/x\}x.M \rrbracket_v(s, r) = \llbracket x.M \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, r) .$$

For the left-hand side, by application of the Sequencing Lemma 6.2.2, we have

$$\begin{aligned} \llbracket \{N/x\}x.M' \rrbracket_v(s, r) &= \\ \llbracket N ; \{N/x\}M' \rrbracket_v(s, r) &= (n + m, t) \\ \text{where } (m, t) &= \llbracket \{N/x\}M' \rrbracket_v(s, u) \\ \text{and } (n, u) &= \llbracket N \rrbracket_v(r) , \end{aligned}$$

and, for the right-hand side,

$$\begin{aligned} \llbracket x.M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, r) &= (n' + m', t') \\ \text{where } (m', t') &= \llbracket M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s, u') \\ \text{and } (n', u') &= v\{x \leftarrow \llbracket N \rrbracket_v\}(x)(r) . \end{aligned}$$

Observe that  $v\{x \leftarrow \llbracket N \rrbracket_v\}(x) = \llbracket N \rrbracket_v$ , which implies  $(n, u) = (n', u')$ . Application of the inductive hypothesis on  $M'$  achieves

$$\llbracket \{N/x\}M' \rrbracket_v = \llbracket M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}} .$$

Thus, we have  $(m, t) = (m', t')$  and the result follows.  $\square$

### 6.3 Beta Reduction is Monotonic

We now prove that reduction inside a term non-strictly decreases the interpretation of that term, which is necessary for the argument case of the main lemma, which claims that reduction *strictly* decreases the measure.

**Lemma 6.3.1.** *For every valuation  $v$ ,*

$$\Gamma \vdash M \rightarrow M' : \tau \text{ implies } \llbracket M \rrbracket_v \geq_{[\tau]} \llbracket M' \rrbracket_v .$$

*Proof.* We proceed by induction on the derivation of one-step reductions.

- The base case,

$$[N]a.a\langle x \rangle.M \rightarrow \{N/x\}M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

where  $x, N : \rho$  requires showing, for arbitrary  $s \in \llbracket \vec{\sigma}_A \rrbracket$ , that

$$\llbracket [N]a.a\langle x \rangle.M \rrbracket_v(s) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket \{N/x\}M \rrbracket_v(s) .$$

Unfolding definitions, we must show

$$(2 + n + \lfloor \llbracket N \rrbracket_v \rfloor, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket \{N/x\}M \rrbracket_v(s) \\ \text{where } (n, t) = \llbracket M \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s) .$$

Applying the Substitution Lemma 6.2.3,

$$\llbracket M \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}} = \llbracket \{N/x\}M \rrbracket_v,$$

we see  $(n, t) = \llbracket \{N/x\}M \rrbracket_v(s)$ . Observing that  $(2 + n + \lfloor \llbracket N \rrbracket_v \rfloor, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (n, t)$ , the result follows.

- The abstraction case,

$$a\langle x \rangle.M \rightarrow a\langle x \rangle.M' : a(\rho)\vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

requires showing, for arbitrary  $(s, a(r)) \in \llbracket \vec{\sigma}_A a(\rho) \rrbracket$ , that

$$\llbracket a\langle x \rangle.M \rrbracket_v(s, a(r)) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket a\langle x \rangle.M' \rrbracket_v(s, a(r)) .$$

Unfolding definitions, we must show

$$(1 + n, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (1 + n', t') \\ \text{where } (n, t) = \llbracket M \rrbracket_{v\{x \leftarrow r\}}(s) \\ \text{and } (n', t') = \llbracket M' \rrbracket_{v\{x \leftarrow r\}}(s) .$$

Applying the inductive hypothesis on  $M \rightarrow M'$ , we achieve

$$\llbracket M \rrbracket_w(s) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket M' \rrbracket_w(s) ,$$

for any valuation  $w$ . In particular, we can set  $w = \{x \leftarrow r\}$  and thus we have that  $(n, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (n', t')$ . The result follows.

- The application, function case,

$$[N]a.M \rightarrow [N]a.M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

requires showing, for arbitrary  $s \in \llbracket \vec{\sigma}_A \rrbracket$ , that

$$\llbracket [N]a.M \rrbracket_v(s) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket [N]a.M' \rrbracket_v(s) .$$

Unfolding definitions, we must show

$$(1 + n + \lfloor \llbracket N \rrbracket_v \rfloor, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (1 + n' + \lfloor \llbracket N \rrbracket_v \rfloor, t')$$

where  $(n, t) = \llbracket M \rrbracket_v(s, a(\llbracket N \rrbracket_v))$   
and  $(n', t') = \llbracket M' \rrbracket_v(s, a(\llbracket N \rrbracket_v))$  .

Applying the inductive hypothesis on  $M \rightarrow M'$ , we achieve

$$\llbracket M \rrbracket_w(s, a(\llbracket N \rrbracket_v)) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket M' \rrbracket_w(s, a(\llbracket N \rrbracket_v)) ,$$

and thus that  $(n, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau} \rrbracket} (n', t')$ . The result follows.

- The application, argument case, where  $N : \rho$ ,

$$\llbracket N \rrbracket. M \rightarrow \llbracket N' \rrbracket. M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A,$$

requires showing, for arbitrary  $s \in \llbracket \vec{\sigma}_A \rrbracket$ , that

$$\llbracket \llbracket N \rrbracket a. M \rrbracket_v(s) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket \llbracket N' \rrbracket a. M \rrbracket_v(s) .$$

Unfolding definitions, we must show

$$(1 + n + \lfloor \llbracket N \rrbracket_v \rfloor, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (1 + n' + \lfloor \llbracket N' \rrbracket_v \rfloor, t')$$

where  $(n, t) = \llbracket M \rrbracket_v(s, a(\llbracket N \rrbracket_v))$   
and  $(n', t') = \llbracket M \rrbracket_v(s, a(\llbracket N' \rrbracket_v))$

Applying the inductive hypothesis on  $N \rightarrow N'$ , we achieve

$$\llbracket N \rrbracket_v \geq_{\llbracket \rho \rrbracket} \llbracket N' \rrbracket_v$$

which allows us to apply monotonicity of  $\llbracket M \rrbracket_v$  to see that  $(n, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (n', t')$ . Remark 6.1.8, stating monotonicity of  $\lfloor - \rfloor$ , then implies that  $(1 + n + \lfloor \llbracket N \rrbracket_v \rfloor, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (1 + n' + \lfloor \llbracket N' \rrbracket_v \rfloor, t')$ , as required.

- The variable case,

$$x. M \rightarrow x. M' : \vec{\rho}_A \Rightarrow \vec{\tau}_A,$$

with  $x : \vec{\rho}_A \Rightarrow \vec{\tau}_A$  requires showing, for arbitrary  $(s, r) \in \llbracket \vec{\sigma}_A \vec{\rho}_A \rrbracket$ , that

$$\llbracket x. M \rrbracket_v(s, r) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket x. M' \rrbracket_v(s, r) .$$

Unfolding definitions, we must show

$$(n + m, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (n + m', t')$$

where  $(m, t) = \llbracket M \rrbracket_v(s, u)$   
and  $(m', t') = \llbracket M' \rrbracket_v(s, u)$   
and  $(n, u) = v(x)(r)$

Applying the inductive hypothesis on  $M \rightarrow M'$ , we achieve, for any  $u \in \llbracket \vec{v}_A \rrbracket$ ,

$$\llbracket M \rrbracket_w(s, u) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} \llbracket M' \rrbracket_w(s, u) ,$$

and thus that  $(m, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (m', t')$  and consequently  $(n + m, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau}_A \rrbracket} (n + m', t')$ , as required.  $\square$

## 6.4 The Measure for Strong Normalisation

The next lemma immediately implies the strong normalisation result: if  $\Gamma \vdash M \rightarrow N : \tau$  then  $\llbracket M \rrbracket >_{\mathbb{N}} \llbracket N \rrbracket$ , so that  $\llbracket M \rrbracket$  gives a bound for the length of any reduction path from  $M$ . The following proof relies essentially on monotonicity of the interpretation of a term, in particular for the application, argument case, as well as the fact the measure of the argument is added to the count as it is pushed to the stack

**Proposition 6.4.1.** *For every valuation  $v$  and for all  $s \in \llbracket \vec{\sigma}_A \rrbracket$ , we have that*

$$\Gamma \vdash M \rightarrow M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A \text{ implies } \pi_1(\llbracket M \rrbracket_v(s)) >_{\mathbb{N}} \pi_1(\llbracket M' \rrbracket_v(s)) .$$

*Proof.* We proceed by induction on the derivation of one-step reductions.

- The base case,

$$[N]a.a\langle x \rangle.M \rightarrow \{N/x\}M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A ,$$

where  $x, N : \rho$  requires showing, for arbitrary  $s \in \llbracket \vec{\sigma}_A \rrbracket$ , that

$$\pi_1(\llbracket [N]a.a\langle x \rangle.M \rrbracket_v(s)) >_{\mathbb{N}} \pi_1(\llbracket \{N/x\}M \rrbracket_v(s)) .$$

Unfolding definitions, we must show

$$2 + n + \llbracket N \rrbracket_v >_{\mathbb{N}} \pi_1(\llbracket \{N/x\}M \rrbracket_v(s))$$

where  $(n, t) = \llbracket M \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}}(s)$  .

Applying the Substitution Lemma 6.2.3,

$$\llbracket M \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v\}} = \llbracket \{N/x\}M \rrbracket_v ,$$

we see  $(n, t) = \llbracket \{N/x\}M \rrbracket_v(s)$ . Indeed, the required result follows immediately from observing that  $n = \pi_1(\llbracket \{N/x\}M \rrbracket_v(s))$ .

- The abstraction case,

$$a\langle x \rangle.M \rightarrow a\langle x \rangle.M' : a(\rho)\vec{\sigma}_A \Rightarrow \vec{\tau}_A ,$$

requires showing, for arbitrary  $(s, a(r)) \in \llbracket \vec{\sigma}_A a(\rho) \rrbracket$ , that

$$\pi_1(\llbracket a\langle x \rangle.M \rrbracket_v(s, a(r))) >_{\mathbb{N}} \pi_1(\llbracket a\langle x \rangle.M' \rrbracket_v(s, a(r))) .$$

Unfolding definitions, we must show

$$\begin{aligned} 1 + n &>_{\mathbb{N}} 1 + n' \\ \text{where } (n, t) &= \llbracket M \rrbracket_{v\{x \leftarrow r\}}(s) \\ \text{and } (n', t') &= \llbracket M' \rrbracket_{v\{x \leftarrow r\}}(s) . \end{aligned}$$

Applying the inductive hypothesis on  $M \rightarrow M'$ , we achieve

$$\pi_1(\llbracket M \rrbracket_w(s)) >_{\mathbb{N}} \pi_1(\llbracket M' \rrbracket_w(s)) ,$$

for any valuation  $w$ . In particular, we can set  $w = \{x \leftarrow r\}$  and thus we have that  $n >_{\mathbb{N}} n'$ , as required.

- The application, function case,

$$[N]a.M \rightarrow [N]a.M' : \vec{\sigma}_A \Rightarrow \vec{\tau}_A ,$$

requires showing, for arbitrary  $s \in \llbracket \vec{\sigma}_A \rrbracket$ , that

$$\pi_1(\llbracket [N]a.M \rrbracket_v(s)) >_{\mathbb{N}} \pi_1(\llbracket [N]a.M' \rrbracket_v(s)) .$$

Unfolding definitions, we must show

$$\begin{aligned} 1 + n + \lfloor \llbracket N \rrbracket_v \rfloor &>_{\mathbb{N}} 1 + n' + \lfloor \llbracket N \rrbracket_v \rfloor \\ \text{where } (n, t) &= \llbracket M \rrbracket_v(s, a(\llbracket N \rrbracket_v)) \\ \text{and } (n', t') &= \llbracket M' \rrbracket_v(s, a(\llbracket N \rrbracket_v)) . \end{aligned}$$

Applying the inductive hypothesis on  $M \rightarrow M'$ , we achieve

$$\pi_1(\llbracket M \rrbracket_w(s, a(\llbracket N \rrbracket_v))) >_{\mathbb{N}} \pi_1(\llbracket M' \rrbracket_w(s, a(\llbracket N \rrbracket_v))) ,$$

and thus that  $n >_{\mathbb{N}} n'$ , as required.

- The application, argument case, where  $N : \rho$ ,

$$[N]a.M \rightarrow [N']a.M : \vec{\sigma}_A \Rightarrow \vec{\tau}_A ,$$

requires showing, for arbitrary  $s \in \llbracket \vec{\sigma}_A \rrbracket$ , that

$$\pi_1(\llbracket [N]a.M \rrbracket_v(s)) >_{\mathbb{N}} \pi_1(\llbracket [N']a.M \rrbracket_v(s)) .$$

Unfolding definitions, we must show

$$\begin{aligned} 1 + n + \lfloor \llbracket N \rrbracket_v \rfloor &>_{\mathbb{N}} 1 + n' + \lfloor \llbracket N' \rrbracket_v \rfloor \\ \text{where } (n, t) &= \llbracket M \rrbracket_v(s, a(\llbracket N \rrbracket_v)) \\ \text{and } (n', t') &= \llbracket M \rrbracket_v(s, a(\llbracket N' \rrbracket_v)) \end{aligned}$$

Applying the inductive hypothesis on  $N \rightarrow N'$ , we achieve

$$\pi_1(\llbracket N \rrbracket_v(s)) >_{\mathbb{N}} \pi_1(\llbracket N' \rrbracket_v(s))$$

In the special case where  $s$  is the minimal element, we recover  $\llbracket N \rrbracket_v >_{\mathbb{N}} \llbracket N' \rrbracket_v$ . Additionally, by Lemma 6.3.1, we have that  $\llbracket N \rrbracket \geq_{\mathbb{N} \times \llbracket \vec{\tau} \rrbracket} \llbracket N' \rrbracket$ . This allows us to apply monotonicity of  $\llbracket M \rrbracket$  to deduce that  $(n, t) \geq_{\mathbb{N} \times \llbracket \vec{\tau} \rrbracket} (n', t')$ . Altogether, this suffices for the result.

- The variable case,

$$x.M \rightarrow x.M' : \tilde{\rho}_A \tilde{\sigma}_A \Rightarrow \vec{\tau}_A,$$

with  $x : \tilde{\rho}_A \Rightarrow \vec{v}_A$  requires showing, for arbitrary  $(s, r) \in \llbracket \vec{\sigma}_A \vec{\rho}_A \rrbracket$ , that

$$\pi_1(\llbracket x.M \rrbracket_v(s, r)) >_{\mathbb{N}} \pi_1(\llbracket x.M' \rrbracket_v(s, r)) .$$

Unfolding definitions, we must show

$$\begin{aligned} n + m &>_{\mathbb{N}} n + m' \\ \text{where } (m, t) &= \llbracket M \rrbracket_v(s, u) \\ \text{and } (m, 't') &= \llbracket M' \rrbracket_v(s, u) \\ \text{and } (n, u) &= v(x)(r) \end{aligned}$$

Applying the inductive hypothesis on  $M \rightarrow M'$ , we achieve, for any  $u \in \llbracket \vec{v}_A \rrbracket$ ,

$$\pi_1(\llbracket M \rrbracket_w(s, u)) >_{\mathbb{N}} \pi_1(\llbracket M' \rrbracket_w(s, u)) ,$$

and thus that  $m >_{\mathbb{N}} m'$ , as required.  $\square$

**Theorem 6.4.2.** *All typed FMC-terms are strongly normalizing with respect to beta reduction.*

*Proof.* For a term  $\Gamma \vdash M : \tau$ , Proposition 6.4.1 gives that any reduction path from  $M$  gives a strictly decreasing sequence in  $\mathbb{N}$ , by application of  $\llbracket - \rrbracket$ , which must therefore have finite length.  $\square$

## 6.5 Permutation Reduction

We show that the measure is invariant under permutation reduction, giving rise to a simple argument proving strong normalisation for the combined reduction relation given by both beta and permutation reduction.

**Lemma 6.5.1** (Permutation). *For terms  $\Gamma \vdash M : \tau$  and  $\Gamma \vdash N : \tau$  and valuations  $v$ , we have*

$$M \rightarrow_{\pi} N \text{ implies } \llbracket M \rrbracket_v = \llbracket N \rrbracket_v .$$



*Proof.* We recall the permutation equivalence:

$$[N]b. a\langle x \rangle. P \rightarrow_{\pi} a\langle x \rangle. [N]b. P : a(\rho)\bar{\sigma}_A \Rightarrow \bar{\tau}_A ,$$

where  $x \notin \text{fv}(N)$ . To see this, observe the following are equivalent:

$$\begin{aligned} \llbracket [N]b. a\langle x \rangle. P \rrbracket_v(s_A, a(r)) &= (n + 2 + \lfloor \llbracket N \rrbracket_v \rfloor, t), \\ \text{where } (n, t) &= \llbracket P \rrbracket_{v\{x \leftarrow r\}}(s, b(\llbracket N \rrbracket_v)), \\ \llbracket a\langle x \rangle. [N]b. P \rrbracket_v(s, a(r)) &= (n + 2 + \lfloor \llbracket N \rrbracket_{v\{x \leftarrow r\}} \rfloor, t), \\ \text{where } (n, t) &= \llbracket P \rrbracket_{v\{x \leftarrow r\}}(s, b(\llbracket N \rrbracket_{v\{x \leftarrow r\}})), \end{aligned}$$

using the Weakening Lemma 6.2.1 ( $x \notin \text{fv}(N)$ ).  $\square$

**Corollary 6.5.2.** *All typed FMC-terms are strongly normalising with respect to beta reduction and permutation reduction.*

*Proof.* It is easy enough to show that permutation reduction alone is strongly normalising.

$$[N]a. b\langle x \rangle. M \rightarrow_{\pi} b\langle x \rangle. [N]a. M$$

Consider the multi-set measure given by, for each application  $[N]a. M$  occurring in the term, the number of abstractions *not* on location  $a$  that are in  $M$ . Then permutation reduction strictly decreases this measure. Since permutation reduction makes no change to the measure for strong normalisation of beta reduction (Lemma 6.5.1), we can interleave beta reduction with any number of permutation reduction steps (which will necessarily be a finite number) without affecting strong normalisation.  $\square$

## 6.6 The Weak Interpretation

In order to clarify the operational intuition of the proof above, we inductively define the *weak interpretation*  $\llbracket - \rrbracket^W$ , which associates to each term a function taking the interpretation of its inputs to the *exact* length of its machine run on those inputs and the interpretation of its outputs. It is defined as the interpretation in the previous section, except without adding the measure of the argument to the count in the application case. Following the definition, we prove that  $\llbracket M \rrbracket^W(\llbracket N_1 \rrbracket^W \dots \llbracket N_n \rrbracket^W)$  measures the exact length of the machine run on  $(N_1 \dots N_n, M)$ , for closed terms.

Recalling Remark 6.1.14, we note that a similar proof to 6.1.13 shows that the weak interpretation is indeed well-defined – that is, the interpretation of a term is indeed a monotonic function.

**Definition 6.6.1.** For each term  $\Gamma \vdash M : \bar{\sigma}_A \Rightarrow \bar{\tau}_A$  and valuation  $v$ , we inductively define a *weak interpretation* function

$$\llbracket \Gamma \vdash M : \tau \rrbracket_v^W : \llbracket \tau \rrbracket.$$

We omit the context and/or types of terms inside an interpretation when it is clear. Each case is defined in the same way as the strong interpretation  $\llbracket - \rrbracket$  in the previous section, except for the application case, which is given by

$$\begin{aligned} \llbracket [N]a. M : \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_v^W(s) &= (1 + n, t) \\ \text{where } (n, t) &= \llbracket M : a(\rho) \tilde{\sigma}_A \Rightarrow \vec{\tau}_A \rrbracket_v^W(s, a(\llbracket N : \rho \rrbracket_v^W)) \end{aligned}$$

We define  $\llbracket \Gamma \vdash M : \tau \rrbracket^W$  to be  $\llbracket \Gamma \vdash M : \tau \rrbracket_w^W$ , for the *least valuation*  $w = \{x \leftarrow 0_\sigma \mid x : \sigma\}$  given by the minimal elements. The interpretation of a stack of terms is defined element-wise, and the interpretation of a memory  $S_A$  is given by  $\Pi_{a \in A} \llbracket S_a \rrbracket$ .

We now recall the substitution lemma from the previous section, applied instead to the *weak* interpretation. The proof of this lemma is similar to the previous one.

**Lemma 6.6.2** (Substitution). *For terms  $\Gamma, x : \omega \vdash M : \tau$  and  $\Gamma \vdash N : \omega$  and valuations  $v$ , we have*

$$\llbracket \{N/x\}M \rrbracket_v^W = \llbracket M \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket_v^W\}}^W .$$

*Proof.* We omit the full proof here, but reference the Substitution Lemma 6.2.3 proved in detail for the strong interpretation, which differs only in the extra count included in the application case (as do the lemmata of Weakening and Sequencing, which it relies on).  $\square$

The following proof verifies that the weak interpretation counts exactly the number of machine steps required to evaluate a term. It relies essentially on machine termination.

**Proposition 6.6.3.** *For any closed term  $M : \tilde{\sigma}_A \Rightarrow \vec{\tau}_A$  and memories of closed terms  $S_A : \vec{\sigma}_A$  and  $T_A : \vec{\tau}_A$ , we have that*

$$(S_A, M) \Downarrow (T_A, \star) \quad \text{implies} \quad \llbracket M \rrbracket^W(\llbracket S_A \rrbracket^W) = (n, \llbracket T_A \rrbracket^W) ,$$

where  $n$  is the number of machine transitions until  $(S_A, M)$  terminates.

*Proof.* We proceed by induction on the length  $n$  of the machine run

$$\frac{(S_A, M)}{(T_A, \star)} .$$

The base case, a run of length 0, implies that  $M \equiv \star : \tilde{\tau}_A \Rightarrow \vec{\tau}_A$  and  $S_A \equiv T_A$ . Thus, it suffices to observe that

$$\llbracket \star \rrbracket^W(\llbracket S_A \rrbracket^W) = (0, \llbracket S_A \rrbracket^W) .$$

Now, let us consider such a machine run of length  $n + 1$ . Because  $M$  is closed, there are just two cases: where  $M \equiv a\langle x \rangle. M' : a(\rho) \tilde{\sigma}_A \Rightarrow \vec{\tau}_A$  and where  $M \equiv [N]a. M' : \tilde{\sigma}_A \Rightarrow \vec{\tau}_A$  (with  $N : \rho$ ). We prove the statement for each case in turn.

- The application case: let

$$(S_A, [N]a.M') \Downarrow (T_A, \star)$$

have run length  $n + 1$ . Then the run  $(S_{A \setminus \{a\}}; S_a \cdot N, M') \Downarrow (T_A, \star)$  has length  $n$ , so we can apply the inductive hypothesis to achieve

$$\llbracket M' \rrbracket^W(\llbracket S_A \rrbracket^W, a(\llbracket N \rrbracket^W)) = (n, \llbracket T_A \rrbracket^W).$$

By definition, we have

$$\begin{aligned} \llbracket [N]a.M' \rrbracket^W(\llbracket S_A \rrbracket^W) &= (m + 1, t) \\ \text{where } (m, t) &= \llbracket M' \rrbracket^W(\llbracket S_A \rrbracket^W, a(\llbracket N \rrbracket^W)). \end{aligned}$$

so altogether we have  $(m, t) = (n, \llbracket T_A \rrbracket^W)$  and thus the required result

$$\llbracket [N]a.M' \rrbracket^W(\llbracket S_A \rrbracket^W) = (n + 1, \llbracket T_A \rrbracket^W).$$

- The abstraction case, let

$$(S_{A \setminus \{a\}}; S_a \cdot N, a\langle x \rangle.M') \Downarrow (T_A, \star)$$

have run length  $n + 1$ . Then the run  $(S_A, \{N/x\}M') \Downarrow (T_A, \star)$  has length  $n$ , so we can apply the inductive hypothesis to achieve

$$\llbracket \{N/x\}M' \rrbracket^W(\llbracket S_A \rrbracket^W) = (n, \llbracket T_A \rrbracket^W).$$

By definition, we have

$$\begin{aligned} \llbracket a\langle x \rangle.M' \rrbracket^W(\llbracket S_A \rrbracket^W, a(\llbracket N \rrbracket)) &= (m + 1, t) \\ \text{where } (m, t) &= \llbracket M' \rrbracket_{v\{x \leftarrow \llbracket N \rrbracket\}}^W(\llbracket S_A \rrbracket^W), \end{aligned}$$

where  $v$  is the least valuation. Applying the Substitution Lemma 6.6.2, we get  $(m, t) = \llbracket \{N/x\}M' \rrbracket^W(\llbracket S_A \rrbracket^W)$  and so altogether we have  $(m, t) = (n, \llbracket T_A \rrbracket^W)$  and thus the required result

$$\llbracket a\langle x \rangle.M' \rrbracket^W(\llbracket S_A \rrbracket^W, a(\llbracket N \rrbracket)) = (n + 1, \llbracket T_A \rrbracket^W). \quad \square$$

**Remark 6.6.4.** Note that reduction is monotonic with respect to the weak interpretation: that is,  $\Gamma \vdash M \rightarrow M' : \tau$  implies  $\llbracket M \rrbracket^W \geq_\tau \llbracket M' \rrbracket^W$ , using a similar proof to Lemma 6.3.1.

This proof confirms the intuition that we have made explicit latent operational intuitions in Gandy's proof of strong normalisation.

## Chapter 7

# Related Literature

Landin was the first to show that the  $\lambda$ -calculus could be used as a tool for studying programming languages by using it as the target of a translation from Algol [62, 63]. Since then, there has been a huge amount of research into applications of the  $\lambda$ -calculus to the theory of programming languages. One major obstacle to its application to *real-world* programming languages is finding the right way to incorporate computational effects.

Something common to the various approaches to effectful  $\lambda$ -calculi is that they provide some way to express the *sequencing* of computations within syntax<sup>1</sup>. Approaches such as Moggi’s computational metalanguage and Call-by-Push-Value, to be discussed – are thus similar in spirit to the Sequential  $\lambda$ -calculus. Note, however, that the various approaches presented here introduce new constructors at term and type level to achieve this, as in Figure 7.1. In contrast, with SLC *decomposes* and *generalizes* the existing variable construct. At type level, there are no new constructors and the typed SLC (and FMC) can be considered a novel computational interpretation of intuitionistic logic, as shown in Chapter 4. This has the advantage of parsimony.

However, the approach of the FMC differs *significantly* from other approaches when adding effects. The major contribution of the FMC is to encode reader/writer effects using the same syntax and operational mechanisms as for higher-order functions. Confluence is retained because the beta-law then captures *both* the semantics of higher-order functions *and* the semantics of these effects.

We now review some of the large amount of related literature on effects, drawing further comparisons as they become relevant.

### 7.1 Effect-Passing Style

So-called *effect-passing* style is perhaps the most primitive form of dealing with effects, as a programming style available in the pure  $\lambda$ -calculus. We lead with an

---

<sup>1</sup>As in the introduction, the notion of “sequencing” referred to here is a semantic one, rather than in the sense of reduction order.

$$\begin{array}{c}
\frac{\Gamma \vdash M : A}{\Gamma \vdash \eta(M) : TA} \qquad \frac{\Gamma \vdash N : TA \quad \Gamma, x : A \vdash M : TB}{\Gamma \vdash \text{let}_T x \leftarrow N \text{ in } M : TB} \\
\\
\frac{\Gamma \vdash_v M : A}{\Gamma \vdash_c \text{return } M : FA} \qquad \frac{\Gamma \vdash_c N : FA \quad \Gamma, x : A \vdash_c M : \underline{B}}{\Gamma \vdash_c N \text{ to } x. M : \underline{B}} \\
\\
\frac{\Gamma \vdash_v M : U\underline{B}}{\Gamma \vdash_c \text{force } M : \underline{B}} \qquad \frac{\Gamma \vdash_c M : \underline{B}}{\Gamma \vdash_v \text{thunk } M : U\underline{B}} \\
\\
\frac{\Gamma, !A \vdash B}{\Gamma, A \vdash B} \qquad \frac{! \Gamma \vdash B}{! \Gamma \vdash !B}
\end{array}$$

Metalanguage:	$(A \rightarrow B)_v = A_v \rightarrow TB_v$
Call-by-Push-Value:	$(A \rightarrow B)_n = UA_n \rightarrow B_n \quad (A \rightarrow B)_v = U(A_v \rightarrow FB_v)$
Linear Logic:	$(A \rightarrow B)_n = !A_n \multimap B_n \quad (A \rightarrow B)_v = !(A_v \multimap B_v)$

Figure 7.1: Typing rules for other approaches to sequencing computation

example: *state-passing* style. To model global state in the  $\lambda$ -calculus, it suffices to extend every function with an extra argument and an extra output, modelling the state at the time the function is called, and the updated state at the time the function returns. That is, a procedure of type  $A \rightarrow B$  making use of state with type  $S$  is modelled as a function of type

$$S \times A \rightarrow S \times B,$$

where the state is then threaded through every computation.

Denotationally, what happens here is in some ways similar to the FMC: stateful processes become pure functions between appropriately enlarged types, as discussed in the introduction to Chapter 4 (and this in part accounts for the Cartesian closed semantics of the FMC given later in that chapter). However, the approach of the FMC is much less heavy-handed than simple state-passing style: permutation equivalence and the admissibility of type expansion means that the programmer need not explicitly record extra input and output types representing, say, initial and final state for terms where it is not used. Yet, these pure terms may still be composed with stateful terms. Further, whereas in state-passing style a type  $S$  of the state is fixed, in the FMC the type held at any given memory cell (modelled by a given location) can vary.

A more complicated, but common, example is that of *continuation-passing* style [102, 110]. Here, every function takes an extra argument, its continuation, which is called with the result of that function, thus making control flow explicit. Thus, a function of type  $A \rightarrow B$  becomes a function

$$(B \rightarrow \perp) \rightarrow A \rightarrow \perp,$$

where  $B \rightarrow \perp$  is the type of continuations of  $B$ . This gives explicit access to the continuation of a function, allowing for expression of various control structures. Similarly, one can use *exception*-passing style to model programs with exceptions, *etc.*

A major downside of effect-passing style is that it requires a restructuring of the entire program to account for effects.

## 7.2 Monads and Moggi's Computational Metalanguage

In Moggi's seminal work of '89, he axiomatized the common structure found in effect-passing styles using the categorical structure of a *strong monad* [76][85]. Since then, monads have been widely adopted as a technique for structuring effectful functional programs, most notably in Haskell [53, 117, 118].

Of course, a monad is just a monoid in the category of endofunctors. A *strong* monad on a monoidal category is a monad equipped with a natural transformation, as below, called the *strength*, satisfying certain coherence conditions.

$$\text{str}_{A,B} : A \otimes TB \rightarrow T(A \otimes B)$$

Moggi notes that the usual semantic interpretation of the type  $A \rightarrow B$  as the set of total functions from  $A$  to  $B$  is inadequate for modelling programs with side-effects. He instead developed a variant of the CBV  $\lambda$ -calculus called the *computational metalanguage* (*ML*), where the type of such programs is given as  $A \rightarrow TB$ , for some strong monad  $T$ . Here,  $B$  denotes *values* of type  $B$ , whereas  $TB$  denotes *computations* which return values of type  $B$ . The type and term constructs for  $T$  in the meta-language are as follows.

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \eta(M) : TA} \quad \frac{\Gamma \vdash N : TA \quad \Gamma, x : A \vdash M : TB}{\Gamma \vdash \text{let}_T x \leftarrow N \text{ in } M : TB}$$

These rules correspond to the *unit* and *Kleisli composition* associated with the monad  $T$ , respectively. As expected, the categorical models of the *ML* are given by Cartesian closed categories equipped with a strong monad. In this sense, *ML* is a *pure* calculus.

While in the metalanguage side-effecting computations are explicitly marked as having monadic type, many real-world programming languages do not do this. In order to model such languages, Moggi introduced a second language, the *computational  $\lambda$ -calculus*, often denoted  $\lambda_c$ , intended as a effectful call-by-value programming language. Intuitively, this can be thought of as *ML* with the monadic type constructor  $T$  made invisible. Thus a program  $\Gamma \vdash M : A$  is a possibly effectful computation of type  $A$ . As such,  $\lambda_c$  is naturally interpreted in the Kleisli category of the monad  $T$ .<sup>2</sup> More precisely, models of  $\lambda_c$  are given by a Cartesian category  $\mathbb{C}$  equipped

---

<sup>2</sup>To be more precise, the interpretation of  $\Gamma \vdash M : A$  is given by a morphism from  $[\Gamma] \rightarrow T[A]$ . As such,  $\lambda x.M : \top \rightarrow A$  where  $x : \top$  is not free in  $M : A$  is given type isomorphic to  $T[A]$ . In particular, the function type is interpreted as possibly effectful.

with a strong monad  $T$  and *Kleisli exponentials*: that is all exponents of the form  $A \Rightarrow TB$  in  $\mathbb{C}$ , which is just enough to ensure an appropriate exponential structure on the Kleisli category of  $T$ .

A major strength of monads is that they capture a remarkable variety of effects. We give just a few example of (necessarily strong) monads on the category **Set**:

Effect	Monad
Input	$I \rightarrow -$
Output	$- \times O^*$
State	$S \rightarrow (- \times S)$
Non-determinism	$\mathcal{P}(-)$
Continuations	$(- \rightarrow \perp) \rightarrow \perp$

where we have a set of inputs  $I$ , a set of outputs  $O$ , where  $O^*$  is the free monoid over  $O$ , a set of states  $S$ ,  $\perp$  is the empty set, and  $\mathcal{P}$  is the powerset operator.

Note how, when taking  $T$  to be the state or continuation monad,  $A \rightarrow TB$  indeed corresponds to the state- or continuation-passing style interpretation of  $A \rightarrow B$  given in the previous subsection, modulo currying.

However, monads have a fundamental problem: they do not compose. That is, given two monads  $S$  and  $T$ ,  $ST$  is not a monad in general. In some specific cases, there is a *distributive law* between  $S$  and  $T$ : a natural transformation  $l_A : STA \rightarrow TSA$ , satisfying certain coherence conditions, which makes  $ST$  into a monad. Such laws do not always exist [55]. Furthermore, when considering more than two monads, *iterated* distributive laws are required [17, 35], and even for the case of three monads, these are required to satisfy large commutative diagrams.

In practice, multiple effects are combined by building a stack of *monad transformers* (which take a monad as an argument and return a monad as a result), with each layer adding both computational overhead and mental overhead for the programmer. In fact, some practical situations require effects to be interleaved in such a way as for no fixed stack of transformers to suffice [55].

The translation of the metalanguage into the SLC extends the encoding of the CBV  $\lambda$ -calculus  $(-)_v$ , given in Section 1.3.4 of the introduction, as follows. Recall that evaluation of the translation of a CBV  $\lambda$ -term returns a value, which is pushed to the stack. Corresponding to the intuition given above, we translate  $T(A)$  as the type of computations which return values of type  $A$  to the stack, with the  $\text{let}_T$  construct as sequencing of computations.

$$T(A)_v \triangleq \Rightarrow A_v \quad \eta(M)_v \triangleq [M_v] \quad (\text{let}_T x \Leftarrow N \text{ in } M)_v \triangleq N_v ; \langle x \rangle . M_v$$

Related, we mention also Moggi's *computational  $\lambda$ -calculus*, which gives the internal language for the Kleisli category of a strong monad on category with products.

In comparison with the FMC, the monadic approach has the undeniable benefit of generality: it accounts for strictly more effects than the FMC (for example, continuations). However, for the effects the FMC does deal with, it promises a less

problematic approach to composition. See Section 4 of [9] for an example comparing the two.

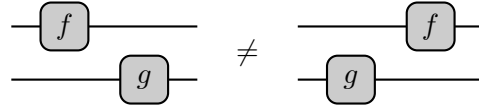
In the introduction of [88], Plotkin and Power argue that Moggi’s original work does not give a satisfactory account of the operational semantics of effects. They indicate that *operations*, such as *raise* for exceptions, *read* and *write* for input and output, and *lookup*, *update* for state, are the source of effects and they give an operational semantics for a subset of *algebraic* effects (Section 2, *loc. cit.*), which we discuss later, in Section 7.6. However, the FMC gives an alternative account of the operational semantics of the relevant operations for reader/writer effects; in fact, an account in which operational semantics is *primary*, as detailed in Section 1.1.2 of the introduction to this thesis.

Other related work includes [114], which develops the study of comonads with applications to modelling stream-based computation. Filinski’s PhD thesis introduces a notion of *monadic reflection* to deal with effects [30].

### 7.3 Pre-monoidal and Freyd Categories

Pre-monoidal categories, introduced by Power and Robinson [98, 97, 96], are like monoidal categories, except where the *interchange* law may fail: that is, unlike in a monoidal category, we may have the following inequation, also given in string diagrams below.

$$(f \otimes \text{id}); (\text{id} \otimes g) \neq (\text{id} \otimes g); (f \otimes \text{id})$$



Equivalently stated, the pre-monoidal tensor  $\otimes$  need not be *bifunctorial*, although it must still be functorial in each of its arguments separately. The failure of interchange enforces a strong notion of *sequentiality*. That is, the pre-monoidal tensor specifies the order of evaluation of its arguments, which in general will affect semantics in the presence of effects. For example, consider  $f$  and  $g$  above to be processes with some side-effect, such as printing to output. Then it matters whether  $f$  or  $g$  is evaluated first, necessitating the inequation given above.

Indeed, the Kleisli category of a strong monad  $T$  on some Cartesian category has a pre-monoidal tensor defined as follows. It inherits the Cartesian product’s structure on objects and, for a morphism  $f : A \rightarrow TB$ , has left- and right-actions given by

$$\begin{aligned} f \otimes C &\triangleq A \times C \xrightarrow{f \times C} TB \times C \xrightarrow{\text{str}} T(B \times C) \\ C \otimes f &\triangleq C \times A \xrightarrow{C \times f} C \times TB \xrightarrow{\text{str}} T(C \times B) \end{aligned}$$

The pre-monoidal product given above is monoidal (*i.e.*, satisfies interchange) if and



only if the monad  $T$  is *commutative*, *i.e.*, the two canonical morphism of type

$$TA \otimes TB \rightarrow T(A \otimes B),$$

formed from the strength of  $T$  and its multiplication, coincide. This is *not* the case for many common effects.

Building on this simple idea, Power and Robinson then introduced (closed) Freyd categories [96][68], axiomatizing the Kleisli category associated with a strong monad over a Cartesian category. These are (higher-order analogues of) symmetric pre-monoidal categories which are additionally equipped with a wide Cartesian subcategory representing *pure* functions, which are thus duplicable, discardable, and satisfy interchange with respect to every other process. *Impure* processes, which live outside of this subcategory, however, cannot be expected to satisfy any of these properties in general.

More formally, a Freyd category is given as a pre-monoidal functor  $J : \mathbb{C} \rightarrow \mathbb{P}$ , where  $\mathbb{C}$  is a Cartesian category and a  $\mathbb{P}$  is a pre-monoidal category with the same objects as  $\mathbb{C}$ , such that  $J$  is identity-on-objects. This models the scenario we have described above where, given a Cartesian category  $\mathbb{C}$  and a strong monad  $T$  on  $\mathbb{C}$ , there is a canonical free functor from  $\mathbb{C}$  to the Kleisli category of  $T$ . If one further asks that the functor  $J(- \times A)$  has a right adjoint, one recovers the notion of a *closed* Freyd category which is equivalent to the notion of a  $\lambda_c$  model described in the previous section.

The strict form of sequentiality imposed by pre-monoidality is only necessary if processes interact through side-effects which are hidden from the type system. An example of this is detailed in the opening of this thesis, in Subsection 1.1.2. Because the FMC makes these explicit, they are immediately accounted for in the semantics, which is then a standard Cartesian (closed) category.

Other related work includes Hughes' *Arrows* [45, 4, 69, 48, 41, 49], which are a generalization of strong monads, *Cartesian effect categories* [26], and Schewimer and Jeffrey's work on graphical models of effectful programming languages based on a version of string diagrams for pre-monoidal categories [105, 51], and the more recent work of Roman [104].

## 7.4 Call-by-Push-Value

Levy, in his PhD thesis, introduced Call-by-Push Value (CBPV) [66, 67] with the slogan a “synthesis of functional and imperative programming”. Here, a strict distinction is drawn between values and computations, with explicit coercions between the two. Following Levy, we will underline computation types. Two new type constructors are introduced:  $U$ , taking computations to values, and  $F$ , taking values to computations, so that  $U\underline{B}$  is the value type of *thunks* of  $B$ , *i.e.*, frozen computations, and  $F A$  is the type of computations which return values of type  $A$ .

CBPV may be seen as arising from a decomposition of Moggi's strong monad into a (strong) adjunction between a category of *values* and a category of *stacks*, a

third natural judgement of CBPV [65].<sup>3</sup> Concrete examples of the decomposition of Moggi's monads into adjunctions are as follows, where Moggi's monad  $T$  is formed from the composite  $UF$ . Again, note these correspond to the given monads (and to state- and continuation-passing transformations) up to the currying isomorphism.

Monad	$U$	$F$
State	$S \rightarrow -$	$S \times -$
Continuations	$(- \rightarrow \perp)$	$(- \rightarrow \perp)$

The typing rules extend the  $\lambda$ -calculus with rules associated to  $U$  and  $F$ , given below. Note, there are two different judgements,  $\vdash_c$  for computations and  $\vdash_v$  for values. The rule for sequencing computations,  $N$  to  $x.M$ , takes a similar form to Moggi's  $\text{let}_T x \leftarrow N$  in  $M$ .

$$\begin{array}{c}
\frac{\Gamma \vdash_v M : A}{\Gamma \vdash_c \text{return } M : FA} \quad \frac{\Gamma \vdash_c N : FA \quad \Gamma, x : A \vdash_c M : \underline{B}}{\Gamma \vdash_c N \text{ to } x.M : \underline{B}} \\
\\
\frac{\Gamma \vdash_v M : U\underline{B}}{\Gamma \vdash_c \text{force } M : \underline{B}} \quad \frac{\Gamma \vdash_c M : \underline{B}}{\Gamma \vdash_v \text{thunk } M : U\underline{B}}
\end{array}$$

One of the main aims of CBPV is to allow the study of the operational and denotational semantics of CBN and CBV in a unified framework. Indeed, it allows to capture both the CBN and CBV evaluation strategies of the  $\lambda$ -calculus, by thunking arguments or functions, respectively, giving rise to two different translations of the  $\lambda$ -calculus into CBPV. In this respect, it is very similar to the SLC. The two translations of the function type of the  $\lambda$ -calculus are given below.

$$(A \rightarrow B)_n = U\underline{A}_n \rightarrow \underline{B}_n \quad (A \rightarrow B)_v = U(A_v \rightarrow FB_v)$$

Note that, in CBPV, unlike CBV, abstractions are considered *computations* and not values, and the computation function type is of the form  $A \rightarrow \underline{B}$ . Levy designed CBPV so that its CBN and CBV translations preserve “all known” semantics of (potentially effectful) CBN and CBV  $\lambda$ -calculi. Note, this is an empirical claim, and indeed CBPV arose as a result of a search for commonalities in existing CBN and CBV semantics. The two translations from the CBN and CBV  $\lambda$ -calculus are as follows (Section 7, [67]).

$$\begin{array}{ll}
x_n \triangleq \text{force } x & x_v \triangleq \text{return } x \\
(MN)_n \triangleq M_n @ (\text{thunk } N_n) & MN \triangleq M_v \text{ to } x. (N_v \text{ to } z. (\text{force } z @ x)) \\
(\lambda x.M)_n \triangleq \lambda x.M_n & (\lambda x.M)_v \triangleq \text{return thunk } \lambda x.M_v
\end{array}$$

The distinction between computations and values in the !FMC, detailed in Section 3.1 of Chapter 3, was inspired by the same in CBPV, and similarly for the

---

<sup>3</sup>Recall that every monad arises from an adjunction. In fact, it may arise from many adjunctions in general. Given a monad  $T$  on  $\mathbb{C}$ , there are *two* canonical such adjunctions: one between  $\mathbb{C}$  and the Kleisli category of  $T$  and one between  $\mathbb{C}$  and the Eilenberg-Moore category of  $T$ .

coercions force and thunk. It is also of interest to compare the CBN and CBV translations for the FMC, as given in Section 1.3.4 of the introduction, to the CBN and CBV translations for CBPV above.

The following translation extends our translation  $(-)_n$  of the CBN  $\lambda$ -calculus, referenced above.<sup>4</sup> Note how the **return** and sequencing constructs mirror our translation of the unit and Kleisli composition constructs in  $ML$ . The value functor  $U$  is made redundant by the structure of the SLC's stack types.

$$\begin{array}{ll} F(A)_n \triangleq \Rightarrow A_n & \vec{\tau} \triangleq U\tau_1 \dots U\tau_n \\ N \text{ to } x.M \triangleq \textcolor{red}{N}; \langle x \rangle.M & \text{return } V \triangleq \textcolor{red}{[V]} \\ \text{thunk } M \triangleq \textcolor{red}{!M} & \text{force } V \triangleq \textcolor{red}{?V} \end{array}$$

Another similarity between CBPV and the SLC is that they can both be considered to be based the operational semantics of a stack machine (Section 5, [67]). The CCC semantics of the SLC can be recovered as a trivial CBPV model based on the identity monad. Other work includes extending CBPV with effects to a dependently typed setting [115]

## 7.5 Linear Logic and the Bang Calculus

While discussing decompositions of CBN and CBV, it would be remiss not to mention Linear Logic [32]. Linear Logic allows control over the structural rules of contraction and weakening via an *exponential* modality ( $!$ ). For formulae tagged with this modality, these rules are permissible, whereas they are forbidden for formulae in general (which are *linear*). We present the intuitionistic rules of LL involving the *exponential*  $!$  below.

$$\frac{\Gamma \vdash B}{\Gamma \vdash !B} \quad \frac{\Gamma, !A \vdash B}{\Gamma, A \vdash B} \quad \frac{\Gamma, !A \vdash B}{\Gamma, !A, !A \vdash B} \quad \frac{\Gamma, !A \vdash B}{\Gamma \vdash B}$$

Note how the first two rules give  $!$  the structure of a *co-monad*, while the latter two give the structural rules of contraction and weakening.

Girard developed two translations of intuitionistic logic into Linear Logic [32],  $(-)_n$  and  $(-)_v$ , corresponding to encodings of the CBN and CBV  $\lambda$ -calculus, respectively. These two translations decompose the intuitionistic implication differently, similarly to CBPV.

$$(A \rightarrow B)_n = !A_n \multimap B_n \quad (A \rightarrow B)_v = !(A_v \multimap B_v)$$

Note that  $!$  has the structure of a *co-monad*, and the CBN-translation corresponds to its co-Kleisli category.

The Bang calculus [29, 36, 15] can be considered an untyped version of CBPV, however it was developed by Ehrhard and Guerrieri from the distinct perspective of

---

<sup>4</sup>It is clear how to map that translation into the FMC with values.

Linear Logic. Their syntax similarly makes the a distinction between computations and values and introduces the Linear Logic operators of *box* (!), introducing the exponential comonad , which marks terms which may be duplicated or erased, and *dereliction* (eliminating the exponential).

$$\begin{aligned} V, W &\triangleq x \mid !M \\ M, N &\triangleq V \mid MN \mid \lambda x.M \mid \mathbf{der} M \end{aligned}$$

Indeed, the calculus has the expected reduction rule:  $\mathbf{der} !M \rightarrow M$ . However, beta reduction is restricted to  $(\lambda x.M)V \rightarrow \{V/x\}M$ , *i.e.*, redexes can only be fired when their argument has been reduced to a value.

It is shown that these two operators correspond to the *force* and *thunk* operators of CBPV, respectively, giving a useful operational intuition to these Linear Logic constructs, and a relationship between CBPV and Linerar Logic. That is, viewing the type/term constructor ! as *U/thunk*, we can see that the CBN translation delays evaluation of its argument, whereas in the CBV translation (dubbed by Girard the“boring” translation), evaluation of the function itself is delayed. The CBPV functor *F* is, however, silent in this presentation.

The Bang calculus factorizes Girard’s translations, giving an intermediate language which can encode both the CBN and CBV  $\lambda$ -calculus. The original idea of CBN and CBV corresponding to Girard’s translations dates back to [72]. These translations to the Bang calculus are given as follows.

$$\begin{aligned} x_n &\triangleq \mathbf{der} x & x_v &\triangleq x \\ (MN)_n &\triangleq M_n(!N_n) & MN &\triangleq !(\lambda x.M_v) \\ (\lambda x.M)_n &\triangleq \lambda x.M_n & (\lambda x.M)_v &\triangleq (\mathbf{der} M_v)N_v \end{aligned}$$

In the SLC with values, we translate their values in the obvious way, and for computations have the following.

$$V \triangleq [V] \quad (MN)_n \triangleq N; M \quad (\lambda x.M)_n \triangleq [! \langle x \rangle . M] \quad \mathbf{der} M = M; \langle x \rangle . ?x$$

Note how their coercion from values to computations reflects the silence of their *F* with respect to CBPV, which in our translation is no longer silent: it mirrors our translation of the *return* of CBPV.

In a similar spirit, We mention here also the (enriched) effect calculus [27, 28], which augments *ML* with constructs from linear logic. Other links between CBPV and Linear Logic are given by Benton’s decomposition of the co-monad (!) into an adjunction between a Cartesian category of values and a category of (linear) computations [12]. This is in some sense dual to CBPV’s decomposition of the monad in the metalanguage into an adjunction.

## 7.6 Universal Algebra and Algebraic Effects

Perhaps lacking in Moggi’s original work on effectful computation is a treatment of the *operations* which generate effects, and of their *operational semantics*. Examples of operations are given by *raise* for exceptions, *read* and *write* for input and output, *choose* for probabilistic or non-deterministic choice, and *lookup* and *update* for state, all familiar from real-world programming languages [11].

Universal algebra is the study of *algebraic theories*, which are generated by a signature of *operations*  $\Sigma$ , like those given above, and a set of *equations*  $\mathcal{E}$ . Linton was first to formalize a general correspondence between universal algebra and monads: any algebraic theory  $(\Sigma, \mathcal{E})$  gives rise to a “free model” monad  $T_\Sigma$  on **Set** whose Eilenberg-Moore category is equivalent to the category of models of that theory [47]. For example, the theory of monoids has a corresponding “free monoid monad” whose Eilenberg-Moore category is equivalent to the usual category of monoids.

Plotkin and Power [90, 89, 86, 88, 85, 89] later worked to generalize this connection by replacing the category **Set** with any  $\lambda_c$  model, in order to give a perspective on computational effects with *operations* at the centre. In order to do this, it was natural to work with *Lawvere theories*, which are a categorical interpretation of the notion of algebraic theory which naturally allows models of a theory to be taken in any category with products.

The correspondence mentioned above can be restated in these terms: the category of Lawvere theories is equivalent to the category of *finitary* monads on **Set**.<sup>5</sup> To ease presentation, we work with an intermediate generality between that of the original correspondence and that of Plotkin and Power: we will work with *infinitary* algebraic theories. The corresponding result is that the category of *infinitary* Lawvere theories is equivalent to the category of *arbitrary* monads on **Set**. In particular, we work on **Set** throughout, rather than an arbitrary category with products.

We will now discuss in detail the three perspectives mentioned above: universal algebra, Lawvere theories, and monads. Fixing an algebraic theory  $(\Sigma, \mathcal{E})$ , each perspective gives a path from left to right in the diagram below, and we will detail how they are equivalent.<sup>6</sup> In particular, we will explain how a Lawvere theory and a monad can be generated from  $(\Sigma, \mathcal{E})$  and how the models of the theory correspond to the Eilenberg-Moore algebras of the monad. Finally, we relate this story back to computational effects and describe how the new perspective of universal algebra accounts for operational considerations.

---

<sup>5</sup>Note, although most monads corresponding to computational effects are finitary, there are notable exceptions such that of the continuation monad.

<sup>6</sup>We write simply  $\Sigma$  instead of  $(\Sigma, \mathcal{E})$  in subscripts, leaving the equations implicit, and elide subscripts altogether in the labels of the arrows of the diagram so as to avoid overly heavy notation.

$$\begin{array}{ccccc}
& & \text{Kl}(T_\Sigma) & \xrightarrow{X \mapsto (TX, \mu)} & \text{EM}(T_\Sigma) \\
& \nearrow F & \parallel & & \searrow U \\
\text{Set} & \xrightarrow{F} & \text{freeModel}_\Sigma & \xrightarrow{\quad} & \text{Model}_\Sigma \xrightarrow{U} \text{Set} \\
& \searrow X \mapsto \star^{|X|} & \parallel & & \nearrow M \mapsto M(\star) \\
& & \mathfrak{L}_\Sigma^{op} & \xrightarrow{\star \mapsto \mathfrak{L}(\star^n, -)} & [\mathfrak{L}_\Sigma, \text{Set}]
\end{array}$$

**Universal Algebra.** Algebraic theories are traditionally generated by a signature  $\Sigma$  of operations, each with a given *arity*  $n \in \mathbb{N}$ , and set of equations  $\mathcal{E}$  over terms generated by these operations and a set of free variables. For example, the theory of a monoid is given by two operations: a nullary unit  $u$  and a binary multiplication  $(\cdot)$ , subject to the following equations:<sup>7</sup>

$$u \cdot x = x \quad x \cdot u = x \quad (x \cdot y) \cdot z = x \cdot (y \cdot z) .$$

A second example is the theory of a pointed set, generated by a single nullary operation and no equations. Other examples are given by groups, semi-lattices, and rings.

A *model* of a theory  $(\Sigma, \mathcal{E})$  is given by a choice of set  $X$  and for each  $n$ -ary operation  $op$  in  $\Sigma$  an interpreting function  $f : X^n \rightarrow X$  such that the resulting interpretation of the equations in  $\mathcal{E}$  hold. Given an appropriate notion of model homomorphism, we can thus form a category of models of  $(\Sigma, \mathcal{E})$ , which we denote  $\text{Model}_\Sigma$ . This is the expected category of structures: the category of models of the theory of monoids is the usual category of monoids, and so on.

There is a forgetful functor  $U : \text{Model}_\Sigma \rightarrow \text{Set}$  sending a model to its underlying set. This functor has a left adjoint  $F : \text{Set} \rightarrow \text{Model}_\Sigma$  which sends a set  $X$  to the freely generated model of  $(\Sigma, \mathcal{E})$  over  $X$ . For example, for the theory of monoids above,  $F$  sends  $X$  to terms (or trees) inductively generated by operations of  $\Sigma$ , with free variables (or leaves) taken from  $X$ :

$$t, t' ::= x \in X \mid u \mid (t \cdot t') ,$$

which are finally quotiented by the equations in  $\mathcal{E}$ .<sup>8</sup> The full subcategory of *free* models generated in this way is denoted  $\text{freeModel}_\Sigma$  in the diagram above.

While universal algebra traditionally deals with *finitary* theories, some generalization was required in order to model computational effects of interest. Reasonably straightforwardly, one can consider *infinitary* theories – that is, where operations may have arity given by (the cardinality of) any set.

**Lawvere Theories.** The definition of a theory as given above is very syntactic: there may be many possible choices of generating operations and equations which

<sup>7</sup>More formally, one would write *e.g.*,  $x \cdot u$  as  $x \vdash \cdot(u(), x)$ : we leave implicit the free variables, use infix notation for multiplication and omit the brackets for the nullary unit operation.

<sup>8</sup>The operation  $\cdot$  is unimaginatively interpreted as the map  $(t, t') \mapsto t \cdot t'$ .

result in the same category of models. This can be a benefit if one cares about operational semantics, but for a more abstract perspective one might consider Lawvere theories. These are a categorical reformulation of the notion of algebraic theory which is invariant, *i.e.*, independent of any particular choice of generators.

A Lawvere theory  $\mathcal{L}$  is a category with products  $(\mathbb{C}, \times, 1)$  with a distinguished object  $\star$  such that every object of  $\mathbb{C}$  is isomorphic to  $\star^n$  for some  $n \in \mathbb{N}$ . Its objects can be considered as abstract *arities*, with *infinitary* Lawvere theories allowing arity  $n$  to be (the cardinality of) any set. Given an algebraic theory  $(\Sigma, \mathcal{E})$ , a corresponding Lawvere theory  $\mathcal{L}_\Sigma$  can be generated by taking hom-sets  $\mathcal{L}_\Sigma(\star^n, \star)$  to be the set of freely generated terms of a theory (modulo equations) with  $n$  free variables.<sup>9</sup> Indeed, every Lawvere theory arises from some algebraic theory in this way.

A model of a Lawvere theory  $\mathcal{L}$  is a (product preserving) functor  $M : \mathcal{L} \rightarrow \mathbf{Set}$ , and the functor category  $[\mathcal{L}, \mathbf{Set}]$  is thus the category of models. This category has an obvious forgetful functor from  $U : [\mathcal{L}, \mathbf{Set}] \rightarrow \mathbf{Set}$  which sends a model  $M : \mathcal{L} \rightarrow \mathbf{Set}$  to  $M(\star)$ . This functor has a left adjoint  $F : \mathbf{Set} \rightarrow [\mathcal{L}, \mathbf{Set}]$  which, as in the diagram above, factorizes as the composition of a functor sending sets  $X$  to  $\star^{|X|}$  in  $\mathcal{L}_\Sigma^{op}$  and the Yoneda embedding of  $\mathcal{L}^{op}$  into  $[\mathcal{L}, \mathbf{Set}]$ .

It can be seen that this perspective is equivalent to the standard perspective of universal algebra: in particular that  $\mathcal{L}_\Sigma^{op}$  is equivalent to  $\mathbf{freeModel}_\Sigma$ .<sup>10</sup> Given this, note that a choice of functor  $M : \mathcal{L}_\Sigma \rightarrow \mathbf{Set}$  packages the same data as a model of  $(\Sigma, \mathcal{E})$ : the functor  $M$  is determined by where it sends the generating object  $\star$  and generating operations  $op \in \mathcal{L}(\star^n, \star)$ , and this information constitutes a model.

Unlike monads, Lawvere theories can be composed, *e.g.*, by the taking tensor or sum of theories, and so this avenue presents a potential inroad into dealing with the combination of effects [46].

**Monads.** The correspondence between monads on  $\mathbf{Set}$  and universal algebra can be formulated as follows. Given a theory  $(\Sigma, \mathcal{E})$ , we can package the data of the signature  $\Sigma$  as a *signature endofunctor*  $S : \mathbf{Set} \rightarrow \mathbf{Set}$

$$S_\Sigma(X) \triangleq \bigoplus_{c \in \Sigma} X^{\mathbf{ar}(c)},$$

where  $\mathbf{ar}(c)$  is the arity of  $c$ . For example, for the theory of monoids, we have  $S_\Sigma(X) = X^0 + X^2$ , corresponding to the signature having one *nullary* and one *binary* operation.

The *algebras* for  $S_\Sigma$  consist of a set  $Y$  and morphism  $\alpha : S_\Sigma(Y) \rightarrow Y$ . For the example of the theory of monoids, this means a choice of function  $\alpha : 1 + Y^2 \rightarrow Y$ , or

<sup>9</sup>Hom-sets  $\mathcal{L}_\Sigma(\star^n, \star^m) \cong \mathcal{L}_\Sigma(\star^n, \star)^m$  are thus  $m$ -tuples of terms with  $n$  variables and composition is given by substitution; thus the category is determined by this choice.

<sup>10</sup>Let  $\underline{n}$  denote a set with  $n$  elements, for the finitary case. The infinitary case generalizes easily. Let the equivalence send  $\star^n$  to the free model over  $\underline{n}$ ,  $F(\underline{n})$ , which is surjective on objects. Model homomorphisms  $\mathbf{freeModel}_\Sigma(F\underline{1}, F\underline{n})$  are determined by where they send the variables of  $F\underline{1}$  (by freeness of  $F\underline{1}$ ), of which there is one. So the set of model morphisms is given by each possible choice of  $F\underline{n}$ , *i.e.*, each element of the free model of  $(\Sigma, \mathcal{E})$  on  $n$  variables. Recall that  $\mathcal{L}(\star^n, \star)$  is the set of terms generated by  $\Sigma$  with  $n$  free variables, modulo  $\mathcal{E}$ . Therefore there is a bijection between  $\mathcal{L}(\star^n, \star)$  and  $\mathbf{freeModel}_\Sigma(F\underline{1}, F\underline{n})$ .

equivalently a choice of a pair of functions  $\alpha_l : 1 \rightarrow Y$  and  $\alpha_r : Y^2 \rightarrow Y$ . That is, the algebras of  $S_\Sigma$  consist of a set with the operations of a monoid, but nothing to specify that the operations must satisfy the equations of a monoid. In fact, the category of  $S_\Sigma$ -algebras is the category of models of the algebraic theory given by the *signature* of monoids *without* any equations.<sup>11</sup> For example, the theory of a pointed set, which has no equations associated, generates a signature endofunctor  $S_\Sigma(X) = X^0$  and its category of  $S_\Sigma$ -algebras is in fact equivalent to the category of pointed sets.

In order to account for the presence of equations we must consider monads. Given a signature endofunctor  $S_\Sigma$ , we can define the *free monad*  $T_\Sigma : \mathbf{Set} \rightarrow \mathbf{Set}$  over it as follows:

$$T_\Sigma(A) \triangleq \mu X. A + S_\Sigma(X)$$

where  $\mu$  is the least fixpoint operator. In particular, the Eilenberg-Moore category  $\mathbf{EM}(T_\Sigma)$  of  $T_\Sigma$ -algebras is equivalent to the category of  $S_\Sigma$ -algebras. Intuitively, this is the free model monad: it takes a set to the set of freely generated terms over  $\Sigma$  with variables from  $A$ .<sup>12</sup> Again, consider the example given by the theory of pointed sets, which has a signature endofunctor  $S_\Sigma(X) = X^0$  and thus  $T_\Sigma$  is exception monad  $A \rightarrow A + 1$ .

We can now consider equations in this setting: an equation in  $n$  elements can be given as a parallel pair of natural transformations in  $\mathbf{Set}$  (natural in  $X$ )

$$X^n \rightrightarrows T_\Sigma(X)$$

corresponding to a pair of terms  $x_1, \dots, x_n \vdash t_1, t_2$  each with  $n$  free variables. Naturality in  $X$  reflects that an equation is specified independent of the domain it is instantiated to. Generally, then, a set of equations can be considered by copairing a number of such transformations together (one for each equation). For example, we can specify the algebraic theory of monoids as a pair of natural transformations

$$X^1 + X^1 + X^3 \rightrightarrows T_\Sigma(X)$$

since we have one free variable for the left and right unit laws, and three free variables in the associativity law. By considering the free monad  $G$  over the endofunctor  $X \mapsto X^1 + X^1 + X^3$  we can equivalently (by freeness) give a parallel pair of monad morphisms  $G \rightrightarrows T_\Sigma$  which specifies the same laws. In general we can find a monad  $G_\mathcal{E}$  and morphisms  $G_\mathcal{E} \rightrightarrows T_\Sigma$  to represent any equational theory  $\mathcal{E}$  over  $\Sigma$ . An algebra  $(X, \alpha : T_\Sigma(X) \rightarrow X)$  for  $T_\Sigma X$  satisfies the equations  $G_\mathcal{E} \rightrightarrows T_\Sigma(X)$  precisely when the two parallel morphisms

$$G_\mathcal{E}(X) \rightrightarrows T_\Sigma(X) \xrightarrow{\alpha} X$$

---

<sup>11</sup>For  $S_\Sigma$  above, this is the category of pointed magmas.

<sup>12</sup>Its multiplication takes a set of terms whose variables are taken from a set of terms to a set of terms in the obvious way, and its unit considers variables as terms in the trivial way.



are equal. Thus we obtain the monad corresponding to the algebraic theory  $(\Sigma, \mathcal{E})$  as the coequalizer of  $G_{\mathcal{E}} \rightrightarrows T_{\Sigma}$  in the category of monads. Indeed, the (Eilenberg-Moore) category of algebras of the monad so obtained is equivalent to the expected category of models of  $(\Sigma, \mathcal{E})$ . Calling this monad  $T_{\Sigma, \mathcal{E}}$ , we thus have an equivalence  $\mathbf{EM}(T_{\Sigma, \mathcal{E}}) \cong \mathbf{Model}_{\Sigma, \mathcal{E}}$ , as expressed in our original diagram.<sup>13</sup>

It is well-known that the Kleisli category of any monad  $T$ ,  $\mathbf{Kl}(T)$ , is the full subcategory of  $\mathbf{EM}(T)$  consisting of free algebras. Intuitively,  $\mathbf{Kl}(T_{\Sigma})$  is thus equivalent to  $\mathbf{freeModel}_{\Sigma}$  since they are both the full subcategories of  $\mathbf{EM}(T_{\Sigma})$  and  $\mathbf{Model}_{\Sigma}$ , respectively, containing freely generated objects. To complete the picture, recall that by construction there are canonical free and forgetful functors  $F$  and  $U$  between  $\mathbf{EM}(T)$  and  $\mathbf{Set}$  such that  $UF = T$ , and similarly for  $\mathbf{Kl}(T)$ .<sup>14</sup> Indeed, in each of the three perspectives described above, a free/forgetful adjunction arises, and each adjunction gives rise to the same monad: the *free model monad*, appearing in the diagram as any path from  $\mathbf{Set}$  to  $\mathbf{Set}$ .

**Algebraic Effects.** To relate computational effects to algebraic theories, we now give some concrete examples of *algebraic effects* and show how they are included in Moggi's computational metalanguage  $ML$ . One could similarly incorporate algebraic effects into CBPV. We consider infinitary theories but continue to work with the category  $\mathbf{Set}$ .

A particularly nice example is that of *non-deterministic choice*. This corresponds to the algebraic theory of a semi-lattice (without unit). That is, we have  $\Sigma = \{\vee : 2\}$ , together with equations

$$x \vee x = x \quad x \vee y = y \vee x \quad (x \vee y) \vee z = x \vee (y \vee z) .$$

Considering  $T(X)$  as the set of free semi-lattice terms, the operation  $\vee$  gives rise to a family of functions (in  $\mathbf{Set}$ ),  $choose_X : T(X)^2 \rightarrow T(X)$ , natural in  $X$ , given by  $(t_1, t_2) \mapsto t_1 \vee t_2$ . Another perspective is given by considering  $T$  as a free model monad: in fact, it is the finite (non-empty) powerset monad. This can be seen since the free semi-lattice over  $X$  can be considered as the set of all finite subsets of  $X$ , with  $\vee$  corresponding to the union of subsets. This interpretation explains how we are modelling non-determinism.

Another example is given by the theory of pointed sets, which we saw before corresponds to the exception monad  $T(X) = X + 1$ . This has signature  $\Sigma = \{\perp : 0\}$ , *i.e.*, a nullary operation with no equations. This operation thus gives rise to a family of functions  $raise_X : T(X)^0 \rightarrow T(X)$ , *i.e.*,  $raise_X : 1 \rightarrow X + 1$ , natural in  $X$ . Clearly we must thus have that  $raise_X$  is given by the right injection.

Finitary operations without parameters are introduced into  $ML$  with the following typing rule

$$\frac{\forall i \in \{1, \dots, n\}. \Gamma \vdash M_i : TA \quad op : n \in \Sigma}{\Gamma \vdash op(M_1, \dots, M_n) : TA}$$

<sup>13</sup>Recalling the omission of equations  $\mathcal{E}$  from the notation used there.

<sup>14</sup>In the diagram  $U$  sends algebras to their underlying carrier,  $F$  considers plain functions as trivially monadic functions, and the embedding of  $\mathbf{Kl}(T)$  into  $\mathbf{EM}(T)$  sends objects  $X$  to the free algebra on  $TX$ .

Operationally, algebraicity is reflected in evaluation contexts commuting with operations, which reflects a certain naturality property characterizing equationally specified operations. An operational semantics can then be given by ‘floating’ operations to the top of the syntax tree. For example, we would have the following rewrite.

$$\text{let } x \Leftarrow \text{op}(M_1, \dots, M_n) \text{ in } N \quad \rightsquigarrow \quad \text{op}(\text{let } x \Leftarrow M_1 \text{ in } N, \dots, \text{let } x \Leftarrow M_n \text{ in } N)$$

There exist more sophisticated type systems which track more closely the usage of effects based on these ideas [92].

**State and relationship with the FMC.** As a leading example from the study of computational effects, we now consider operations and equations for global state. First we will need to generalize our theory so that one can introduce (effectively) an infinite number of operations via *parameterization*: for example, in the theory of vector spaces, there is an operation corresponding to multiplication of a vector by a scalar, say in  $\mathbb{R}$ . One could introduce  $\mathbb{R}$ -many unary operations to model this, or one could introduce a single unary operation with a *parameter*  $r \in \mathbb{R}$ . Opting for the latter case, we generalize signatures so that each operation has an arity and an associated set from which it can draw parameters. Then for a signature with generalized operations of the form  $\text{op} : (P, A)$ , with parameter  $P$  and arity  $A$  (here, again, a set), we can give a model as a set  $X$  and for each operation a map  $f : P \times X^A \rightarrow X$ .

We now introduce the operations for global state. We work with a single memory cell, but one can consider multiple memory cells by giving a signature where operations take an extra parameter telling which cell is being referenced. One natural choice of operations is as follows: fix a set of states  $S$  and let the signature  $\Sigma$  consist of the pair of operations

$$\text{lookup} : (1, S) \quad \text{update} : (S, 1)$$

where we write 1 for the single-element set. The lookup operation is an  $S$ -ary operation, and the update operation is a unary operation which takes a parameter in  $S$ . We leave discussion of the associated equations until later. However, given these equations, the corresponding monad is of course the state monad (although it takes some work to show this: see Section 3, [85]). The theory gives rise to two families of functions, natural in  $X$ , as follows.

$$l_X : T(X)^S \rightarrow T(X) \quad u_X : S \times T(X)^1 \rightarrow T(X)$$

Recalling the isomorphism  $T(X)^S \cong S \rightarrow T(X)$ , we can read the intended meaning of  $l_X$  as taking an input continuation which tells us how to return an element of  $T(X)$  given the state  $S$ ; it then *looks up* the current value of the state, and returns the appropriate value of  $T(X)$ .

We give the following examples, due to [11], using  $\lambda$ -notation for set functions. Let  $S = \mathbb{N}$ . The first function below takes an input  $x : \mathbb{N}$ , updates the state to  $x + 1$  and then returns the original value  $x$ . Now, recall that *lookup* takes a continuation

$M : \mathbb{N} \rightarrow \mathbb{N}$  as input. The second function below continues as  $M(x) : \mathbb{N}$  with  $x$  bound to the value held in state.

$$\lambda x. \text{update}_{\mathbb{N}}(x + 1, \lambda \_ . x) : \mathbb{N} \rightarrow \mathbb{N} \quad \text{lookup}_{\mathbb{N}}(\lambda x. M(x)) : \mathbb{N}$$

Thus, the following function increments the number held in memory by one, and returns the original contents:  $\text{lookup}_{\mathbb{N}}(\lambda x. \text{update}_{\mathbb{N}}(x + 1, x))$ .

We can introduce such infinitary operations into  $ML$  using the same syntax, as follows<sup>15</sup> The operational semantics then follows that of the previous section.

$$\frac{\Gamma \vdash N : P \quad \Gamma, x : A \vdash M : TB \quad op : (P, A) \in \Sigma}{\Gamma \vdash op(N, \lambda x. M) : TB}$$

$$\text{let } x \Leftarrow op(N, \lambda y. M) \text{ in } M' \quad \rightsquigarrow \quad op(N, \lambda y. \text{let } x \Leftarrow M \text{ in } M')$$

The main result currently connecting the FMC to algebraic effects, due to Heijltjes [9], is that the algebraic laws for reader/writer effects arise from beta and permutation reduction in the FMC. We now give interpretations of *lookup* and *update* in the FMC and finally introduce the equations between *lookup* and *update* needed to complete the story.<sup>16</sup>

$$\begin{aligned} \text{lookup}(\lambda x. M) &\triangleq a\langle x \rangle. [x]a. M \\ \text{update}(v, M) &\triangleq a\langle \_ \rangle. [v]a. M \end{aligned}$$

Here, the location  $a$  is the single memory cell. In the following, we abbreviate the *lookup* and *update* operations to  $l$  and  $u$ , respectively.

$$\begin{aligned} l(\lambda y. u(y, x)) &= a\langle y \rangle. [y]a. a\langle \_ \rangle. [y]a. x \quad \rightarrow_{\beta} a\langle y \rangle. [y]a. x &=_{\eta} x \\ l(\lambda y. l(\lambda x. M)) &= a\langle y \rangle. [y]a. a\langle x \rangle[x]. M \quad \rightarrow_{\beta} a\langle y \rangle. [y]a. \{y/x\}M &= l(\lambda y. \{y/x\}M) \\ u(v, u(v', x)) &= a\langle \_ \rangle. [v]a. a\langle \_ \rangle. [v']a. M \quad \rightarrow_{\beta} a\langle \_ \rangle. [v']a. x &= u(v', x) \\ u(v, l(\lambda x. M)) &= a\langle \_ \rangle. [v]a. a\langle x \rangle. [x]a. M \quad \rightarrow_{\beta} a\langle \_ \rangle. [v]a. \{v/x\}M &= u(v, \{v/x\}M) \end{aligned}$$

Note the algebraic equations for state are Hilbert-Post complete.

For simplicity, we have kept to working with models over **Set**, but it is worth emphasizing that Plotkin and Power worked to generalize from **Set** to any  $\lambda_c$  model, so as to give the most proper setting for interpreting programs.

<sup>15</sup>Technically, the binder  $\lambda x$  is part of the operation syntax here. We have such a binder in our notation for *update*, whose continuation takes only trivial input.

<sup>16</sup>Note, these equations between operations could also naturally be expressed via their corresponding *generic effects*, denoted here by  $!a : TS$  and  $a := v : T1$  and obtained by passing the trivial continuation in to  $l$  and  $u$  [90, 89, 86]. By the Yoneda lemma, these are no less general. Further writing  $N; M$  as sugar for  $\text{let } () \Leftarrow N \text{ in } M$ , we can equivalently view the lookup and update equations as the program equations below.

$$\begin{aligned} \text{let } y \Leftarrow !a \text{ in } a := y; x &= x \\ \text{let } y \Leftarrow !a \text{ in let } x \Leftarrow !a \text{ in } M &= \text{let } y \Leftarrow !a \text{ in } M\{y/x\} \\ a := v; a := v'; x &= a := v'; x \\ a := v; \text{let } x \Leftarrow !a \text{ in } M &= a := v; \{v/x\}M \end{aligned}$$

There are at least two significant advantages to taking the (universal) algebraic rather than monadic perspective: first, finding a good presentation of an effect in terms of operations allows for easy incorporation into a programming language and gives the opportunity for an account of its operational semantics. Second, as mentioned briefly before, the Lawvere theoretic perspective allows for natural notions of composition of theories (unlike monads, which do not compose in general). Since the original work on algebraic effects, Plotkin and Pretnar also developed a notion of *effect handlers*, which in some sense are *deconstructors* of effects, contra algebraic operations which *construct* effects [92]. This development has given rise to a large body of research which we do not give an account of here [99, 16, 2].

## 7.7 Concatenative Programming and $\kappa$ -calculus

Concatenative programming is a relatively uncommon style of programming which is natural for the Sequential  $\lambda$ -calculus: sequentially composing functions which act on a stack. In one paper, concatenative programming is called “an overlooked paradigm” [40]. There are, however, several languages expressing a higher-order variant of this style: for example,  $\lambda$ -FORTH [70], Factor [82] and Joy [116]. For first investigations into type inference for such systems, see [24] and [109].

The  $\kappa$ -calculus is perhaps the only one that is remotely well-studied from a theoretical perspective [39]. It was introduced by Hasegawa as part of a semantic decomposition of the  $\lambda$ -calculus into the first-order  $\kappa$ -calculus, as a language for Cartesian categories, and the higher-order  $\zeta$ -calculus, dealing with continuations. Power and Thielecke developed the *higher-order*  $\kappa$ -calculus (Section 3.2, [95]), similar to [25] and different from the  $\zeta$ -calculus, and applied it to the study of effects. Similar to the SLC and CBPV, the  $\kappa$ -calculus comes equipped with a stack-machine based operational intuition. Indeed, the type system for the sequential  $\lambda$ -calculus follows that of the  $\kappa$ -calculus [95]. However, since the 90’s, and until recent work on the SLC, this calculus seems to have all but been forgotten.

This chapter has not come close to providing an exhaustive list of approaches to effects. We briefly mention the following approaches: *uniqueness types* [108], and representing references in the  $\pi$ -calculus [42].

## Chapter 8

# Conclusion

The results contained in this thesis, and the approach of the FMC itself, suggest many new and promising avenues of research. A handful of these are listed below.

### 8.1 Further Research

**Connection with deep inference:** The three branches of the Curry-Howard-Lambek correspondence are core to the study of the simply-typed  $\lambda$ -calculus. In this thesis, we have presented in detail the link between the simply-typed FMC and Cartesian closed categories. This implies that the type system of the FMC, which is given in natural deduction style, is thus a certain presentation of intuitionistic logic – although, to achieve this, one must quotient type derivations by the equational theory given in Chapter 4. Although this equational theory appears reasonable as an equivalence of terms, it does not appear to be a particularly natural (that is, independently justified) quotient on type derivations. So, it would seem that the link between the FMC and intuitionistic logic is not as direct as might be hoped.

A promising solution to this is to consider a presentation of intuitionistic logic in a *deep inference* proof system [37, 113, 44]. In such a system, the given equational theory appears extremely natural. In this way, we might achieve a second, convincing perspective on the Curry-Howard-Lambek correspondence. Previous work on the *atomic  $\lambda$ -calculus* has investigated computational interpretations of inference rules peculiar to deep inference (that is, the *distributor* and *medials*), and it might be interesting to investigate a computational interpretation of these rules in the setting of the FMC [38].

**String diagrams:** There has been much activity in the applied category theory community making use of string diagrams recently [81, 19, 13]. It follows trivially from the results of this thesis that the first-order fragment of the FMC gives a sound and complete term language for string diagrams. Because we take sequencing as the primitive form of composition, the language thus given is arguably more natural as a representation of string diagrams than the  $\lambda$ -calculus. One source describes a “dearth of computational tools for working with SMCs”, circa 2020

[80]. Another source bemoans the lack of general results about the complexity of decision procedures for string diagrams [23], and perhaps, by giving the right term language, the FMC could help with this. It further appears that the FMC gives a natural language for the recently investigated *higher-order* string diagrams [3]. These give a way to embed graphs into a higher-order calculus, combining graph and term structure by allowing the “boxing” of subgraphs, exposing only an interface. The higher-order machinery of the FMC gives a natural extension of the first-order fragment allowing for exactly this. One can thus factorize a string diagram by extracting sub-graphs which occur multiple times.

**Frobenius algebras:** One common structure in applied category theory, especially as it relates to string diagrams, is the Frobenius algebra. Early investigations suggest a way to capture this structure via an extension of the first-order fragment of the FMC, by taking seriously the symmetries revealed in its syntax. This would provide a term language for a larger class of string diagrams, and put the FMC in contact with recent work on signal flow graphs, Graphical Linear Algebra, and Categorical Quantum Mechanics (*loc. cit.*). Operationally, such a construct could be used to model *conditioning*, as in probabilistic programming languages, as well as provide links with quantum and reversible computation.

**Linear Logic:** It would be easy enough to extend the results of Chapter 5 to give a characterization of the *linear* FMC as the free symmetric monoidal closed category. Term calculi for linear logic have typically been variants of the  $\lambda$ -calculus which additionally require a number of *commuting conversions* in the equational theory. The FMC, by contrast, restricts nicely to a linear calculus (modulo Remark 4.2.4). From here, it is natural to ask about adding exponentials. The discussion in Section 7.5 of Related Literature, suggests that the *thunk* construct (!) may be closely related to Girard’s exponential. This, along with the question of how Girard’s translations might relate to our translations of CBN and CBV into the SLC, would make a good starting point for further investigations.

**Weaker type systems: forgetting locations and stream types:** The type system of the FMC is currently *too strong* to be practically useful for modelling effects in many circumstances. For example, when reading from a probabilistic generator, the programmer does not typically want to know *how many times* the generator was consulted, which is currently information recorded in the type system. Indeed, such information is *intensional* and not even observable in general. One obvious way to weaken the type system, recovering something more akin the the effectful  $\lambda$ -calculus, is to *forget* the type information at certain locations (*e.g.*, the random input stream). Initial investigations indicate that, in the first-order setting, ignoring the types at the location of the random stream and suitably generalizing the definition of machine equivalence results not in a Cartesian category, but in a Markov category — each term can ‘secretly’ consume random inputs, making it a stochastic process in general, rather than a function. Markov categories are a natural abstract setting for studying probabilistic processes, where we can express Bayesian networks and even do causal reasoning [50]. Of course, one also wants to

maintain all the good properties of the FMC, so, given this, the question is how far this approach can be usefully pushed. A more sophisticated approach to types for the random stream could be given by a *stream type*  $\tau^*$ , which represents a stream of  $\tau$ 's.

**Imperative Programming:** One promising avenue for immediate investigation is the application of the FMC to imperative programming. Perhaps primary is the investigation of *local* state, probably via the inclusion of a *new* location construct. Given the multiple arrow types occurring in the semantics of the FMC given in Chapter 4, it might further be interesting to investigate a computational interpretation of the logic of Bunched Implications, alternate to the  $\alpha\lambda$ -calculus, where the linear and intuitionistic arrows correspond to linear and non-linear locations [78, 77]. Further, one might attempt to find links between the FMC and type systems for imperative programming such as Reynolds' Syntactic Control of Interference, or Separation Logic [100, 101, 103].

**Other:** One, perhaps obvious, direction is to include sum types, datatypes, and error handling, and it looks possible to capture all three in a uniform way. Another is to investigate alternate, perhaps more elegant, equivalent equational theories the FMC, and in particular to find one that serves as a well-behaved rewriting system. Extending the FMC to incorporate computational effects beyond reader/writer effects is of interest, perhaps especially aiming to be able to express *continuations*. One might consider extending the FMC in such a way as to capture the  $\lambda\mu$ -calculus [79]. Further, the use of *locations* in the FMC is reminiscent of *channels* in the  $\pi$ -calculus, except composition in the FMC is *sequential* rather than *parallel* [74, 75]. This could prove a promising avenue of investigation.

## 8.2 Summary

Recall that we aim to extend reasoning techniques familiar from the  $\lambda$ -calculus to real-world, effectful and imperative programming languages. We thus began by asking, how can we extend the  $\lambda$ -calculus to incorporate effects, without losing the good properties which make it a central object of study in the first place? The approach of the FMC towards modelling effects, while maintaining confluence, appears remarkable, yet raises many questions. One might worry that such an innovation would come at a cost. This thesis contributes doubly towards allaying these concerns: it is proved that two further fundamental properties of the  $\lambda$ -calculus (beyond confluence) – its categorical semantics, and the ability of its type system to guarantee strong normalisation – are preserved. The results serve as a sanity check, and it appears that the calculus is remarkably well-behaved. The results herein suggest that it has a type system and a denotational semantics as strong and natural (respectively) as the simply-typed  $\lambda$ -calculus itself.

The proofs contained in this thesis often have an operational flavour – especially that of the proof of strong normalisation. It would appear that the FMC naturally leads to clearer links between the operational and the semantic perspectives. The

work of a mathematician is not just to prove new theorems, but also to find the right definitions and proofs which simplify, bring new perspectives to, and new *understanding* to, existing ones. It is hoped that the reader is convinced that the novel definition of the FMC is indeed a *natural* extension of the  $\lambda$ -calculus, and that the reader may have found some new perspectives on old ideas.



# Bibliography

- [1] Samson Abramsky. What are the fundamental structures of concurrency? we still don't know! *CoRR*, abs/1401.4973, 2014.
- [2] Danel Ahman and Sam Staton. Normalization by evaluation and algebraic effects. *Electronic Notes in Theoretical Computer Science*, 298:51–69, 2013.
- [3] Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. *CoRR*, abs/2107.13433, 2021.
- [4] Robert Atkey. What is a categorical model of arrows? *Electronic Notes in Theoretical Computer Science*, 229(5):19–37, 2011.
- [5] Samuel Balco and Alexander Kurz. Nominal string diagrams. In Markus Roggenbach and Ana Sokolova, editors, *8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019, June 3-6, 2019, London, United Kingdom*, volume 139 of *LIPICs*, pages 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
- [6] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
- [7] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.
- [8] Chris Barrett and Alessio Guglielmi. A subatomic proof system for decision trees. *CoRR*, abs/2105.01382, 2021.
- [9] Chris Barrett, Willem Heijltjes, and Guy McCusker. The functional machine calculus. June 2022. 38th International Conference on Mathematical Foundations of Programming Semantics, MFPS 2022 ; Conference date: 11-07-2022 Through 13-07-2022.
- [10] Chris Barrett, Willem Heijltjes, and Guy McCusker. The Functional Machine Calculus II: Semantics. In Bartek Klin and Elaine Pimentel, editors,

- 31st EACSL Annual Conference on Computer Science Logic (CSL 2023), volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [11] Andrej Bauer. What is algebraic about algebraic effects and handlers? *CoRR*, abs/1807.05923, 2018.
  - [12] Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 420–431. IEEE, 1996.
  - [13] Filippo Bonchi, Joshua Holland, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Diagrammatic algebra: from linear to concurrent systems. volume 3 of *POPL*, pages 1–28. ACM New York, NY, USA, 2019.
  - [14] Paola Bruscoli, Alessio Guglielmi, Tom Gundersen, and Michel Parigot. Quasipolynomial normalisation in deep inference via atomic flows and threshold formulae. 2009.
  - [15] Antonio Bucciarelli, Delia Kesner, Alejandro Ríos, and Andrés Viso. The bang calculus revisited. In Keisuke Nakano and Konstantinos Sagonas, editors, *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*, volume 12073 of *Lecture Notes in Computer Science*, pages 13–32. Springer, 2020.
  - [16] Sivaramakrishnan Krishnamoorthy Chandrasekaran, Daan Leijen, Matija Pretnar, and Tom Schrijvers. Algebraic effect handlers go mainstream (dagstuhl seminar 18172). *Dagstuhl Reports*, 8(4):104–125, 2018.
  - [17] Eugenia Cheng. Iterated distributive laws. *Mathematical Proceedings of the Cambridge Philosophical Society*, 150:459 – 487, 2011.
  - [18] Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1941.
  - [19] Bob Coecke and Aleks Kissinger. Categorical quantum mechanics i: causal quantum processes. *Categories for the Working Philosopher*, pages 286–328, 2015.
  - [20] Ugo Dal Lago, Giulio Guerrieri, and Willem Heijltjes. Decomposing probabilistic lambda-calculi. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures*, volume 12077 of *LNCS*, pages 136–156, Cham, 2020. Springer International Publishing.
  - [21] Nicolaas Govert de Bruijn. Algorithmic definition of lambda-typed lambda calculus. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 131–146. Cambridge University Press, 1993.

- [22] Roel de Vrijer. Exactly estimating functionals and strong normalization. *Indagationes Mathematicae (Proceedings)*, 90(4):479–493, 1987.
- [23] Antonin Delpeuch and Jamie Vicary. Normalization for planar string diagrams and a quadratic equivalence algorithm. *Log. Methods Comput. Sci.*, 18(1), 2022.
- [24] Christopher Diggins. Simple type inference for higher-order stack-oriented languages. Technical Report Cat-TR-2008-001, 2008.
- [25] Rémi Douence and Pascal Fradet. A systematic study of functional language implementations. *ACM Transactions on Programming Languages and Systems*, 20(2):344–387, 1998.
- [26] Jean-Guillaume Dumas, Dominique Duval, and Jean-Claude Reynaud. Cartesian effect categories are freyd-categories. *CoRR*, abs/0903.3311, 2009.
- [27] Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. Enriching an effect calculus with linear types. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2009.
- [28] Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. The enriched effect calculus: syntax and semantics. *Journal of Logic and Computation*, 24(3):615–654, 2014.
- [29] Thomas Ehrhard and Giulio Guerrieri. The bang calculus: An untyped lambda-calculus generalizing call-by-name and call-by-value. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP’16)*, pages 174–187, 2016.
- [30] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.
- [31] Robin O. Gandy. Proofs of strong normalisation. In J.P Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press, 1980.
- [32] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [33] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.
- [34] Inge Gørtz, Signe Reuss, and Morten Sørensen. Strong normalization from weak normalization by translation into the Lambda-I-calculus. *Higher-Order and Symbolic Computation*, 16:253–285, 2003.

- [35] Alexandre Goy. Weakening and iterating laws using string diagrams. *CoRR*, abs/2205.03640, 2022.
- [36] Giulio Guerrieri and Giulio Manzonetto. The bang calculus and the two girard’s translations. In *Linearity-TLLA*, 2018.
- [37] Alessio Guglielmi, Tom Gundersen, and Michel Parigot. A proof calculus which reduces syntactic bureaucracy. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.
- [38] Tom Gundersen, Willem Heijltjes, and Michel Parigot. Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, jun 2013.
- [39] Masahito Hasegawa. Decomposing typed lambda-calculus into a couple of categorical programming languages. In *International Conference on Category Theory and Computer Science*, 1995.
- [40] Dominikus Herzberg and Tim Reichert. Concatenative programming: An overlooked paradigm in functional programming. In *4th International Conference on Software and Data Technologies (ICSOFT)*, 2009.
- [41] Chris Heunen and Bart Jacobs. Arrows, like monads, are monoids. In Stephen D. Brookes and Michael W. Mislove, editors, *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2006, Genova, Italy, May 23-27, 2006*, volume 158 of *Electronic Notes in Theoretical Computer Science*, pages 219–236. Elsevier, 2006.
- [42] Daniel Hirschhoff, Enguerrand Prebet, and Davide Sangiorgi. On the representation of references in the pi-calculus. In *31st International Conference on Concurrency Theory (CONCUR)*, LIPIcs, pages 34:1–34:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- [43] Gérard Huet. Residual theory in lambda-calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.
- [44] Dominic J.D. Hughes. Deep inference proof theory equals categorical proof theory minus coherence, 2004.
- [45] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.
- [46] Martin Hyland, Gordon D. Plotkin, and John Power. Combining computational effects: commutativity & sum. In Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro, editors, *Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP 17<sup>th</sup> World Computer*

- Congress - TC1 Stream / 2<sup>nd</sup> IFIP International Conference on Theoretical Computer Science (TCS 2002), August 25-30, 2002, Montréal, Québec, Canada*, volume 223 of *IFIP Conference Proceedings*, pages 474–484. Kluwer, 2002.
- [47] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electronic Notes in Theoretical Computer Science*, 172:437–458, 2007. Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.
  - [48] Bart Jacobs and Ichiro Hasuo. Freyd is kleisli, for arrows. In Conor McBride and Tarmo Uustalu, editors, *MSFP@MPC*, Workshops in Computing. BCS, 2006.
  - [49] Bart Jacobs, Chris Heunen, and Ichiro Hasuo. Categorical semantics for arrows. *J. Funct. Program.*, 19(3-4):403–438, 2009.
  - [50] Bart Jacobs, Aleks Kissinger, and Fabio Zanasi. Causal inference by string diagram surgery. International Conference on Foundations of Software Science and Computation Structures, pages 313–329. Springer, 2019.
  - [51] Alan Jeffrey. Premonoidal categories and flow graphs. *Electronic Notes in Theoretical Computer Science*, 10:51, 1998. HOOTS II, Second Workshop on Higher-Order Operational Techniques in Semantics.
  - [52] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
  - [53] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 71–84. ACM Press, 1993.
  - [54] André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in Mathematics*, 88(1):55–112, 1991.
  - [55] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013.
  - [56] Jan Willem Klop, Vincent van Oostrom, and Roel C. de Vrijer. Lambda calculus with patterns. *Theor. Comput. Sci.*, 398(1-3):16–31, 2008.
  - [57] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20:199–207, 2007.

- [58] Joachim Lambek. Deductive Systems and Categories II.
- [59] Joachim Lambek. Deductive Systems and Categories III.
- [60] Joachim Lambek. Cartesian closed categories and typed lambda- calculi. In Guy Cousineau, Pierre-Louis Curien, and Bernard J. Robinet, editors, *Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings*, volume 242 of *Lecture Notes in Computer Science*, pages 136–175. Springer, 1985.
- [61] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.
- [62] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
- [63] Peter J. Landin. Correspondence between Algol 60 and Church’s lambda-notation: Part I. *Communications of the ACM*, 8(2), 1965.
- [64] Tom Leinster. *Basic Category Theory*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2014.
- [65] Paul Levy. Adjunction models for call-by-push-value with stacks. January 2003. Conference on Category Theory and Computer Science, 9th ; Conference date: 15-08-2002 Through 17-08-2002.
- [66] Paul Blain Levy. *Call-by-push-value: A functional/imperative synthesis*, volume 2 of *Semantic Structures in Computation*. Springer Netherlands, 2003.
- [67] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19:377–414, 2006.
- [68] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185:182–210, 2003.
- [69] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. *Journal of Functional Programming*, 20(1):51–69, 2010.
- [70] Angel Robert Lynas and Bill Stoddart. Adding lambda expressions to Forth. In *22nd EuroForth Conference*, 2006.
- [71] Saunders Mac Lane. *Categories for the working mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1998.
- [72] John Maraist, Martin Odersky, David N Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. *Theoretical Computer Science*, 228:175–210, 1999.

- [73] Paul-André Melliès. Local states in string diagrams. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 2014.
- [74] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.
- [75] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile process, I. *Information and Computation*, 100:1–40, 1992.
- [76] Eugenio Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE, 1989.
- [77] Peter W. O’Hearn. Resource interpretations, bunched implications and the  $\alpha$  lambda-calculus. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA’99, L’Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 258–279. Springer, 1999.
- [78] Peter W. O’Hearn and David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999.
- [79] Michel Parigot.  $\lambda\mu$ -Calculus: an algorithmic interpretation of classical natural deduction. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 624 of *Lecture Notes in Computer Science (LNCS)*, pages 190–201, 1992.
- [80] Evan Patterson, David I. Spivak, and Dmitry Vagner. Wiring diagrams as normal forms for computing in symmetric monoidal categories. In David I. Spivak and Jamie Vicary, editors, *Proceedings of the 3rd Annual International Applied Category Theory Conference 2020, ACT 2020, Cambridge, USA, 6-10th July 2020*, volume 333 of *EPTCS*, pages 49–64, 2020.
- [81] Dusko Pavlovic. Monoidal computer i: Basic computability by string diagrams. *Information and Computation*, 226:94–116, 2013. Special Issue: Information Security as a Resource.
- [82] Slava Pestov, Daniel Ehrenberg, and Joe Groff. Factor: A dynamic stack-based programming language. *ACM SIGPLAN Notices*, 45(12):43–58, 2010.
- [83] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
- [84] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher order operational techniques in semantics*, pages 227–273. Isaac Newton Institute for Mathematical Sciences, 1998.

- [85] Gordon Plotkin and John Power. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 342–356. Springer, Berlin, Heidelberg, 2002.
- [86] Gordon Plotkin and John Power. Computational effects and operations: An overview. *Electronic Notes in Theoretical Computer Science*, 73:149–163, 2004. Proceedings of the Workshop on Domains VI.
- [87] Gordon D. Plotkin. Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.
- [88] Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.
- [89] Gordon D. Plotkin and John Power. Semantics for algebraic operations. In Stephen D. Brookes and Michael W. Mislove, editors, *Seventeenth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2001, Aarhus, Denmark, May 23-26, 2001*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 332–345. Elsevier, 2001.
- [90] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003.
- [91] Gordon D. Plotkin, John Power, Donald Sannella, and Robert D. Tennent. Lax logical relations. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 85–102. Springer, 2000.
- [92] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming (ESOP)*, pages 80–94. Springer, Berlin, Heidelberg, 2009.
- [93] Jaco van de Pol. Two different strong normalization proofs? In *Selected Papers from the Second International Workshop on Higher Order Algebra, Logic, and Term Rewriting (HOA '95)*, volume 1074 of *LNCS*, pages 201–220, 1995.
- [94] Jaco van de Pol and Helmut Schwichtenberg. Strict functionals for termination proofs. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications (TLCA '95)*, pages 350–364. Springer-Verlag, 1995.



- [95] A.J. Power and Hayo Thielecke. Closed Freyd- and  $\kappa$ -categories. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1644 of *LNCS*, pages 625–634. Springer, 1999.
- [96] J. Power. Models for the computational lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 40:288–301, 2000.
- [97] John Power. Premonoidal categories as categories with algebraic structure. *Theoretical Computer Science*, 278(1–2):303–321, May 2002.
- [98] John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7:453–468, 1997.
- [99] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22–25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 19–35. Elsevier, 2015.
- [100] John C. Reynolds. Syntactic control of interference. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 39–46. ACM Press, 1978.
- [101] John C. Reynolds. Syntactic control of inference, part 2. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11–15, 1989, Proceedings*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer, 1989.
- [102] John C. Reynolds. The discoveries of continuations. *LISP Symb. Comput.*, 6(3–4):233–248, 1993.
- [103] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.
- [104] Mario Román. Promonads and string diagrams for effectful categories. *CoRR*, abs/2205.07664, 2022.
- [105] Ralf Schweimeier. Categorical and graphical models of programming languages. 2001.
- [106] Helmut Schwichtenberg. Complexity of normalization in the pure typed lambda-calculus. In A.S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 453–457. Elsevier, 1982.

- [107] P. Selinger. *A Survey of Graphical Languages for Monoidal Categories*, pages 289–355. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [108] J.E.W. Smetsers, E. Barendsen, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Workshop Graph Transformations in Computer Science*, Lecture Notes in Computer Science, Berlin, January 4–8 1993. Schloss Dagstuhl, Springer-Verlag.
- [109] Bill Stoddart and Peter J. Knaggs. Type inference in stack based languages. *Formal Aspects of Computing*, 5:289–298, 1993.
- [110] Th. Streicher and B. Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.
- [111] W.W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.
- [112] Masako Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 118(1):120–127, 1995.
- [113] Alwen Tiu. A local system for intuitionistic logic. In *Logic for programming, artificial intelligence, and reasoning*, volume 4246 of *Lecture Notes in Comput. Sci.*, pages 242–256. Springer, Berlin, 2006.
- [114] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. In Jiri Adámek and Clemens Kupke, editors, *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science, CMCS 2008, Budapest, Hungary, April 4-6, 2008*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 263–284. Elsevier, 2008.
- [115] Matthijs Vákár. A framework for dependent types and effects. *CoRR*, abs/1512.08009, 2015.
- [116] Manfred von Thun. Joy: Forth’s functional cousin. In *Proc. 17th EuroForth Conference*, 2001.
- [117] Philip Wadler. Comprehending monads. In Gilles Kahn, editor, *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, pages 61–78. ACM, 1990.
- [118] Philip Wadler. The essence of functional programming. In Ravi Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14. ACM Press, 1992.
- [119] Glynn Winskel. *The formal semantics of programming languages: An introduction*. MIT Press, Cambridge, Massachusetts, 1993.