The Functional Machine Calculus

Willem Heijltjes
Talk proposal — HOPE'21

The Functional Machine Calculus (FMC) is a new approach to combining the λ-calculus with computational effects. This is an important problem in theoretical computer science, as its solutions may form the basis of typed, higher-order programming languages, with perhaps as ultimate aim the unification of the imperative and functional programming paradigms. It has a rich history, including Plotkin's seminal study of call–by–name and call–by–value λ-calculus, Moggi's solution using monads, now at home in the Haskell programming language and a staple of programming theory, the semantic characterization of algebraic effects by Plotkin, Power, and others in Lawvere theories and premonoidal categories, the related arrow calculus by Hughes, Levy's call–by–push–value paradigm and its later refinements in Ehrhard and Guerrieri's bang-calculus and Egger, Møgelberg, and Simpson's enriched effect calculus, and recently Plotkin and Pretnar's effect handlers.

The solution presented by the FMC consists of two natural modifications to the λ-calculus, *locations* and *sequencing*.

Locations The abstraction and application constructs of the λ-calculus are parameterized in a set of *locations*. This allows a natural encoding of the algebraic effects of input, output, and mutable higher-order store, where a location may represent an input stream, an output stream, or a storage cell, and operations for effects are encoded as combinations of abstractions (as consumers) and applications (as producers) for the corresponding location. Probabilities and non-determinism are included as special cases of input. Effects are evaluated by standard β-reduction, but via their algebraic laws, not their operational semantics. The usual problem of non-confluence is then avoided, and β-reduction remains confluent without imposing a strategy. Traditional simple types are parameterized in the same set of locations, to give a new type system for effects.

Sequencing The variable construct of the λ-calculus is separated into a *variable-with-continuation* and an *end-of-instructions* construct. Conceptually, where a variable in the λ-calculus represents a *value*, a variable in the FMC represents a *command* or a *computation*, after termination of which the program continues with the next sequence of instructions. The modification gives a natural encoding of imperative *sequencing* and *skip*, and gives control over execution, allowing to express *lazy* and *eager* evaluation of computation and of effects. It is sufficient to encode Moggi's computational metalanguage and Levy's call–by–push–value. As a programming language, the FMC is akin to Haskell's *do*-notation, generalized to allow any number of return values (as opposed to

exactly one) and with a new type system that gives information about how effects are used.

A second perspective on the FMC is as an instruction language for a stack machine, a common simplification of the Krivine Abstract Machine omitting environments. An application pushes its argument to the stack, and an abstraction pops the top element off the stack and substitutes it for its bound variables. *Locations* are represented in the machine by multiple stacks (or streams), one for each location, which may naturally model input streams, output streams, and storage cells. *Sequencing* gives concatenation of instruction sequences and the empty instruction sequence. In this way, the machine gives an operational semantics for the calculus. Typing gives an abstract account of the net consumption and production behaviour of the machine, where a type derivation is a direct proof of termination of the machine.

Attached is a preliminary draft detailing the current state of development.

The Functional Machine Calculus

Willem Heijltjes

(Draft, May 2021)

Abstract

This paper presents the Functional Machine Calculus (FMC) as a simple model of higher-order computation with effects, including mutable store, input and output, and probabilistic and non-deterministic computation as special cases of input. The FMC is derived from the lambda-calculus via two independent generalizations. One enables the encoding of effects into the calculus; the other provides control when they are executed, and encodes the imperative features of "sequencing" and "skip".

These generalizations are particularly natural from the perspective of the FMC as an instruction language for a simple stack machine (a simplified Krivine Abstract Machine, and the "M" in "FMC"). The first generalization corresponds to allowing multiple stacks, which may represent storage cells and input and output streams to capture the operational semantics of effects. The second generalization gives composition of instruction sequences, and the empty sequence as a unit.

The FMC naturally encodes various previous approaches, including Plotkin's Call-By-Value lambda-calculus, Moggi's Computational Metalanguage, and Levy's Call-By-Push-Value. The calculus is confluent, which is possible because beta-reduction encodes the evaluation of effects via their algebraic equations, not their operational semantics. Different evaluation strategies in the lambda-calculus are captured by different encodings into the FMC.

The FMC can be simply typed, and a simple, direct proof shows that types confer termination of the stack machine, or equivalently of weak head reduction. Types give a natural solution to the problem of types for higher-order store.

1 Introduction

The λ-calculus [2, 1] is one of our most effective models of computation, due to a combination of features that includes in particular: a compact, intuitive syntax; confluent reduction; powerful typing disciplines that confer strong normalization and other properties; and its amenability to formal reasoning, i.e., semantics. On its own, however, the λ-calculus models only pure, isolated calculation. Real-world computation is interactive and involves *computational*

effects: input and output, mutable store, non-determinism and probabilities, exceptions, continuations, concurrency, and more. In the presence of effects, the good properties of the λ-calculus are not naturally preserved, and to recover them is an important challenge.

When primitive operations for effects are introduced naïvely, confluence fails. The distinction between reduction strategies becomes significant, and since *eager* or *lazy* evaluation yield different results, an implementation needs to offer both, to give explicit control over when effects are called. Common in practice (e.g. in Scheme, ML, Scala) is to use *call–by–value* as primary strategy, supplemented with constructions to delay evaluation, such as closures, coroutines, thunks, or laziness annotations. The semantic study of effects has yielded several alternatives: Moggi's celebrated account of effects as *monads* [10], taken up in Haskell; Levy's *call-by-push-value* paradigm [8], which interprets both *call–by–name* and *call–by–value* so that effects can safely be introduced; related solutions based in linear logic such as the *enriched effect calculus* by Egger, Møgelberg, and Simpson [4] and the *bang-calculus* by Ehrhard and Guerrieri [5]; and Plotkin and Pretnar's highly versatile *effect handlers* [13]. What these solutions have in common, however, is that they introduce a significant amount of structure, in the form of primitive operations for effects and constructions to control evaluation behaviour, contrary to the original simplicity of the λ-calculus.

Here, I propose a new solution, the *Functional Machine Calculus* (FMC), that stands out for its radical simplicity. The highlights are as follows.

Syntax The FMC generalizes the λ-calculus via two natural modifications: one, to parametrize application and abstraction in a set of *locations*; and two, to include a natural notion of *sequencing* by generalizing the variable construct. The calculus remains as simple and compact as the λ-calculus itself.

Effects The computational effects of mutable higher-order store, input, and output, are encoded in the parameterized application and abstraction constructs, with each effect assigned a separate location. Non-deterministic and probabilistic computation are included as special cases of input.

Abstract machine The FMC is an instruction language for an abstract machine in the style of Krivine [7], the "M" in FMC, which provides its operational semantics. The first modification, *locations*, is embodied in the machine by multiple stacks (or streams), which represent input streams, output streams, and memory cells (as stacks of depth at most one). The second, *sequencing*, naturally gives sequencing of machine instructions.

Confluence Reduction is by a modified β-rule. This remains confluent, which is possible because it evaluates effects via their algebraic equations (as described by Plotkin and Power [11]), and not via their operational semantics.

Strategies Because the calculus is confluent, to give different (eager or lazy) evaluation behaviour requires giving different terms. Different evaluation regimes for the λ-calculus thus require different interpretations into the

FMC; the λ-calculus is embedded as a fragment, while Plotkin's call–by–value λ-calculus [12] is interpreted.

Related calculi For the given effects, the FMC interprets call–by–push–value, the bang-calculus, and the computational metalanguage.

Types The FMC can be simply typed, where typeability is direct proof of termination of the abstract machine.

The FMC originates in an attempt to restore confluence with effects, following recent work that does so for probabilistic λ-calculi by Dal Lago, Guerrieri, and the author [3].

2 Encoding effects

From the perspective of the Krivine Abstract Machine (KAM), the λ-calculus is an instruction language for an abstract machine with a single stack. An application $N\,M$ *pushes* its argument M onto the stack and continues with the function N. An abstraction $\lambda x.\, N$ *pops* the first element M off the stack and binds it to x in N. Considering effects in the context of such a stack machine, a key observation is that various effects can be modelled through push and pop actions:

- Reading from input is a pop action from a stream.
- Writing to output is a push action to a stream.
- Updating a storage cell is a pop action, discarding the old value, followed by a push action, instating the new value.
- Reading from a storage cell is a pop action, retaining the popped value for use, followed by a push action of the same value, to reinstate the cell.

Where these operations differ from each other, and from regular abstraction and application, is the *source* and *target* for their pop and push actions. To capture these effects it is then sufficient to *parameterize* abstraction and application in a set of global *locations*, to represent input and output streams, storage cells, and the original application stack of the machine. The stack machine, for its part, will feature a separate stack, or stream, for each location. Stacks representing storage cells will then be bounded to a depth of one, for input and output the machine will have streams, and by considering further input streams as probabilistically or non-deterministically generated such operations may also be modelled.

The *poly-λ-calculus* implements this idea, below, with application and abstraction parameterized in a set $\mathcal{A}$ of *locations*, ranged over by $a, b, c, \ldots$. The notation adjusts that of the λ-calculus to add locations and to emphasize the stack machine interpretation. That is, an abstraction $\lambda x.\, N$ is written as $\langle x \rangle.\, N$,

and parameterized in a location a as $a\langle x\rangle.\, N$. Similarly, an application NM becomes $[M].\, N$, with the argument M first to emphasize that it pushes M and continues as N, and is parameterized in a as $[M]a.\, N$. For reduction, a redex is formed only by an application and abstraction on the same location, as $[M]a.\, a\langle x\rangle.\, N$, but more generally these may be separated by applications and abstractions on other locations.

Definition 1. The *poly-λ-calculus* $P\Lambda$ is given by

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

with from left to right: a *variable*, an *application* or *push action on* a where N is the function and M is the argument, and an *abstraction* or *pop action on* a where x becomes bound in N. Reduction is by the rule

$$[M]a.\, A_1 \dots A_n.\, a\langle x\rangle.\, N \ \longrightarrow \ A_1 \dots A_n.\, \{M/x\}N$$

where each A_i is an action *not* along a, and $\{M/x\}N$ is a capture-avoiding substitution of M for x in N.

The regular λ-calculus is embedded via a reserved location λ, which may be omitted for brevity.

$$\lambda x.\, N \triangleq \lambda\langle x\rangle.\, N \ = \ \langle x\rangle.\, N$$
$$N\, M \triangleq [M]\lambda.\, N = [M].\, N$$

We will use the poly-λ-calculus to explore the encoding of effect operators. Glancing ahead, since it is confluent, and since the operational semantics for the effect operators will be given by a poly-stack Krivine Abstract Machine, it forces a strict call–by–name semantics on them. The second modification, *sequencing*, will allow the encoding of call–by–value evaluation, and will give the Functional Machine Calculus.

First, we consider the abstract machine. To represent it in a simple, uniform format, transitions are given by a horizontal rule, and initial and final states as transitions without source, respectively without target, so that a run can be pictured as a column of states. Since it's purpose is mainly to give an operational semantics, and not an implementation, it is simplified from the KAM to avoid environments, instead using substitutions directly.

Definition 2. The *poly-stack abstract machine* (PAM) has *states* (S, N) where N is a poly-λ-term and $S : \mathcal{A} \to P\Lambda^{\mathbb{N}}$ is the *memory*, a function assigning to each location $a \in \mathcal{A}$ a stack or stream of poly-λ-terms $S_a \in P\Lambda^{\mathbb{N}}$. Empty stacks are given as ε_a, a stack with top element M and remaining stack S_a is given as $S_a \cdot M$, and the stack S_a is separated from the remaining memory S as $S; S_a$. Initial states, final states, and transitions are as follows, and a double line will indicate multiple steps.

$$\frac{}{(\ \varepsilon\ ,\ N\ )} \qquad \frac{(\ S\ ,\ x\ )}{} \qquad \frac{(\ S\ ;\ \varepsilon_a\ ,\ a\langle x\rangle.\, N\ )}{}$$

$$\frac{(\ S\ ;\ S_a\ ,\ [M]a.\,N\ )}{(\ S\ ;\ S_a\cdot M\ ,\ N\ )} \qquad \frac{(\ S\ ;\ S_a\cdot M\ ,\ a\langle x\rangle.\,N\ )}{(\ S\ ;\ S_a\ ,\ \{M/x\}N\ )}$$

The abstract machine further illustrates why in the reduction rule, the application and abstraction may be separated by actions on other locations: in a run, these affect other, independent stacks only, and the redex matches a consecutive push and pop *on the same stack*. Actions on different stacks may then also permute without affecting the overall computation, provided that binding of variables is preserved. This suggests a natural equivalence $\sim$ on terms, defined below, where $a \neq b$.

$$\begin{aligned} [M]a.\,[N]b.\,P &\sim [N]b.\,[M]a.\,P \\ a\langle x\rangle.\,[N]b.\,P &\sim [N]b.\,a\langle x\rangle.\,P \qquad \text{if } x \notin \mathsf{fv}(N) \\ a\langle x\rangle.\,b\langle y\rangle.\,P &\sim b\langle y\rangle.\,a\langle x\rangle.\,P \end{aligned}$$

By these permutations the actions separating a redex may be moved up past the application,

$$[M]a.\,A_1\ldots A_n.\,a\langle x\rangle.\,N \;\sim\; A_1\ldots A_n.\,[M]a.\,a\langle x\rangle.\,N$$

since on the left no A_i binds in M (though to prevent capture, some variables may need to be renamed. Then if terms are considered modulo $\sim$ the reduction step becomes (almost) a standard β-reduction:

$$[M]a.\,a\langle x\rangle.\,N \;\rightarrowtail\; \{M/x\}N$$

However, with the current definition of β-reduction, which does allow a "split" redex, the equivalence $\sim$ is not necessary for reduction. We now consider encoding effects into the poly-λ-calculus.

Input: Reading from input is given by a pop action $\mathsf{in}\langle x\rangle.\,N$ on a dedicated *input* location $\mathsf{in} \in \mathcal{A}$. To simulate an input stream of terms $I_1, I_2, I_3, \ldots$ a term N is evaluated in the context of a stream of *push* actions $[I_i]\mathsf{in}$,

$$\ldots[I_3]\mathsf{in}.\,[I_2]\mathsf{in}.\,[I_1]\mathsf{in}.\,N$$

so that reduction binds the first input term I_1 to the variable x of the first input operation $\mathsf{in}\langle x\rangle$ in the term:

$$\begin{aligned} \ldots[I_3]\mathsf{in}.\,[I_2]\mathsf{in}.\,[I_i]\mathsf{in}.\,A_1\ldots A_n.\,\mathsf{in}\langle x\rangle.\,N &\rightarrowtail \\ \ldots[I_3]\mathsf{in}.\,[I_2]\mathsf{in}.\,A_1\ldots A_n.\,\{I_1/x\}N & \end{aligned}$$

where each A_i is an action not on in. Similarly, the stack machine is initialized with the stream $S_{\mathsf{in}} = \cdots I_3\cdot I_2\cdot I_1$ to give an operational reading via the following transition.

$$\frac{(\ S\ ;\ \cdots I_3\cdot I_2\cdot I_1\ ,\ \mathsf{in}\langle x\rangle.\,N\ )}{(\ S\ ;\ \cdots I_3\cdot I_2\ ,\ \{I_1/x\}N\ )}$$

Output: Writing a term M to output is by a push action $[M]\mathsf{out}.\, N$ on a dedicated *output* location $\mathsf{out} \in \mathcal{A}$. There is no corresponding pop action; instead, that a term N reduces to M while outputting a series of terms $O_1, O_2, \ldots, O_n$ is given by a reduction leaving an initial sequence of push actions:

$$N \twoheadrightarrow [O_1]\mathsf{out}.\, [O_2]\mathsf{out} \ldots [O_n]\mathsf{out}.\, M\ .$$

The operational reading is illustrated for the ith output in a reduction, by a machine transition

$$\frac{(\ S\ ;\ \varepsilon_{\mathsf{out}} \cdot O_1 \cdots O_{i-1}\ ,\ [O_i]\mathsf{out}.\, N\ )}{(\ S\ ;\ \varepsilon_{\mathsf{out}} \cdot O_1 \cdots O_{i-1} \cdot O_i\ ,\ N\ )}\ .$$

Note the peculiarity that output should be considered a *queue*, not a *stack*: the first output will be at the *bottom* of the stack, but would be at the *head* of a queue. We will come back to this in the Conclusion, Section 9.

Higher-order mutable store: A subset $\mathcal{C} \subseteq \mathcal{A}$ of locations is designated as storage cells, whose associated stacks are expected to hold at most one value. The standard operations *update* $c := M\ ;\ N$, which updates the cell c with value M and then continues as N, and *read* $!c$, which reads the value from the cell c and executes it, are encoded as follows,

$$\begin{aligned} c := M\ ;\ N &\triangleq c\langle\_\rangle.\, [M]c.\, N \\ !c &\triangleq c\langle x\rangle.\, [x]c.\, x \end{aligned}$$

where $(\_)$ represents a variable that does not occur in M or N. Following the earlier informal description, the update $c := M$ is encoded by a popping and discarding the value held by the cell c, and then pushing the new value M. The read operation $!c$ is encoded by first popping the value of the cell c and binding it to the local variable x, then restoring the value to c with a push action, and finally executing it. The abstract machine gives the formal operational perspective:

$$c := M\ ;\ N: \quad \frac{\begin{array}{r}(\ S\ ;\ \varepsilon_c \cdot P\ ,\ c\langle\_\rangle.\, [M]c.\, N\ ) \\ (\ S\ ;\ \varepsilon_c\ ,\ [M]c.\, N\ )\end{array}}{(\ S\ ;\ \varepsilon_c \cdot M\ ,\ N\ )}$$

$$!c: \quad \frac{\begin{array}{r}(\ S\ ;\ \varepsilon_c \cdot P\ ,\ c\langle x\rangle.\, [x]c.\, x\ ) \\ (\ S\ ;\ \varepsilon_c\ ,\ [P]c.\, P\ )\end{array}}{(\ S\ ;\ \varepsilon_c \cdot P\ ,\ P\ )}$$

Since the machine does not reduce under abstractions, and both encoded operators start with an abstraction, the stack for each cell location $c \in \mathcal{C}$ must be initialized with a (dummy) value. For reduction, this is not necessary, except to obtain a tight correspondence with the machine. Then if $\mathcal{C} = \{c_1, \ldots, c_n\}$, a term N should be evaluated in the context of a push action of a dummy value $\star$ for each c_i:

$$[\star]c_1.\ \ldots [\star]c_n.\, N\ .$$

Observe the interesting situation that the *machine* implements what is normally the *reduction* behaviour of the update and read operators. Reduction in the calculus implements two familiar algebraic laws,

$$\begin{aligned} c := N \,;\, c := M \,;\, P &= c := M \,;\, P \\ c := M \,;\, !c &= c := M \,;\, M \,. \end{aligned}$$

which are central to the algebraic characterization of state, and by extension to the idea of *algebraic effects* [11]. The reductions are as follows, with reduced redexes underlined.

$$\begin{aligned} c := N \,;\, c := M \,;\, P &= c\langle\_\rangle.\, \underline{[N]c.\, c\langle\_\rangle}.\, [M]c.\, P \\ &\to c\langle\_\rangle.\, [M]c.\, P \\ &= c := M \,;\, P \end{aligned}$$

$$\begin{aligned} c := M \,;\, !c &= c\langle\_\rangle.\, \underline{[M]c.\, c\langle x\rangle}.\, [x]c.\, x \\ &\to c\langle\_\rangle.\, [M]c.\, M \\ &= c := M \,;\, M \end{aligned}$$

Non-deterministic and probabilistic computation: This account follows [3], where more details can be found. Non-deterministic and probabilistic choice can be modelled along the lines of input, with two dedicated locations $\mathsf{nd}, \mathsf{rnd} \in \mathcal{A}$. The associated pop actions $\mathsf{nd}\langle x\rangle.\, N$ and $\mathsf{rnd}\langle x\rangle.\, N$ expect a Boolean value $T = \lambda x.\, \lambda y.\, x$ or $F = \lambda x.\, \lambda y.\, y$ for x, in Church encoding as we have not introduced primitives, which can be used inside N to project a subterm xMP onto one value M or another P. The machine is then initialized with input streams of Booleans B_i, one non-deterministically generated for nd and one probabilistically generated for rnd. Reduction similarly takes place in a corresponding context of push actions:

$$\begin{aligned} &\ldots [B_3]\mathsf{nd}.\, [B_2]\mathsf{nd}.\, [B_1]\mathsf{nd}.\, N \\ &\ldots [B_3]\mathsf{rnd}.\, [B_2]\mathsf{rnd}.\, [B_1]\mathsf{rnd}.\, N \quad . \end{aligned}$$

The algebraic properties that make the choice for each B_i as T or F into a probabilistic or non-deterministic one are then encoded in how these context streams are generated. For instance, a non-deterministic or fair probablistic generarator can be simulated via the following rules (which would apply at top-level only, and not in context).

$$\begin{aligned} N &\to_{\mathsf{nd}} [T]\mathsf{nd}.\, N + [F]\mathsf{nd}.\, N \\ N &\to_{\mathsf{rnd}} [T]\mathsf{rnd}.\, N \oplus_{\frac{1}{2}} [F]\mathsf{rnd}.\, N \end{aligned}$$

A traditional non-deterministic sum $+$ and fair probabilistic sum $\oplus$, internal to the calculus, can then be encoded by

$$\begin{aligned} N + M &\triangleq \mathsf{nd}\langle x\rangle.\, x\, M\, N \\ N \oplus M &\triangleq \mathsf{rnd}\langle x\rangle.\, x\, M\, N \,. \end{aligned}$$

$(\ \varepsilon_a \cdot \star\ ;\ \varepsilon_\lambda$	,	$a\langle\_\rangle.\,[2]a.\,[a\langle\_\rangle.\,[3]a.\,\langle x\rangle.\,x].\,\langle f\rangle.\,[a\langle y\rangle.\,[y]a.\,y].\,f\ )$
$(\ \varepsilon_a \cdot 2\ ;\ \varepsilon_\lambda$	,	$[a\langle\_\rangle.\,[3]a.\,\langle x\rangle.\,x].\,\langle f\rangle.\,[a\langle y\rangle.\,[y]a.\,y].\,f\ )$
$(\ \varepsilon_a \cdot 2\ ;\ \varepsilon_\lambda \cdot (a\langle\_\rangle.\,[3]a.\,\langle x\rangle.\,x)$	,	$\langle f\rangle.\,[a\langle y\rangle.\,[y]a.\,y].\,f\ )$
$(\ \varepsilon_a \cdot 2\ ;\ \varepsilon_\lambda$	,	$[a\langle y\rangle.\,[y]a.\,y].\,a\langle\_\rangle.\,[3]a.\,\langle x\rangle.\,x\ )$
$(\ \varepsilon_a \cdot 2\ ;\ \varepsilon_\lambda \cdot (a\langle y\rangle.\,[y]a.\,y)$	,	$a\langle\_\rangle.\,[3]a.\,\langle x\rangle.\,x\ )$
$(\ \varepsilon_a \cdot 3\ ;\ \varepsilon_\lambda \cdot (a\langle y\rangle.\,[y]a.\,y)$	,	$\langle x\rangle.\,x\ )$
$(\ \varepsilon_a \cdot 3\ ;\ \varepsilon_\lambda$	,	$a\langle y\rangle.\,[y]a.\,y\ )$
$(\ \varepsilon_a \cdot 3\ ;\ \varepsilon_\lambda$	,	$3\ )$

Figure 1: An example run of the Poly-stack Abstract Machine

Example 3. *Consider the following example. Numbers can be taken either as primitives or as Church numerals, and the colour-coding helps visualize the encoding of operations into the poly-λ-calculus and their reduction.*

$$a := 2\,;(\lambda f.\,f\,!a)\,(a := 3\,;\lambda x.\,x)$$
$$=$$
$$a\langle\_\rangle.\,[2]a.\,[a\langle\_\rangle.\,[3]a.\,\langle x\rangle.\,x].\,\langle f\rangle.\,[a\langle y\rangle.\,[y]a.\,y].\,f$$

The example reduces as follows. The matching application and abstractions of reduced redexes are underlined, which may be separated by actions on other locations.

$$\begin{aligned}
& a\langle_\rangle.\,[2]a.\,\underline{[a\langle_\rangle.\,[3]a.\,\langle x\rangle.\,x].\,\langle f\rangle}.\,[a\langle y\rangle.\,[y]a.\,y].\,f \\
\longrightarrow\ & a\langle_\rangle.\,[2]a.\,\underline{[a\langle y\rangle.\,[y]a.\,y]}.\,a\langle_\rangle.\,[3]a.\,\underline{\langle x\rangle}.\,x \\
\longrightarrow\ & a\langle_\rangle.\,\underline{[2]a}.\,a\langle_\rangle.\,[3]a.\,a\langle y\rangle.\,[y]a.\,y \\
\longrightarrow\ & a\langle_\rangle.\,\underline{[3]a.\,a\langle y\rangle}.\,[y]a.\,y \\
\longrightarrow\ & a\langle_\rangle.\,[3]a.\,3 \\
=\ & a := 3\,;3
\end{aligned}$$

The corresponding run of the machine is given in Figure 1, with a stack λ for regular abstraction and application, and a stack for the cell a initialized with $\star$.

3 The functional machine calculus

The poly-λ-calculus forces a call–by–name interpretation on effect operators. For the calculus to be practically useful, it will also need to allow call–by–value behaviour. We will explore what is needed through another example.

Example 4. *Naïve effectful reduction gives different results for call–by–name and call–by–value, as shown by the following example.*

$$a := 2\,;\,(\lambda x.\,!a)\;(a := 3\,;\,5) \quad \begin{matrix} \twoheadrightarrow_{\mathsf{cbn}} 2 \\ \twoheadrightarrow_{\mathsf{cbv}} 3 \end{matrix}$$

With cbv*, the argument $a := 3\,;5$ is evaluated and updates the cell a with the value 3 before $!a$ is called; with* cbn*, the argument is deleted without evaluation, and the value stored at a remains 2 when $!a$ is called. The encoding in the poly-λ-calculus follows the* cbn*-reduction:*

$$\begin{aligned} & a\langle\_\rangle.\,[2]a.\,\underline{[a\langle\_\rangle.\,[3]a.\,5].\,\langle x\rangle}.\,a\langle y\rangle.\,[y]a.\,y \\ \longrightarrow\; & a\langle\_\rangle.\,\underline{[2]a.\,a\langle y\rangle}.\,[y]a.\,y \\ \longrightarrow\; & a\langle\_\rangle.\,[2]a.\,2 \\ =\; & a := 2\,;2 \end{aligned}$$

Building on this example, a key observation is that its call–by–value behaviour is captured by a slightly different term:

$$\begin{aligned} & a\langle\_\rangle.\,\underline{[2]a.\,a\langle\_\rangle.\,[3]a.\,[5].\,\langle x\rangle}.\,a\langle y\rangle.\,[y]a.\,y \\ \longrightarrow\; & a\langle\_\rangle.\,[3]a.\,\underline{[5].\,\langle x\rangle}.\,a\langle y\rangle.\,[y]a.\,y \\ \longrightarrow\; & a\langle\_\rangle.\,\underline{[3]a.\,a\langle y\rangle}.\,[y]a.\,y \\ \longrightarrow\; & a\langle\_\rangle.\,[3]a.\,3 \\ =\; & a := 3\,;3 \end{aligned}$$

Since the aim is confluence, the interpretation of cbn and cbv would have to be through different translations. However, while the above is a perfectly valid term of the poly-λ-calculus, it is unavailable as a translation from the term in Example 4. That is because the subterm $a := 3\,;5$ of Example 4 corresponds in the desired cbv-term to a fragment $a\langle\_\rangle.\,[3]a.\,[5]$, as underlined below, that is *not a subterm*, since it is *not a poly-λ-term*.

$$a\langle\_\rangle.\,[2]a.\,\underline{a\langle\_\rangle.\,[3]a.\,[5]}.\,\langle x\rangle.\,a\langle y\rangle.\,[y]a.\,y$$

To see what modifications are required to make $a\langle\_\rangle.\,[3]a.\,[5]$ into a subterm, consider replacing it with a free variable z, for which $a\langle\_\rangle.\,[3]a.\,[5]$ may then be substituted again:

$$a\langle\_\rangle.\,[2]a.\,\underline{z}.\,\langle x\rangle.\,a\langle y\rangle.\,[y]a.\,y\;.$$

The change needed is the following. In the λ-calculus and poly-λ-calculus, a term is a sequence of abstractions and applications, ending in a variable; this is called the *spine* of the term. To allow the above term, variables need to occur *within* the sequence as well. To adapt the syntax, the variable construct x is decomposed into a *variable–with–continuation* $x.\,N$ and an *end–of–instructions*

construct $\star$, so that the original variable constructor is recovered as $x.\star$. This will allow the above construction, and a translation along the desired lines, as there will now be the following terms.

$$a\langle\_\rangle.\,[2]a.\,z.\,\langle x\rangle.\,a\langle y\rangle.\,[y]a.\,y.\star \qquad a\langle\_\rangle.\,[3]a.\,[5].\star$$

This decomposition of the variable, as a modification to a λ-calculus, will be called *sequentiality*. Applied to the poly-λ-calculus, it gives the Functional Machine Calculus.

Definition 5. The *Functional Machine Calculus* (FMC) is given by the grammar

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

with from left to right an *end* or *nil*, a *(sequential) variable*, an *application* or *push action* on the location a, and an *abstraction* or *pop action* on the location a which binds x in N. Reduction is by the β-rewrite rule

$$[M]a.\,A_1 \ldots A_n.\,a\langle x\rangle.\,N \;\longrightarrow\; A_1 \ldots A_n.\,\{M/x\}N$$

where each A_i is an action *not* on a and $\{M/x\}N$ is a *capture-avoiding substitution* of M for x in N, defined by

$$\begin{aligned}
\{L/y\}\star &= \star \\
\{L/y\}\,y.\,N &= L.\,\{L/y\}N \\
\{L/y\}\,x.\,N &= x.\,\{L/y\}N \\
\{L/y\}\,[M]a.\,N &= [\{L/y\}M]a.\,\{L/y\}N \\
\{L/y\}\,a\langle y\rangle.\,N &= a\langle y\rangle.\,N \\
\{L/y\}\,a\langle x\rangle.\,N &= a\langle z\rangle.\,\{L/y\}\{z/x\}N \quad (z \text{ is fresh})
\end{aligned}$$

where *capture-avoiding composition* $M.\,N$ is defined by

$$\begin{aligned}
\star\,.\,N &= N \\
(x.\,M).\,N &= x.\,(M.\,N) \\
([L]a.\,M).\,N &= [L]a.\,(M.\,N) \\
(a\langle x\rangle.\,M).\,N &= a\langle z\rangle.\,((\{z/x\}M).\,N) \quad (z \text{ is fresh})\,.
\end{aligned}$$

Terms are considered modulo α-equivalence. Usually the trailing $.\star$ of a term will be omitted, so that x is shorthand for $x.\star$, $[M]a$ for $[M]a.\star$, and $a\langle x\rangle$ for $a\langle x\rangle.\star$. The equivalence that permutes actions on different locations, $(\sim)$, translates directly to the FMC.

Definition 6. The *Functional Abstract Machine* (FAM) has *states* (S, N) where N is a FMC-term and $S : \mathcal{A} \to \mathrm{FMC}^{\mathbb{N}}$ is the *memory* function assigning to each location $a \in \mathcal{A}$ a stack or stream of FMC-terms $S_a \in \mathrm{FMC}^{\mathbb{N}}$. Initial states, final states, and transitions are as follows.

$$\frac{}{(\,\varepsilon\,,\,N\,)} \qquad \frac{(\,S\,,\,\star\,)}{} \qquad \frac{(\,S\,,\,x.\,N\,)}{} \qquad \frac{(\,S\,;\,\varepsilon_a\,,\,a\langle x\rangle.\,N\,)}{}$$

$$\frac{(\,S\,;\,S_a\,,\,[M]a.\,N\,)}{(\,S\,;\,S_a\cdot M\,,\,N\,)} \qquad \frac{(\,S\,;\,S_a\cdot M\,,\,a\langle x\rangle.\,N\,)}{(\,S\,;\,S_a\,,\,\{M/x\}N\,)}$$

Programming in the FMC Separating the constructs $\star$ and $x.\,N$ introduces *sequencing* or *composition* $M.\,N$, with $\star$ as the unit. This allows to naturaly express the difference between lazy and eager evaluation: $[M].\,N$ passes the term M along to N without executing it, as a thunk, while $M.\,N$ executes first M and then N, passing on the result of M to N. The result of M must be pushed onto a stack, and popped by N; it is expected that M is of the form $M'.\,[R]$, and N of the form $\langle x\rangle.\,N'$, to pick up the result R as x.

To work in this paradigm, natural operations for state (for a memory cell c), output, input, and random choice are as follows.

$$\begin{aligned}
\mathsf{get}\ c &= c\langle x\rangle.\,[x]c.\,[x]\\
\mathsf{set}\ c &= \langle x\rangle.\,c\langle_\rangle.\,[x]a\\
\mathsf{print} &= \langle x\rangle.\,[x]\mathsf{out}\\
\mathsf{read} &= \mathsf{in}\langle x\rangle.\,[x]\\
\mathsf{rand} &= \mathsf{rnd}\langle x\rangle.\,[x]
\end{aligned}$$

For a *cell* location c, the term $\mathsf{get}\ c$ retrieves the value stored at c and pushes it onto the main stack. The term $\mathsf{set}\ c$ pops the last result from the main stack, and updates c with that value. The term rand transports a random value from rnd onto the main stack, and print takes the last result to the output stream.

Primitives such as Booleans and integers, follow a similar pattern, operating as a standard stack calculus. Values are $\top$, $\bot$, 0, 1, -1, 2, -2, $\ldots$, and the primitive operations of addition $+$, multiplication $\times$, and conditional if pop the required number of items from the main stack, and reinstate their result. Reduction implements this behaviour as follows. For brevity, any actions on other locations are omitted, which the full reduction rule would allow between the operator and any inputs. (That is, reduction is given modulo $\sim$.)

$$\begin{aligned}
[j].\,[i].\,+ &\longrightarrow [k] \qquad \text{where } k = i + j\\
[N].\,[M].\,[\top].\,\mathsf{if} &\longrightarrow [M]\\
[N].\,[M].\,[\bot].\,\mathsf{if} &\longrightarrow [N]
\end{aligned}$$

This results in an imperative style of programming similar to Haskell's *do*-notation, but with the difference that a term can have *any* number of return values, by pushing multiple items to the stack - or none. Likewise, a function may consume any number of previously returned values.

Example 7. *Consider the following example, where $x = M\ .\ N$ abbreviates a redex $[M].\,\langle x\rangle.\,N$, and the random values of rnd are taken to be natural numbers.*

$$f = (\mathsf{rand}\ .\ \mathsf{set}\ a\ .\ \mathsf{get}\ a)\ .\ f\ .\ f\ .\ +\ .\ \mathsf{print}$$

The term assigns f to be the function that draws a random number, stores it in cell a, and reads the value at a again as its return value. It then executes f twice, sums the results, and prints that. The decoded term and its reduction are as follows. After the first four steps the context $\ldots\,[5]\mathsf{rnd}.\,[2]\mathsf{rnd}.\,[\star]a$ is made explicit, which initializes the

cell a with $\star$ as the null value and provides the random numbers 2 and 5.

$$
\begin{aligned}
&[\mathsf{rnd}\langle x\rangle \,.\, \underline{[x] \,.\, \langle y\rangle}.\, a\langle\_\rangle.\, [y]a \,.\, a\langle z\rangle.\, [z]a.\, [z]] \,.\, \langle f\rangle \,.\, f \,.\, f \,.\, + \,.\, \langle p\rangle.\, [p]out &\longrightarrow\\
&[\mathsf{rnd}\langle x\rangle \,.\, .\, a\langle\_\rangle.\, \underline{[x]a \,.\, a\langle z\rangle}.\, [z]a.\, [z]] \,.\, \langle f\rangle \,.\, f \,.\, f \,.\, + \,.\, \langle p\rangle.\, [p]out &\longrightarrow\\
&\underline{[\mathsf{rnd}\langle x\rangle \,.\, a\langle\_\rangle \,.\, [x]a.\, [x]]} \,.\, \langle f\rangle \,.\, f \,.\, f \,.\, + \,.\, \langle p\rangle.\, [p]out &\longrightarrow\\
&\mathsf{rnd}\langle x\rangle.\, a\langle\_\rangle.\, \underline{[x]a}.\, [x] \,.\, \mathsf{rnd}\langle x\rangle.\, \underline{a\langle\_\rangle}.\, [x]a.\, [x] \,.\, + \,.\, \langle p\rangle.\, [p]out &\longrightarrow\\
&\mathsf{rnd}\langle x\rangle.\, a\langle\_\rangle.\, [x] \,.\, \mathsf{rnd}\langle x\rangle.\, [x]a.\, [x] \,.\, + \,.\, \langle p\rangle.\, [p]out
\end{aligned}
$$

$$
\begin{aligned}
&\ldots [5]\mathsf{rnd}.\, [2]\mathsf{rnd}.\, \underline{[\star]a} \,.\, \mathsf{rnd}\langle x\rangle.\, \underline{a\langle\_\rangle}.\, [x] \,.\, \mathsf{rnd}\langle x\rangle.\, [x]a.\, [x] \,.\, + \,.\, \langle p\rangle.\, [p]out &\longrightarrow\\
&\ldots [5]\mathsf{rnd}.\, \underline{[2]\mathsf{rnd} \,.\, \mathsf{rnd}\langle x\rangle}.\, [x] \,.\, \mathsf{rnd}\langle x\rangle.\, [x]a.\, [x] \,.\, + \,.\, \langle p\rangle.\, [p]out &\longrightarrow\\
&\ldots \underline{[5]\mathsf{rnd}} \,.\, [2] \,.\, \underline{\mathsf{rnd}\langle x\rangle}.\, [x]a.\, [x] \,.\, + \,.\, \langle p\rangle.\, [p]out &\longrightarrow\\
&\ldots \underline{[2]} \,.\, [5]a.\, \underline{[5] \,.\, +} \,.\, \langle p\rangle.\, [p]out &\longrightarrow\\
&\ldots [5]a.\, \underline{[7] \,.\, \langle p\rangle}.\, [p]out &\longrightarrow\\
&\ldots [5]a \,.\, [7]out
\end{aligned}
$$

The overal result is to consume the random inputs 2 and 5, update the cell a with the value 5, and send 7 to output.

The two modifications that make the FMC, *locations* and *sequentiality*, are independent, and conservative: they can be undone by forcing $\mathcal{A} = \{\lambda\}$ and, respectively, forcing sequential variables and nil to always occur together. This gives three fragments of the FMC, where that without locations but with sequentiality will be called the *sequential λ-calculus*:

- The λ-calculus:
$$N, M \;::=\; x.\star \;\mid\; [M].\, N \;\mid\; \langle x\rangle.\, N$$
- The poly-λ-calculus:
$$N, M \;::=\; x.\star \;\mid\; [M]a.\, N \;\mid\; a\langle x\rangle.\, N$$
- The *sequential λ-calculus:*
$$N, M \;::=\; \star \;\mid\; x.\, N \;\mid\; [M].\, N \;\mid\; \langle x\rangle.\, N$$

The decomposition of the variable is a subtle but profound change to the λ-calculus, whose consequences are hard to oversee immediately. Reassurance that it is well behaved will come from properties proved later. First, I will argue that from the perspective of the abstract machine, and that of the ultimate aim of unifying functional and imperative computation, it is also a natural adaptation.

Sequencing: Imperative languages are built around a *sequence* operator, usually $(;)$, that strings together commands, and while it is possible to introduce this into the λ-calculus as a primitive, a true merger of the two paradigms

may be expected to feature sequencing as a more fundamental notion. The *capture-avoiding composition* of the sequential λ-calculus fits this description well. Like sequencing, it is right-associative (modulo capture-avoidance), and it has the *end* term $\star$ as a unit, like the empty command skip is for sequencing. We may add the following definitions for clarity and completeness:

$$M\,;N \triangleq M.\,N \qquad\qquad \mathsf{skip} \triangleq \star$$

Machine termination: From the perspective of the machine, the *variable* construct of the λ-calculus serves a dual purpose: to be substituted for, and to terminate a sequence of instructions. There is no reason why these rôles should not be separated: it is perfectly fine for an instruction sequence to end with a push or pop action. More strongly, there is *every* reason to separate these rôles, since they are opposites: a variable that needs to be substituted for implies the *continuation* of a run, not its *termination*. Even more strongly: by merging these rôles, *successful* termination of the machine is ruled out, and a run may only end with a free variable or an abstraction on an empty stack, both of which are really *errors*. Arguably, then, the only final state of the machine should be that with the *end* construct, with those for *variable* and *abstraction* classed as failure to terminate successfully. A formal reason is that a run

$$\frac{(\,\varepsilon\;,\,N\,)}{(\,S\;,\;\star\,)}$$

can be *continued*, in the sense that a run for the composition $N.\,M$ would start as above, and then continue to evaluate M in the state (S, M). A run ending in a variable or abstraction is and remains stuck, and cannot be recovered by composition.

Values and commands: Calculi that combine functional and imperative features are often forced to distinguish *values* and *commands*. The former are computations that return an answer, associated with the functional side; the latter are computations that do not return an answer, whose main purpose is to modify the state, or some other effect. From the present perspective, *commands* would be characterized by the machine terminating successfully in an *end*, and *values* such as regular λ-terms would have runs ending in a variable or abstraction, which is then returned. The effect of decomposing the variable is then to include commands into the calculus.

However, the previous point on *machine termination* gave a stronger conclusion: that ending in a variable or abstraction is not a *value*, but an *error*. This conforms to the expectation that a value should available for use by a further computation, while as argued above, a run ending in an abstraction or variable cannot be recovered. But then, if all terms are commands or errors, how do they return anything useful, apart from through mutable store? The answer is that in a run

$$\frac{(\,\varepsilon\qquad\;,\,N\,)}{(\,S\,;\,S_\lambda\,,\;\star\;)}$$

the stack S_λ for regular abstraction and application should be considered to hold the return values, since in a further composition $N.\,M$, these are exactly what is picked up by the abstractions of M. This perspective renders the distinction between *values* and *commands* moot, since their features are compatible: a successful run leaves return values on the λ-stack, but not on the main sequence of the term, the spine, as traversed by the machine.

4 Confluence

For an intuition why the FMC retains conflence, we consider both modifications, *sequencing* and *locations*, in turn. Sequencing creates a new configuration $N.\,x.\,M$ to consider. Reduction on this term is either reduction in N (with substitutions in $x.\,M$) or reduction in M, and so this does not create new critical pairs. Note that the application and abstraction of a redex may only be separated by actions, not by a variable, so that for example $[M].\,x.\,\langle y\rangle.\,N$ is not a redex.

Reduction with locations creates two configurations of interlocking redexes, of the following two forms:

$$[M]a \ldots [N]b \ldots b\langle y\rangle \ldots a\langle x\rangle.\,P$$

$$[M]a \ldots [N]b \ldots a\langle x\rangle \ldots b\langle y\rangle.\,P$$

But these are equivalent under $\sim$ to the more familiar form:

$$[M]a.\,a\langle x\rangle \ldots [N]b.\,b\langle y\rangle \ldots .\,P$$

The confluence proof is then standard by the parallel-reduction technique by Martin-Löf, Tait, and Takahashi [16].

Theorem 8. *Reduction $\twoheadrightarrow$ is confluent.*

Proof. See Appendix A. □

5 Types

Simple types for the FMC will be subtly different from familiar type systems for the λ-calculus. This is almost by necessity: the cbn and cbv λ-calculus have the same simple types, but different interpretations in the FMC, so we cannot expect to unambiguously derive types for the FMC this way. And unlike previous approaches, the FMC is not rooted in a semantics from which types might be construed. Instead, we will continue as we have: following operational considerations. Remarkably, this will lead to a simple conjunction–implication system, semantically a Cartesian closed category, without any primitive monadic functors, though parametrized in *locations*. Avoiding the complication of parametricity, we will start with types for the sequential λ-calculus.

Definition 9. *Sequential types* are given by the following grammar, where the only constructor is an *implication*.

$$\rho, \sigma, \tau, \upsilon \ ::= \ \sigma_n \dots \sigma_1 \Rightarrow \tau_1 \dots \tau_m$$

That is, a type ρ consists of a vector $\sigma_n \dots \sigma_1$ of *antecedents* and a vector $\tau_1 \dots \tau_m$ of *consequents*, each of which may be empty. Type vectors will be written $\vec{\sigma} = \sigma_1 \dots \sigma_n$, with the same vector in reverse as $\overleftarrow{\sigma} = \sigma_n \dots \sigma_1$, so that a type is given as $\overleftarrow{\sigma} \Rightarrow \vec{\tau}$. A type vector can intuitively be seen as a conjunction, and a type can be interpreted with standard implication and conjunction as follows.

$$(\sigma_n \wedge \ \dots \ \wedge \sigma_1) \to (\tau_1 \wedge \ \dots \ \wedge \tau_m)$$

We extend vector notation to variables, $\vec{x} = x_1, \dots, x_n$, and to simultaneous substitutions: if $T = \varepsilon \cdot N_1 \cdots N_n$ then

$$\{T/\vec{x}\} = \{N_1/x_1, \dots, N_n/x_n\} \ .$$

The intuitive meaning of a type is to describe the behaviour of the stack machine. A term will have a type, and a stack a vector of types. Then if N has type $\overleftarrow{\sigma} \Rightarrow \vec{\tau}$ and S has the types $\vec{\sigma}$, the machine will have a run

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

producing a stack T with the types $\vec{\tau}$. Thus, a type describes the *net* behaviour of the abstract machine: only the initial and final configurations, and not the intermediate stack use.

Definition 10. A sequential term N is *typed* by $\Gamma \vdash N : \tau$ according to the rules in Figure 2, where Γ is a *context* of typed variables

$$\Gamma \ = \ \vec{x} : \vec{\sigma} \ = \ x_1 : \sigma_1, \dots, x_n : \sigma_n \ .$$

A stack $S = \varepsilon \cdot N_1 \cdots N_n$ is typed $\Gamma \vdash S : \vec{\tau}$ if $\vec{\tau} = \tau_1 \dots \tau_n$ and $\Gamma \vdash N_i : \tau_i$ for $1 \leq i \leq n$.

Example 11. *The term $\lambda x.\, x\, x = \langle x \rangle.\, [x.\star].\, x.\star$ can be typed by assigning x a type of the form $\Rightarrow \vec{\tau}$ that does not consume input. In this way, the self-application does not create a clash. The corresponding most general derivation (without instantiating $\vec{\tau}$) is as follows.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \Rightarrow \vec{\tau} \ \vdash \ \star : \overleftarrow{\tau} \Rightarrow \vec{\tau}}\,\star}{x : \Rightarrow \vec{\tau} \ \vdash \ x.\star : \Rightarrow \vec{\tau}}\,\mathsf{var} \qquad \dfrac{\dfrac{}{x : \Rightarrow \vec{\tau} \ \vdash \ \star : \overleftarrow{\tau}(\Rightarrow\vec{\tau}) \Rightarrow (\Rightarrow\vec{\tau})\vec{\tau}}\,\star}{x : \Rightarrow \vec{\tau} \ \vdash \ x.\star : (\Rightarrow\vec{\tau}) \Rightarrow (\Rightarrow\vec{\tau})\vec{\tau}}\,\mathsf{var}}{x : \Rightarrow \vec{\tau} \ \vdash \ [x.\star].\, x.\star : \ \Rightarrow (\Rightarrow\vec{\tau})\vec{\tau}}\,\mathsf{app}}{\vdash \ \langle x \rangle.\, [x.\star].\, x.\star : (\Rightarrow\vec{\tau}) \Rightarrow (\Rightarrow\vec{\tau})\vec{\tau}}\,\mathsf{abs}$$

$$\frac{}{\Gamma \vdash \star : \overleftarrow{\tau} \Rightarrow \vec{\tau}}\ \star$$

$$\frac{\Gamma, x : \overleftarrow{\rho} \Rightarrow \vec{\sigma} \vdash N : \overleftarrow{\sigma}\,\overleftarrow{\tau} \Rightarrow \vec{\upsilon}}{\Gamma, x : \overleftarrow{\rho} \Rightarrow \vec{\sigma} \vdash x.\, N : \overleftarrow{\rho}\,\overleftarrow{\tau} \Rightarrow \vec{\upsilon}}\ \mathsf{var}$$

$$\frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \rho\,\overleftarrow{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash [M].\, N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}}\ \mathsf{app}$$

$$\frac{\Gamma, x : \rho \vdash N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash \langle x \rangle.\, N : \rho\,\overleftarrow{\sigma} \Rightarrow \vec{\tau}}\ \mathsf{abs}$$

Figure 2: Natural-deduction typing rules for the sequential λ-calculus

For effects, this is crucial: the call–by–name read *construct* $!a = a\langle x \rangle.\, [x]a.\, x$ *is just* $\lambda x.\, xx$ *but on a different location* a*, and should in many cases be typeable. For anything which does not consume input from* a*, it would. But the fixed point combinators of Example* **??** *are not typeable, nor is* $(\lambda x.\, xx)(\lambda x.\, xx)$*, since the argument matching the first* λx *takes input.*

Primitives can be typed by extending types with constants κ, which will here include $\mathbb{B}$ for Booleans and $\mathbb{Z}$ for integers.

$$\rho, \sigma, \tau, \upsilon \ ::=\ \kappa \ \mid\ \overleftarrow{\sigma} \Rightarrow \vec{\tau}$$

The types for primitives are then as follows.

$$\begin{array}{rrrr} \top, \bot & : & \mathbb{B} & \qquad \mathsf{if} \ : \ \tau\,\tau\,\mathbb{B} \Rightarrow \tau \quad (\text{for any } \tau) \\ \ldots, -1, 0, 1, \ldots & : & \mathbb{Z} & \qquad +, \times \ : \ \mathbb{Z}\,\mathbb{Z} \Rightarrow \mathbb{Z} \end{array}$$

The type system creates a natural distinction between *values* of type κ and *computations* of type $\overleftarrow{\sigma} \Rightarrow \vec{\tau}$. The pure calculus, without constants, has only computations, and the type system does not allow variables to range over values, since the typing rule for $x.\, N$ requires x to have a computation type. To manipulate values, we need to re-introduce a typed variable without continuation, $x : \kappa$, as a term. For this and the typed constants $k : \kappa$ and $f : \overleftarrow{\rho} \Rightarrow \vec{\sigma}$ we have the following typing rules, and the corresponding machine transition for f, which determines for every input stack $R : \vec{\rho}$ an output stack $S : \vec{\sigma}$.

$$\frac{}{\Gamma, x : \kappa \vdash x : \kappa} \qquad \frac{}{\Gamma \vdash k : \kappa} \qquad \frac{\Gamma \vdash \quad N : \overleftarrow{\sigma}\,\overleftarrow{\tau} \Rightarrow \vec{\upsilon}}{\Gamma \vdash f.\, N : \overleftarrow{\rho}\,\overleftarrow{\tau} \Rightarrow \vec{\upsilon}} \qquad \frac{(\ T \cdot R\ ,\ f.\, N\ )}{(\ T \cdot S\ ,\quad N\ )}$$

The following proposition establishes four basic properties, including the familiar *subject reduction*. The *expansion* property states that if a term expecting a stack $R : \vec{\rho}$ is given a stack $T \cdot R : \vec{\tau}\,\vec{\rho}$, then it returns the remaining stack $T : \vec{\tau}$ untouched.

Proposition 12. *Typed terms satisfy the following properties:*

- *Expansion: if* $\Gamma \vdash N : \overleftarrow{\rho} \Rightarrow \vec{\sigma}$ *then* $\Gamma \vdash N : \overleftarrow{\rho}\,\overleftarrow{\tau} \Rightarrow \vec{\tau}\,\vec{\sigma}$.
- *Composition: if* $\Gamma \vdash N : \overleftarrow{\rho} \Rightarrow \vec{\sigma}$ *and* $\Gamma \vdash M : \overleftarrow{\sigma} \Rightarrow \vec{\tau}$ *then* $\Gamma \vdash N.\, M : \overleftarrow{\rho} \Rightarrow \vec{\tau}$.
- *Subject substitution: if* $\Gamma \vdash M : \sigma$ *and* $\Gamma, x : \sigma \vdash N : \tau$ *then* $\Gamma \vdash \{M/x\}N : \tau$.
- *Subject reduction: if* $\Gamma \vdash N : \tau$ *and* $N \longrightarrow M$ *then* $\Gamma \vdash M : \tau$.

Proof. The first three properties are by induction on N, and the last by induction on the context of the reduction step. □

A remarkable aspect of the type system is how it gives a direct connection with termination of the machine. To expose this, we formalize the intuitive meaning of types as describing the initial and final stack of a run of the machine.

Definition 13. The set $\textsc{run}(\overleftarrow{\sigma} \Rightarrow \vec{\tau})$ is the set of terms N such that for any stack $S \in \textsc{run}(\vec{\sigma})$ there is a stack $T \in \textsc{run}(\vec{\tau})$ and a run of the machine

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

where $\textsc{run}(\tau_1 \dots \tau_n)$ is the set of stacks $\varepsilon \cdot N_1 \cdots N_n$ such that $N_i \in \textsc{run}(\tau_i)$.

If $N : \tau$ implies $N \in \textsc{run}(\tau)$, then a type derivation *is* a termination proof of the machine. This is Theorem 14, and proving it gives an interestingly concrete *Tait-style* reducibility proof [15], where $\textsc{run}(\tau)$ takes the rôle of the reducibility set for τ. By using the properties of machine runs, such as *expansion* and *composition* as given for types in Proposition 12, the proof is then a simple, direct induction on type derivations.

Theorem 14. *If* $\vec{w} : \vec{\omega} \vdash N : \tau$ *then for any* $W \in \textsc{run}(\vec{\omega})$, $\{W/\vec{w}\,\}N \in \textsc{run}(\tau)$.

Proof. By induction on the type derivation. In each case, let $\Gamma = \overleftarrow{w} : \overleftarrow{\omega}$, let W be a stack in $\textsc{run}(\vec{\omega})$, and let $N' = \{W/\overleftarrow{w}\}N$.

- If the derivation is a $\star$-rule for $\Gamma \vdash \star : \overleftarrow{\tau} \Rightarrow \vec{\tau}$ then there is a trivial zero-step run from the state $(T, \star)$ to itself.
- If the derivation ends in

$$\frac{\Gamma, x : \overleftarrow{\rho} \Rightarrow \vec{\sigma} \vdash \quad N : \overleftarrow{\sigma}\,\overleftarrow{\tau} \Rightarrow \vec{\upsilon}}{\Gamma, x : \overleftarrow{\rho} \Rightarrow \vec{\sigma} \vdash x.\, N : \overleftarrow{\rho}\,\overleftarrow{\tau} \Rightarrow \vec{\upsilon}}\ \textsf{var}$$

  then for any $M \in \textsc{run}(\vec{\rho} \Rightarrow \vec{\sigma})$, the inductive hypothesis gives a run for $\{M/x\}N'$ from any $T \cdot S \in \textsc{run}(\vec{\tau}\,\vec{\sigma})$ to some $U \in \textsc{run}(\vec{\upsilon})$ (below left). For M there is a run from any $R \in \textsc{run}(\vec{\rho})$ to some $S \in \textsc{run}(\vec{\sigma})$ (below right).

$$\frac{(\ T \cdot S\ ,\ \{M/x\}N'\ )}{(\ U\quad ,\quad \star\ )} \qquad \frac{(\ R\ ,\ M\ )}{(\ S\ ,\ \star\ )}$$

These runs compose into one for $\{W/\vec{w},\, M/x\}x.\, N \;=\; M.\, \{M/x\}N'$ as below, expanding the stack on the run for M by T.

$$\dfrac{\dfrac{(\; T \cdot R \;,\; M.\, \{M/x\}N' \;)}{(\; T \cdot S \;,\; \{M/x\}N' \;)}}{(\; U \;,\; \star \;)}$$

- If the derivation ends in

$$\dfrac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \rho \overleftarrow{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash [M].\, N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}}\ \textsf{app}$$

then by the inductive hypothesis, $M' = \{W/\vec{w}\}M \in \textsc{run}(\rho)$, and there is a run for N' from M' and any $S \in \textsc{run}(\overleftarrow{\sigma})$ to some $T \in \textsc{run}(\vec{\tau})$. This gives the following run for $\{W/\vec{w}\}[M].\, N = [M'].\, N'$.

$$\dfrac{\dfrac{(\; S \;,\; [M'].\, N' \;)}{(\; S \cdot M' \;,\; N' \;)}}{(\; T \;,\; \star \;)}$$

- If the derivation ends in

$$\dfrac{\Gamma, x : \rho \vdash N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash \langle x \rangle.\, N : \rho \overleftarrow{\sigma} \Rightarrow \vec{\tau}}\ \textsf{abs}$$

then for any $M \in \textsc{run}(\rho)$ and $S \in \textsc{run}(\overleftarrow{\sigma})$ the inductive hypothesis gives a run for $\{M/x\}N'$ to some $T \in \textsc{run}(\vec{\tau})$. This gives the following run for $\{W/\vec{w}\}\langle x \rangle.\, N = \langle x \rangle.\, N'$.

$$\dfrac{\dfrac{(\; S \cdot M \;,\; \langle x \rangle.\, N' \;)}{(\; S \;,\; \{M/x\}N' \;)}}{(\; T \;,\; \star \;)}$$

□

Corollary 15. *For a typed term the machine terminates.*

Appendix C will show that reduction ($\twoheadrightarrow$) on typed sequential λ-terms is strongly normalizing, by typing the continuation encoding of Section 8 with second-order types.

6 Poly-types

It remains to lift sequential types to the FMC. This is conceptually simple: analogously to the change from a single-stack to a multiple-stack abstract machine, a type for the FMC will be an implication not between *single* type vectors $\vec{\tau}$, but a *family* of vectors $\vec{\tau}_{\mathcal{A}} = \{\vec{\tau}_a \mid a \in \mathcal{A}\}$ parameterized in the set of locations $\mathcal{A}$.

$$\frac{}{\Gamma \vdash \star \colon \overleftarrow{\tau}_{\mathcal{A}} \Rightarrow \vec{\tau}_{\mathcal{A}}}\ \star$$

$$\frac{\Gamma, x \colon \overleftarrow{\rho}_{\mathcal{A}} \Rightarrow \vec{\sigma}_{\mathcal{A}} \vdash N \colon \overleftarrow{\sigma}_{\mathcal{A}}\,\overleftarrow{\tau}_{\mathcal{A}} \Rightarrow \vec{\upsilon}_{\mathcal{A}}}{\Gamma, x \colon \overleftarrow{\rho}_{\mathcal{A}} \Rightarrow \vec{\sigma}_{\mathcal{A}} \vdash x.\, N \colon \overleftarrow{\rho}_{\mathcal{A}}\,\overleftarrow{\tau}_{\mathcal{A}} \Rightarrow \vec{\upsilon}_{\mathcal{A}}}\ \text{var}$$

$$\frac{\Gamma \vdash M \colon \rho \qquad \Gamma \vdash N \colon a(\rho)\,\overleftarrow{\sigma}_{\mathcal{A}} \Rightarrow \vec{\tau}_{\mathcal{A}}}{\Gamma \vdash [M]a.\, N \colon \overleftarrow{\sigma}_{\mathcal{A}} \Rightarrow \vec{\tau}_{\mathcal{A}}}\ \text{app}$$

$$\frac{\Gamma, x \colon \rho \vdash N \colon \overleftarrow{\sigma}_{\mathcal{A}} \Rightarrow \vec{\tau}_{\mathcal{A}}}{\Gamma \vdash a\langle x\rangle.\, N \colon a(\rho)\,\overleftarrow{\sigma}_{\mathcal{A}} \Rightarrow \vec{\tau}_{\mathcal{A}}}\ \text{abs}$$

Figure 3: Natural-deduction typing rules for the Functional Machine Calculus

Definition 16. *Poly-types* $\rho, \sigma, \tau, \upsilon$ are given by:

$$\begin{aligned} \tau &::= \overleftarrow{\sigma}_{\mathcal{A}} \Rightarrow \vec{\tau}_{\mathcal{A}} \\ \vec{\tau}_{\mathcal{A}} &::= \{\vec{\tau}_a \mid a \in \mathcal{A}\} \\ \vec{\tau} &::= \tau_1 \ldots \tau_n \end{aligned}$$

The following notation is further added:

- *concatenation:* $\vec{\sigma}_{\mathcal{A}}\vec{\tau}_{\mathcal{A}} = \{\vec{\sigma}_a\vec{\tau}_a \mid a \in \mathcal{A}\}$;
- *injection:* $a(\vec{\tau})$ is the family given by $a(\vec{\tau})_a = \vec{\tau}$ and $a(\vec{\tau})_b = \varepsilon$ (the empty type vector) if $a \neq b$.

Definition 17. An FMC-term is *typed* $\Gamma \vdash N \colon \tau$ by a poly-type τ in a context Γ by the type system in Figure 3.

Example 18. *The* injection *construct* $a(\vec{\tau})$ *gives a natural way of writing types in practice. We may type the term from Example 7 as follows. The type expresses that it pops two integers from* rnd *and one from* a*, and pushes one integer to* a *and one to* out*. (Note that there are other ways of writing the same type.)*

$$f = (\mathsf{rand}\ .\ \mathsf{set}\ a\ .\ \mathsf{get}\ a)\ .\ f\ .\ f\ .\ +\ .\ \mathsf{print} \ :\ \mathsf{rnd}(\mathbb{Z}\ \mathbb{Z})\ a(\mathbb{Z}) \Rightarrow a(\mathbb{Z})\ \mathsf{out}(\mathbb{Z})$$

Various subterms can be given the following types, for any types σ *and* τ*, which ultimately must specialize to* $\mathbb{Z}$ *in the context of the term. As with terms, the default location* λ *is omitted, with* $\lambda(\vec{\tau})$ *written simply as* $\vec{\tau}$*.*

$$\begin{aligned} \mathsf{rand} &= \mathsf{rnd}\langle x\rangle.\,[x] &&: \mathsf{rnd}(\tau) \Rightarrow \tau \\ \mathsf{set}\ a &= \langle x\rangle.\,a\langle\_\rangle.\,[x]a &&: \tau\ a(\sigma) \Rightarrow a(\tau) \\ \mathsf{get}\ a &= a\langle x\rangle.\,[x]a.\,[x] &&: a(\tau) \Rightarrow a(\tau)\ \tau \\ \mathsf{rand}\ .\ \mathsf{set}\ a\ .\ \mathsf{get}\ a &&&: \mathsf{rnd}(\tau)\ a(\sigma) \Rightarrow a(\tau)\ \tau \\ f = (\mathsf{rand}\ .\ \mathsf{set}\ a\ .\ \mathsf{get}\ a)\ .\ f\ .\ f &&&: \mathsf{rnd}(\tau\ \tau)\ a(\tau) \Rightarrow a(\tau)\ \tau\ \tau \end{aligned}$$

Theorem 19. *For a typed* FMC*-term the Functional Abstract Machine terminates.*

Proof. Analogous to the proof of Theorem 14. □

7 Encoding call–by–(push–)value

Various calculi have been proposed to capture both call–by–value and call–by–name strategies, to which effects are be added to make use of this feature. Here, we will consider Plotkin's call–by–value [12], Moggi's Computational Metalanguage [10], Ehrhard and Guerrieri's Bang Calculus [5], and Levy's call–by–push–value [8].

Call–by–value: The call–by–value λ-calculus [12] is a restriction of the λ-calculus that limits β-reduction to the case where the argument is a *value* V, a variable or an abstraction:

$$V, W ::= x \mid \lambda x.N \qquad (\lambda x.M)V \to_{\mathtt{v}} \{V/x\}M$$

Definition 20. The *call-by-value translation* $N_{\mathtt{v}}$ of a λ-term N into an FMC-term is given by:

$$\begin{aligned} x_{\mathtt{v}} &= [x] \\ (\lambda x.N)_{\mathtt{v}} &= [\langle x\rangle.\, N_{\mathtt{v}}] \\ (N\,M)_{\mathtt{v}} &= M_{\mathtt{v}}.\, N_{\mathtt{v}}.\, \langle x\rangle.\, x \end{aligned}$$

The idea of this translation is that a *value* is the result of a successful computation, which returns it by pushing it on the stack. The interpretation of an application NM first evaluates the argument $M_{\mathtt{v}}$, pushing the result onto the stack; then the same for the function $N_{\mathtt{v}}$, leaving both returned values on the stack; and finally the function is *called* by popping it from the stack and executing it.

Theorem 21. *If* $M \to_{\mathtt{v}} N$ *then* $M_{\mathtt{v}} \twoheadrightarrow N_{\mathtt{v}}$.

Proof. Let $(\lambda x.M)V \to_{\mathtt{v}} \{V/x\}M$ and let $V_{\mathtt{v}} = [N]$. Then

$$\begin{aligned} [N].\,[\langle x\rangle.\, M_{\mathtt{v}}].\,\langle y\rangle.\, y &\to [N].\,\langle x\rangle.\, M_{\mathtt{v}} \\ &\to \{N/x\}M_{\mathtt{v}} \\ &= (\{V/x\}M)_{\mathtt{v}} \end{aligned}$$

where the last equality is a straightforward induction on M with the following base case.

$$\{N/x\}x_{\mathtt{v}} = \{N/x\}[x] = [N] = V_{\mathtt{v}} = (\{V/x\}x)_{\mathtt{v}} \qquad \square$$

The store operators *update* and *read* are given a call–by–value interpretation as follows.

$$\begin{aligned} (a := N)_{\mathtt{v}} &= N.\,\langle x\rangle.\, a\langle \_\rangle.\, [x]a \\ (!a)_{\mathtt{v}} &= a\langle x\rangle.\, [x]a.\, [x] \end{aligned}$$

The behaviour of *update* is to evaluate N, pop its return value from the stack as x, and use that to replace the first element on the stack for a. The behaviour of *read* is as for the call–by–name interpretation, except the value that is read is put on the stack instead of evaluated immediately.

Example 22. *The call–by–value reduction from Example 4,*

$$a := 2\,;(\lambda x.\,!a)\,(a := 3\,;5) \;\to_{\mathsf{cbv}}\; 3$$

is captured as follows. Numbers may again be taken as constants or Church numerals; in both cases, they are values, and the translation will use $i_{\mathrm{v}} = [i]$*. The two update operations are then translated and simplified as follows:*

$$\begin{aligned}(a := 2)_{\mathrm{v}} &= \underline{[2].\,\langle x\rangle.\,a\langle\_\rangle.\,[x]a} \;\to\; a\langle\_\rangle.\,[2]a\\ (a := 3)_{\mathrm{v}} &= \underline{[3].\,\langle x\rangle.\,a\langle\_\rangle.\,[x]a} \;\to\; a\langle\_\rangle.\,[3]a\end{aligned}$$

The remaining cbv*-interpretation and reduction of the example is as follows.*

$$\begin{aligned}&a\langle\_\rangle.\,\underline{[2]a.\,a\langle\_\rangle.\,[3]a.\,[5]}.\,[\langle x\rangle.\,a\langle y\rangle.\,[y]a.\,[y]].\,\langle z\rangle.\,z\\ \to\;&a\langle\_\rangle.\,[3]a.\,[5].\,\underline{[\langle x\rangle.\,a\langle y\rangle.\,[y]a.\,[y]].\,\langle z\rangle.\,z}\\ \to\;&a\langle\_\rangle.\,[3]a.\,\underline{[5].\,\langle x\rangle.\,a\langle y\rangle.\,[y]a.\,[y]}\\ \to\;&a\langle\_\rangle.\,\underline{[3]a.\,a\langle y\rangle.}\,[y]a.\,[y]\\ \to\;&a\langle\_\rangle.\,[3]a.\,[3]\\ \leftarrow\;&(a := 3\,;3)_{\mathrm{v}}\end{aligned}$$

The computational metalanguage: Moggi's *computational metalanguage* [10] extends the call–by–name λ-calculus with two constructs, parameterized in a monad T: *return* $[N]_T$, which *for any* T is included as $[N]$ in the FMC; and *let*, included as follows, again *for any* T:

$$\mathtt{let}_T\; x \Leftarrow N\; \mathtt{in}\; M \;=\; N.\,\langle x\rangle.\,M$$

The translation directly gives the required reduction:

$$\mathtt{let}_T\; x \Leftarrow [N]_T\; \mathtt{in}\; M \;=\; [N].\,\langle x\rangle.\,M \;\to\; \{N/x\}M$$

The use of effects in this setting is illustrated by an example.

Example 23. *We will look at two interpretations, one for* cbn *and one for* cbv*, of the non-confluent term of Example 4,*

$$a := 2\,;(\lambda x.\,!a)\,(a := 3\,;5)$$

in the metalanguage with state *monad. Monadic* update $a :- N \;=\; a\langle\_\rangle.\,[N]a.\,[\star]$ *returns a unit, and* read *gives a monadic return value,* $\mathtt{read}\; a = a\langle x\rangle.\,[x]a.\,[x]$*. The* cbn *version and its* FMC *embedding, below, return* $[2]$*.*

$$\begin{aligned}&\mathtt{let}\;\_ \Leftarrow a :- 2\;\mathtt{in}\;(\lambda x.\,\mathtt{read}\;a)\,(\mathtt{let}\;\_ \Leftarrow a :- 3\;\mathtt{in}\;[5])\\ &a\langle\_\rangle.\,[2]a.\,[\star].\,\langle\_\rangle.\,[a\langle\_\rangle.\,[3]a.\,[\star].\,\langle\_\rangle.\,[5]].\,\langle x\rangle.\,a\langle z\rangle.\,[z]a.\,[z]\end{aligned}$$

The cbv *version, below, returns* $[3]$*.*

$$\begin{aligned}&\mathtt{let}\;\_ \Leftarrow a :- 2\;\mathtt{in}\;(\mathtt{let}\;\_ \Leftarrow a :- 3\;\mathtt{in}\;(\lambda x.\,\mathtt{read}\;a)\,[5])\\ &a\langle\_\rangle.\,[2]a.\,[\star].\,\langle\_\rangle.\,a\langle\_\rangle.\,[3]a.\,[\star].\,\langle\_\rangle.\,[[5]].\,\langle x\rangle.\,a\langle z\rangle.\,[z]a.\,[z]\end{aligned}$$

An interesting observation is that for metalanguage terms embedded into the FMC, redexes are *properly nested*, which means that the restricted reduction relation $[M]a.\, a\langle x\rangle.\, N \;\longrightarrow\; \{M/x\}N$ becomes sufficient. This relates to the fact that the metalanguage has a separate monad for each effect, while effects can mix freely in the FMC.

The bang-calculus: The *bang-calculus* [5] is a simplification of call-by-push-value based on linear logic. It, too, encodes both call–by–value and call–by–name. The syntax distinguishes *values* V, W as a subset of *terms* S, T, as given by the following grammars.

$$S, T ::= V \mid \lambda x.T \mid \langle T\rangle S \mid \mathtt{der}\ T \qquad V, W ::= x \mid T^{!}$$

There are two reduction rules:

$$\langle \lambda x.\, T\rangle V \;\longrightarrow_{\mathsf{b}}\; \{V/x\}T \qquad \mathtt{der}(T^{!}) \;\longrightarrow_{!}\; T$$

Definition 24. The *bang-translation* T_{b} of a bang-term T into an FMC-term is given by:

$$\begin{aligned} x_{\mathsf{b}} &= [x] \\ (T^{!})_{\mathsf{b}} &= [T_{\mathsf{b}}] \\ (\lambda x.\, T)_{\mathsf{b}} &= \langle x\rangle.\, T_{\mathsf{b}} \\ (\langle T\rangle S)_{\mathsf{b}} &= S_{\mathsf{b}}.\, T_{\mathsf{b}} \\ (\mathtt{der}\ T)_{\mathsf{b}} &= T_{\mathsf{b}}.\, \langle x\rangle.\, x \end{aligned}$$

Some interesting aspects stand out about this translation. One is how the bang-calculus allows many of the key constructions of the sequential λ-calculus, such as a term ending in a push action in $T^{!}$, and sequencing in $\langle T\rangle S$. The main missing construct is the sequential variable $x.\, N$, but this can be simulated by a dereliction and sequencing: $\langle T\rangle\, (\mathtt{der}\ x)$. This gives a partial return translation for the sequential λ-calculus, below, which covers all terms except $\star$ and those ending in a pop action, $\langle x\rangle.\, \star$. The other thing that stands out is how close also the reduction relations are, which is expressed in the two subsequent theorems.

Definition 25. The *sequencing translation* N_s is a partial interpretation of the sequential λ-calculus into the bang-calculus, given by:

$$\begin{aligned} (x.\, N)_{\mathsf{s}} &= \langle N_{\mathsf{s}}\rangle\, (\mathtt{der}\ x) & (x.\, \star)_{\mathsf{s}} &= \mathtt{der}\ x \\ ([M].\, N)_{\mathsf{s}} &= \langle N_{\mathsf{s}}\rangle\, (M_{\mathsf{s}})^{!} & ([M].\, \star)_{\mathsf{s}} &= (M_{\mathsf{s}})^{!} \\ (\langle x\rangle.\, N)_{\mathsf{s}} &= \lambda x.\, N_{\mathsf{s}} \end{aligned}$$

Theorem 26. *If* $T \longrightarrow_{\mathsf{b}} S$ *or* $T \longrightarrow_{!} S$ *then* $T_{\mathsf{b}} \longrightarrow S_{\mathsf{b}}$.

Proof. Let $V_{\mathsf{b}} = [N]$. Then:

$$\begin{aligned} (\langle \lambda x.\, T\rangle V)_{\mathsf{b}} &= [N].\, \langle x\rangle.\, T_{\mathsf{b}} \longrightarrow \{N/x\}T_{\mathsf{b}} = (\{V/x\}T)_{\mathsf{b}} \\ (\mathtt{der}(T^{!}))_{\mathsf{b}} &= [T_{\mathsf{b}}].\, \langle x\rangle.\, x \longrightarrow T_{\mathsf{b}} \end{aligned}$$

□

Theorem 27. *If $N \longrightarrow M$ and N_{s} exists then $N_{\mathsf{s}} \longrightarrow_{\mathsf{b}} \twoheadrightarrow_{!} M_{\mathsf{s}}$.*

Proof. The reduction is below. The last step follows because in a translation N_{s} every variable occurs as $\mathtt{der}\, x$, and $\{T^{!}/x\}\, \mathtt{der}\, x = \mathtt{der}\, (T^{!}) \longrightarrow_{!} T$.

$$\begin{aligned} ([M].\, \langle x \rangle .\, N)_{\mathsf{s}} \;&=\; \langle \lambda x.\, N_{\mathsf{s}} \rangle\, (M_{\mathsf{s}})^{!} \\ &\longrightarrow_{\mathsf{b}}\; \{(M_{\mathsf{s}})^{!}/x\} N_{\mathsf{s}} \\ &\longrightarrow\!\!\!\!\longrightarrow_{!}\; (\{M/x\}N)_{\mathsf{s}} \end{aligned}$$

□

Call–by–push–value: Call–by–push–value (cbpv) [8, 9] is an earlier approach to combining call–by–name and call–by–value that stands out for the thorough study of its semantics. Following its many models, it features a strict separation between *computations* and *values*, which is semantically helpful but leads to an elaborate syntax. Here we consider only the core fragment, without products and coproducts. We follow the exposition in [9].

Definition 28. The terms of *call–by–push–value* and their translation $N \Mapsto N_{\mathsf{p}}$ into the FMC are as follows.

$$\begin{array}{lrcl} \text{Values } V, W \;::= & x & \Mapsto & x \\ & \mathtt{return}\, V & \Mapsto & [V_{\mathsf{p}}] \\ & \mathtt{thunk}\, M & \Mapsto & [M_{\mathsf{p}}] \\ \\ \text{Terms } M, N \;::= & \lambda x.M & \Mapsto & \langle x \rangle .\, M_{\mathsf{p}} \\ & V \text{`} M & \Mapsto & [V_{\mathsf{p}}].\, M_{\mathsf{p}} \\ & \mathtt{let}\, V\, \mathtt{be}\, x\, .\, M & \Mapsto & [V_{\mathsf{p}}].\, \langle x \rangle .\, M_{\mathsf{p}} \\ & N\, \mathtt{to}\, x\, .\, M & \Mapsto & N_{\mathsf{p}}.\, \langle x \rangle .\, M_{\mathsf{p}} \\ & \mathtt{force}\, M & \Mapsto & M_{\mathsf{p}}.\, \langle x \rangle .\, x \end{array}$$

The FMC then simulates cbpv. The following theorem gives the big-step semantics $M \Downarrow T$, with the *terminal* term T of the form $\lambda x.M$ or $\mathtt{return}\, V$, and demonstrates the simulation.

Theorem 29. *If $M \Downarrow T$ then $M_{\mathsf{p}} \twoheadrightarrow T_{\mathsf{p}}$.*

Proof. On the left is the big-step semantics of cbpv (see [9, Figure 4]), with premises stacked for space; on the right are the corresponding reductions in

the FMC.

$$\frac{}{\texttt{return}\, V \;\Downarrow\; \texttt{return}\, V} \qquad [V_\mathsf{p}] \;=\; [V_\mathsf{p}]$$

$$\frac{}{\lambda x.\, M \;\Downarrow\; \lambda x.\, M} \qquad \langle x\rangle.\, M_\mathsf{p} \;=\; \langle x\rangle.\, M_\mathsf{p}$$

$$\frac{[V/x]M \;\Downarrow\; T}{\texttt{let}\, V\, \texttt{be}\, x.\, M \;\Downarrow\; T} \qquad \begin{aligned}[V_\mathsf{p}].\, \langle x\rangle.\, M_\mathsf{p} &\longrightarrow [V_\mathsf{p}/x]M_\mathsf{p} \\ &\twoheadrightarrow T_\mathsf{p}\end{aligned}$$

$$\frac{M \;\Downarrow\; \texttt{return}\, V \qquad [V/x]N \;\Downarrow\; T}{M\, \texttt{to}\, x.\, N \;\Downarrow\; T} \qquad \begin{aligned}M_\mathsf{p}.\, \langle x\rangle.\, N_\mathsf{p} &\twoheadrightarrow [V_\mathsf{p}].\, \langle x\rangle.\, N_\mathsf{p} \\ &\longrightarrow [V_\mathsf{p}/x]N_\mathsf{p} \\ &\twoheadrightarrow T_\mathsf{p}\end{aligned}$$

$$\frac{M \;\Downarrow\; T}{\texttt{force}\, \texttt{thunk}\, M \;\Downarrow\; T} \qquad \begin{aligned}[M_\mathsf{p}].\, \langle x\rangle.\, x &\longrightarrow M_\mathsf{p} \\ &\twoheadrightarrow T_\mathsf{p}\end{aligned}$$

$$\frac{M \;\Downarrow\; \lambda x.\, N \qquad [V/x]N \;\Downarrow\; T}{V\text{`}M \;\Downarrow\; T} \qquad \begin{aligned}[V_\mathsf{p}].\, M_\mathsf{p} &\twoheadrightarrow [V_\mathsf{p}].\, \langle x\rangle.\, N_\mathsf{p} \\ &\longrightarrow [V_\mathsf{p}/x]N_\mathsf{p} \\ &\twoheadrightarrow T_\mathsf{p}\end{aligned}$$

□

Since the big-step semantics does not reduce under abstractions, the simulation in the FMC guarantees a corresponding run of the Functional Abstract Machine. Of the two types of terminal, as argued earlier we may consider $(\lambda x.N)_\mathsf{p} = \langle x\rangle.\, N_\mathsf{p}$ an error (if it halts the machine, not as output for the big-step semantics), and $(\texttt{return}\, V)_\mathsf{p} = [V_\mathsf{p}]$ a successful return value. That is, if $M \;\Downarrow\; \texttt{return}\, V$ then:

$$\frac{(\ \varepsilon \quad ,\ M_\mathsf{p}\ )}{(\ \varepsilon \cdot V_\mathsf{p}\ ,\ \star\ )}$$

With the machine as operational semantics, also the effects of cbpv can be incorporated. Following [9, Figure 5] *output* and *store* are considered. For output, cbpv extends the semantics with a string m of characters c, concatenated with $+$, giving the following rule for the `print` command.

$$\frac{M \;\Downarrow\; m, T}{\texttt{print}\, c.\, M \;\Downarrow\; [c] + m, T}$$

In the FMC, the `print` c command is the push action $[c]\mathsf{out}$, and m is modelled by the out-stream. The corresponding run of the machine is the following.

$$\frac{\dfrac{(\ S\ ;\ S_\mathsf{out} \quad ,\ [c]\mathsf{out}.\, M_\mathsf{p}\ )}{(\ S\ ;\ S_\mathsf{out} \cdot c \quad ,\ M_\mathsf{p}\ )}}{(\ S\ ;\ S_\mathsf{out} \cdot c \cdot m\ ,\ T_\mathsf{p}\ )}$$

For store, cbpv adds a single memory location `cell` storing values $s, s', s'' \in S$ and commands for *update* `cell` $:= s$ and *read* `read-cell-as` $\{s.M_s\}_{s\in S}$,

where the construction $\{s.M_s\}_{s\in S}$ is a function from S to cbpv-terms. The big-step semantics then carries a stored value s.

$$\frac{s', M \Downarrow s'', T}{s, \mathtt{cell} := s'.M \Downarrow s'', T}$$

$$\frac{s', M_{s'} \Downarrow s'', T}{s', \mathtt{read\text{-}cell\text{-}as}\ \{s.M_s\}_{s\in S} \Downarrow s'', T}$$

Since the FMC features higher-order store, we will consider s, s', s'' as arbitrary terms. The *update* construct is then as for the call–by–name version, $\mathtt{cell}\langle\_\rangle.\,[s]\mathtt{cell}$, with the following machine run to interpret the semantics.

$$\frac{\frac{(\ S\ ;\ \varepsilon_{\mathtt{cell}}\cdot s\ \ ,\ \mathtt{cell}\langle\_\rangle.\,[s']\mathtt{cell}.\,M_{\mathsf{p}}\ )}{(\ S\ ;\ \varepsilon_{\mathtt{cell}}\cdot s'\ ,\ \ M_{\mathsf{p}}\ )}}{(\ S\ ;\ \varepsilon_{\mathtt{cell}}\cdot s''\ ,\ \ T_{\mathsf{p}}\ )}$$

For the *read* construct, we consider the family $\{M_s\}_{s\in S}$ as interpreted by a term M_{p} with a free variable x expecting a value s, so that M_s is interpreted by $\{s/x\}M_{\mathsf{p}}$. Then the interpretation of $\mathtt{read\text{-}cell\text{-}as}\ \{s.M_s\}_{s\in S}$ is given by $\mathtt{cell}\langle x\rangle.\,[x]\mathtt{cell}.\,M_{\mathsf{p}}$, with the following machine run.

$$\frac{\frac{(\ S\ ;\ \varepsilon_{\mathtt{cell}}\cdot s'\ \ ,\ \mathtt{cell}\langle x\rangle.\,[x]\mathtt{cell}.\,M_{\mathsf{p}}\ )}{(\ S\ ;\ \varepsilon_{\mathtt{cell}}\cdot s'\ ,\ \ \{s'/x\}M_{\mathsf{p}}\ )}}{(\ S\ ;\ \varepsilon_{\mathtt{cell}}\cdot s''\ ,\ \ T_{\mathsf{p}}\ )}$$

8 Sequencing and linear continuations

Continuations are a powerful technique to structure control flow in a functional program. For instance, they can be used to encode call–by–value and call–by–name into each other [12]. The sequential λ-calculus is readily encoded into the λ-calculus using linear continuations. A sequential term is encoded by $[\![-]\!]_k$ with reference to a variable k, the *continuation* variable, which replaces the *end* $\star$ of the term. Then *sequencing* $M.\,N$ is interpreted by substituting the continuation variable k of $[\![M]\!]_k$ with $[\![N]\!]_k$.

Definition 30. The *continuation encoding* $[\![-]\!]$ from sequential λ-terms to λ-terms is as follows, where k in $[\![N]\!]_k$ is fresh.

$$\begin{aligned}
[\![\star]\!]_k &= k\\
[\![x.\,N]\!]_k &= x\,[\![N]\!]_k\\
[\![[M].\,N]\!]_k &= [\![N]\!]_k\,[\![M]\!]\\
[\![\langle x\rangle.\,N]\!]_k &= \lambda x.\,[\![N]\!]_k\\
[\![N]\!] &= \lambda k.\,[\![N]\!]_k
\end{aligned}$$

Proposition 31. $[\![M.\,N]\!]_k = \{[\![N]\!]_k/j\}[\![M]\!]_j$.

Proof. By induction on N. □

Proposition 32. *If* $N \longrightarrow M$ *then* $[\![N]\!] \twoheadrightarrow [\![M]\!]$.

Proof. First, it is shown that

$$\{[\![M]\!]/x\}[\![N]\!]_k \twoheadrightarrow [\![\{M/x\}N]\!]_k \ .$$

This follows by induction on N, where the interesting case is $N = x.\, P$, which is given by:

$$\begin{array}{rll} & \{[\![M]\!]/x\}[\![x.P]\!]_k & = \{[\![M]\!]/x\}(x\,[\![P]\!]_k) \\ = & [\![M]\!]\,\{[\![M]\!]/x\}[\![P]\!]_k & \twoheadrightarrow [\![M]\!]\,[\![\{M/x\}P]\!]_k \\ = & (\lambda j.[\![M]\!]_j)\,[\![\{M/x\}P]\!]_k & \longrightarrow \{[\![\{M/x\}P]\!]_k/j\}[\![M]\!]_j \\ = & [\![M.\{M/x\}P]\!]_k & = [\![\{M/x\}x.P]\!]_k \end{array}$$

where the reduction in the second line is by the inductive hypothesis and the penultimate equality is by Proposition 31. The remaining cases are immediate by induction. Then to prove the statement, let $[N].\,\langle x\rangle.\,M \to \{N/x\}M$. The encoding has the required reduction:

$$(\lambda x.\,[\![N]\!]_k)\,[\![M]\!] \longrightarrow \{[\![M]\!]/x\}[\![N]\!]_k \twoheadrightarrow [\![\{M/x\}N]\!]_k \ .$$ □

9 Conclusion

Going forward, there are a few immediate gaps in the basic theory of the FMC, currently under investigation. It is expected that the FMC is semantically captured by *closed Freyd categories*, as described by Power and Thielecke [14], and there is a close relation with Hughes's concept of *arrows* in Haskell [6]. Strong normalization of reduction for the FMC is still open, now only proved for the sequential fragment, which is however less crucial with the current emphasis on the abstract machine. Another salient topic is the complexity of type inference.

Then, at present the calculus is quite limited in its scope. Many topics are waiting to be explored, from basic language features such as coproducts, data types, and type polymorphism, to other computational effects such as exception handling and continuations, to matters of efficient implementation. An interesting next direction would be *concurrency* and the relation with *process calculi*, as the stack operations of the FMC are similar to those for communication along a channel. A hint that this is what is needed is glimpsed in the peculiarities of input and output, which are now modelled as infinite streams, where output appears reversed (the first output is at the bottom of the stack). It appears that viewing an FMC computation as a sequential process, connected by channels to separate, concurrent input and output processes, would give a more accurate model.

Acknowledgment

I would like to thank Alex Simpson, Dominic Hughes, Giulio Guerrieri, Guy McCusker, Jim Laird, John Power, and Ugo Dal Lago for valuable discussions. This work was supported by EPSRC Project EP/R029121/1 *Typed lambda-calculi with sharing and unsharing*.

References

[1] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[2] Alonzo Church. *The calculi of lambda-conversion*. Princeton University Press, 1941.

[3] 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.

[4] 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.

[5] 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.

[6] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.

[7] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20:199–207, 2007.

[8] Paul Blain Levy. *Call-by-push-value: A functional/imperative synthesis*, volume 2 of *Semantic Structures in Computation*. Springer Netherlands, 2003.

[9] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19:377–414, 2006.

[10] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[11] 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.

[12] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[13] Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming (ESOP)*, pages 80–94. Springer, Berlin, Heidelberg, 2009.

[14] A.J. Power and Hayo Thielecke. Closed Freyd- and κ-categories. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1644 of *LNCS*, pages 625–634. Springer, 1999.

[15] W.W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[16] Masako Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 118(1):120–127, 1995.

A The confluence proof

This appendix will give the necessary definitions and lemmata to complete the confluence proof.

Definition 33. A *redex-marked* or *marked* term $N^\blacklozenge$ has a selection of its redexes marked by matching symbols $(\blacklozenge, \blacklozenge)$ as

$$[M]a\blacklozenge.\, A_1 \dots A_n.\blacklozenge a\langle x\rangle.\, N\ .$$

The *marked reduct* $\lfloor N^\blacklozenge \rfloor$ of a marked term $N^\blacklozenge$ is given by

$$\lfloor \star \rfloor = \star \qquad \lfloor [M^\blacklozenge]a.\, N^\blacklozenge \rfloor = [\lfloor M^\blacklozenge \rfloor]a.\, \lfloor N^\blacklozenge \rfloor$$
$$\lfloor x.\, N^\blacklozenge \rfloor = x.\, \lfloor N^\blacklozenge \rfloor \qquad \lfloor a\langle x\rangle.\, N^\blacklozenge \rfloor = a\langle x\rangle.\, \lfloor N^\blacklozenge \rfloor$$

$$\lfloor [M^\blacklozenge]a\blacklozenge.\, A_1^\blacklozenge \dots A_n^\blacklozenge.\blacklozenge a\langle x\rangle.\, P^\blacklozenge \rfloor$$
$$=$$
$$\{\lfloor M^\blacklozenge \rfloor / x\}\lfloor A_1^\blacklozenge \dots A_n^\blacklozenge.\, P^\blacklozenge \rfloor$$

where in the last case x is not free in each A_i, if necessary by α-conversion. A *parallel reduction step* $N \Longrightarrow M$ is a reduction $N \twoheadrightarrow \lfloor N^\blacklozenge \rfloor = M$ for some marking of N.

Observe that a marked reduct is an unmarked term, since all marks have been reduced. Then in the case of the redex:

$$\{\lfloor M^\blacklozenge \rfloor / x\}\lfloor A_1^\blacklozenge \dots A_n^\blacklozenge.\, P^\blacklozenge \rfloor \ = \ \lfloor A_1^\blacklozenge \dots A_n^\blacklozenge.\, \{\lfloor M^\blacklozenge \rfloor / x\}P^\blacklozenge \rfloor$$

Proposition 34. *Parallel reduction is well-defined:* $N \twoheadrightarrow \lfloor N^\blacklozenge \rfloor$.

Proof. By induction on the size of N. The redex case is by the reduction

$$\begin{aligned}
&[M^\blacklozenge]a\blacklozenge.\, A_1^\blacklozenge \dots A_n^\blacklozenge.\blacklozenge a\langle x\rangle.\, P^\blacklozenge \\
&\twoheadrightarrow [\lfloor M^\blacklozenge \rfloor]a\blacklozenge.\, A_1^\blacklozenge \dots A_n^\blacklozenge.\blacklozenge a\langle x\rangle.\, P^\blacklozenge \\
&\longrightarrow A_1^\blacklozenge \dots A_n^\blacklozenge.\, \{\lfloor M^\blacklozenge \rfloor / x\}P^\blacklozenge \\
&= \{\lfloor M^\blacklozenge \rfloor / x\}A_1^\blacklozenge \dots A_n^\blacklozenge.\, P^\blacklozenge \\
&\twoheadrightarrow \{\lfloor M^\blacklozenge \rfloor / x\}\lfloor A_1^\blacklozenge \dots A_n^\blacklozenge.\, P^\blacklozenge \rfloor
\end{aligned}$$

where the equality in line four follows since $x \notin \mathsf{fv}(N)$. □

Lemma 35. *For a doubly marked term,* $N^{\blacklozenge\diamond}$*, reducing each marking in turn gives the same result as reducing both simultaneously:* $\lfloor \lfloor N^\blacklozenge \rfloor^\diamond \rfloor = \lfloor N^{\blacklozenge\diamond} \rfloor$.

Proof. By induction on N. The only non-trivial case is

$$N^{\blacklozenge\diamond} \ = \ [L^{\blacklozenge\diamond}]a\blacklozenge.\, A_1^{\blacklozenge\diamond} \dots A_n^{\blacklozenge\diamond}.\blacklozenge a\langle x\rangle.\, P^{\blacklozenge\diamond}\ .$$

This is resolved as follows.

$$\begin{aligned}
\lfloor\lfloor N^{\blacklozenge}\rfloor^{\diamond}\rfloor &= \lfloor[\lfloor M^{\blacklozenge}\rfloor^{\diamond}]a\!\!\downarrow.\, \lfloor A_1^{\blacklozenge}\ldots A_n^{\blacklozenge}.\, \triangleright a\langle x\rangle.\, P^{\blacklozenge}\rfloor^{\diamond}\rfloor \\
&= \{\lfloor\lfloor M^{\blacklozenge}\rfloor^{\diamond}\rfloor/x\}\lfloor\lfloor A_1^{\blacklozenge}\ldots A_n^{\blacklozenge}.\, P^{\blacklozenge}\rfloor^{\diamond}\rfloor \\
&= \{\lfloor M^{\blacklozenge\diamond}\rfloor/x\}\lfloor A_1^{\blacklozenge\diamond}\ldots A_n^{\blacklozenge\diamond}.\, P^{\blacklozenge\diamond}\rfloor \\
&= \lfloor N^{\blacklozenge\diamond}\rfloor
\end{aligned}$$

The first step is the marked reduction on $(\blacklozenge)$. Observe that in $\lfloor A_1^{\blacklozenge}\ldots A_n^{\blacklozenge}.\, \triangleright a\langle x\rangle.\, P^{\blacklozenge}\rfloor$ only actions A_i are removed, so that the abstraction $\triangleright a\langle x\rangle$ remains part of the marked redex with $[\lfloor M^{\blacklozenge}\rfloor^{\diamond}]a\!\!\downarrow$. The second step is the marked reduction on $(\diamond)$, and the third is by the inductive hypothesis. □

Proposition 36. *Parallel reduction is diamond: if*

$$M \Leftarrow N \Rightarrow P$$

then there is a term Q such that

$$M \Rightarrow Q \Leftarrow P\,.$$

Proof. Let $N^{\blacklozenge} \Rightarrow \lfloor N^{\blacklozenge}\rfloor = M$ and $N^{\diamond} \Rightarrow \lfloor N^{\diamond}\rfloor = P$ for two separate markings $N^{\blacklozenge}$ and $N^{\diamond}$. Let $N^{\blacklozenge\diamond} = N^{\diamond\blacklozenge}$ denote the term with both markings, and let both marked reductions preserve the other marking to get the span

$$M^{\diamond} = \lfloor N^{\blacklozenge}\rfloor^{\diamond} \Leftarrow (N^{\blacklozenge})^{\diamond} = (N^{\diamond})^{\blacklozenge} \Rightarrow \lfloor N^{\diamond}\rfloor^{\blacklozenge} = P^{\blacklozenge}\,.$$

Then let $Q = \lfloor N^{\blacklozenge\diamond}\rfloor = \lfloor N^{\diamond\blacklozenge}\rfloor$ to obtain the desired reductions, using Lemma 35.

$$M^{\diamond} = \lfloor N^{\blacklozenge}\rfloor^{\diamond} \Rightarrow \lfloor\lfloor N^{\blacklozenge}\rfloor^{\diamond}\rfloor = Q = \lfloor\lfloor N^{\diamond}\rfloor^{\blacklozenge}\rfloor \Leftarrow \lfloor N^{\diamond}\rfloor^{\blacklozenge} = P^{\blacklozenge}$$

□

Theorem 8 (Restatement). *Reduction $\longrightarrow$ is confluent.*

Proof. A reduction step $N \longrightarrow M$ is a parallel step $N^{\blacklozenge} \Rightarrow \lfloor N^{\blacklozenge}\rfloor = M$ by marking the redex reduced. A span $M \twoheadleftarrow N \twoheadrightarrow P$ is then one $M \twoheadleftarrow N \Rightarrow P$. By the diamond property, Proposition 36, this converges with parallel reductions, $M \Rightarrow Q \twoheadleftarrow P$, and then by Proposition 34 also with non-parallel reductions, $M \twoheadrightarrow Q \twoheadleftarrow P$. □

B Adequacy of the machine

In this appendix it will be shown how the machine corresponds to reduction. To do so, a combined syntax is developed which embeds machine states into the calculus.

Definition 37. The *readback* $\|S_a\|$ of a stack is defined by $\|\varepsilon_a\| = \star$ and $\|S_a \cdot N\| = [N]a.\, \|S_a\|$. The *readback* $\|S,\, N\|$ of a state (S, N) is the term $\|S\|.\, N$ where $\|S\|$ is the sequence $\|S_a\|$ for every $a \in \mathcal{A}$, prepended to N in no particular order.

Observe that readback is unique modulo $\sim$. To prove the adequacy of the machine, a state (S, N) is embedded into the calculus. The calculus FMC *with states* extends terms with

$$N, M \quad ::= \quad \ldots \quad | \quad |S,\, N|.\, M$$

(where the ellipsis indicates that previous actions are still included). Write $\|N\|$ for a term where every embedded state $|S,\, M|$ is replaced with its readback $\|M,\, S\|$. Reduction $N \longrightarrow M$ with states includes that of the plain calculus, machine transitions on an embedded state, and the following *boundary* transitions on the boundary of an embedded state,

$$\begin{aligned}
|\varepsilon, \star| &\longrightarrow \star \\
|S, x.\, N| &\longrightarrow x.\, |S, N| \\
|S\,;\, S_a \cdot M, N| &\longrightarrow [M]a.\, |S\,;\, S_a, N| \\
|S\,;\, \varepsilon_a, a\langle x\rangle.\, N| &\longrightarrow a\langle x\rangle.\, |S\,;\, \varepsilon_a, N| \\
[M]a.\, N.\, |S\,;\, \varepsilon_a, a\langle x\rangle.\, P| &\longrightarrow N.\, |S\,;\, \varepsilon_a, \{M/x\}P|
\end{aligned}$$

where in the last step N consists of push and pop actions not along a. Observe that the first four steps implement readback by a reduction $N \twoheadrightarrow \|N\|$, so that normal forms of reduction with states are normal forms of the plain calculus.

Proposition 38. *If $N \longrightarrow M$ with embedded states then $\|N\| \longrightarrow \|M\|$ or $\|N\| = \|M\|$ (modulo $\sim$).*

Proof. The case for a plain reduction step is immediate. There are two cases for a step within an embedded state. If

$$\frac{(\ S\ ;\ S_a\ ,\ [M]a.\, N\ )}{(\ S\ ;\ S_a \cdot M\ ,\ N\ )}$$

then

$$\|S\,;\, S_a, [M]a.\, N\| = \|S\|.\, \|S_a\|.\, [M]a.\, N = \|S\,;\, S_a \cdot M, N\|$$

and if

$$\frac{(\ S\ ;\ S_a \cdot M\ ,\ a\langle x\rangle.\, N\ )}{(\ S\ ;\ S_a\ ,\ \{M/x\}N\ )}$$

then

$$\begin{aligned}
\|S\,;\, S_a \cdot M, a\langle x\rangle.\, N\| &= \|S\|.\, \|S_a\|.\, [M]a.\, a\langle x\rangle.\, N \\
&\longrightarrow \|S\|.\, \|S_a\|.\, \{M/x\}N \\
&= \|S\,;\, S_a\, \{M/x\}N\|\ .
\end{aligned}$$

The first four boundary steps are readback steps, for which $N \rightarrow M$ implies $\|N\| = \|M\|$. The last boundary step

$$[M]a.\, N.\, |S\,; \varepsilon_a, a\langle x\rangle.\, P| \;\rightarrow\; N.\, |S\,; \varepsilon_a, \{M/x\}P|$$

gives

$$\begin{aligned} [M]a.\, N.\, \|S\,; \varepsilon_a, a\langle x\rangle.\, P\| &= [M]a.\, N.\, \|S\|.\, a\langle x\rangle.\, P \\ &\rightarrow N.\, \|S\|.\, \{M/x\}P \\ &= N.\, \|S\,; \varepsilon_a, \{M/x\}P\| \end{aligned}$$

where $\|S\|$ consists only of push actions not along a. □

In the other direction:

Proposition 39. *If $N \rightarrow M$ then*

$$\frac{(\ S\ ,\ N\ )}{(\ T\ ,\ \star\ )} \qquad \textit{implies} \qquad \frac{(\ S\ ,\ M\ )}{(\ T\ ,\ \star\ )}\,.$$

Proof. Let

$$[M]a.\, A_1 \ldots A_n.\, a\langle x\rangle.\, N \;\rightarrow\; A_1 \ldots A_n.\, \{M/x\}N\,.$$

The machine run for the left-hand side looks as follows, since each A_i is an action not on a.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\ R\,;\ S_a\ ,\ [M]a.\, A_1 \ldots A_n.\, a\langle x\rangle.\, N\ )}{(\ R\,;\ S_a \cdot M\ ,\ A_1 \ldots A_n.\, a\langle x\rangle.\, N\ )}}{(\ S\,;\ S_a \cdot M\ ,\ a\langle x\rangle.\, N'\ )}}{(\ S\,;\ S_a\ ,\ \{M/x\}.\, N'\ )}}{(\ T\ ,\ \star\ )}$$

Then for the right-hand side there is the following corresponding run, since no abstraction in A_i binds in M.

$$\dfrac{\dfrac{(\ R\,;\ S_a\ ,\ A_1 \ldots A_n.\, \{M/x\}N\ )}{(\ S\,;\ S_a\ ,\ \{M/x\}.\, N'\ )}}{(\ T\ ,\ \star\ )}$$

□

C SN for the sequential λ-calculus

To show their strong normalization, typed sequential λ-terms will be encoded in second-order λ-calculus ($\Lambda 2$). For sequential types, the sequent calculus of Figure **??** is used, and a sequent-calculus type system for $\Lambda 2$ is given in Figure 4. Here, the notation $\Gamma \cup x\colon \tau$ allows $x\colon \tau \in \Gamma$ (whereas $\Gamma, x\colon \tau$ implies x is

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\,\mathsf{ax}$$

$$\frac{\Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash \lambda x^{\sigma}.N : \sigma \to \tau}\,{\to}\mathsf{R}$$

$$\frac{\Gamma \vdash M : \rho \qquad \Gamma, y : \sigma \vdash N : \tau}{\Gamma \cup x : \rho \to \sigma \vdash \{xM/y\}N : \tau}\,{\to}\mathsf{L}$$

$$\frac{\Gamma \vdash N : \tau}{\Gamma \vdash \Lambda a.N : \forall \alpha.\, \tau}\,\forall\mathsf{R}\ (\alpha \notin \Gamma)$$

$$\frac{\Gamma, y : \{\sigma/\alpha\}\rho \vdash N : \tau}{\Gamma \cup x : \forall \alpha.\, \rho \vdash \{x\,\sigma/y\}N : \tau}\,\forall\mathsf{L}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash \{M/x\}N : \tau}\,\mathsf{cut}$$

Figure 4: A sequent calculus for second-order λ-calculus

not in the domain of Γ). The encoding is the typed version of the *continuation encoding* of Definition 30. To preserve this literally, apart from the definition in Figure 4, the interpretation will omit the type abstraction $\Lambda\alpha.N$ and application $N\,\tau$ from $\Lambda 2$-terms, as well as the type annotation τ on abstractions, λx^{τ}. These can be reconstructed from the type derivations. A sequential type will be encoded into second-order polymorphic types as follows.

$$[\![\sigma_n \dots \sigma_1 \Rightarrow \tau_1 \dots \tau_m]\!] \\ = \\ \forall \alpha.\, ([\![\tau_m]\!] \to \dots [\![\tau_1]\!] \to \alpha) \to [\![\sigma_n]\!] \to \dots [\![\sigma_1]\!] \to \alpha$$

Definition 40. The *continuation encoding* $[\![\tau]\!]$ and *context encoding* $[\![\tau]\!]_\alpha$ for a type variable α, from sequential types to second-order types, are given by

$$\begin{aligned} [\![\tau]\!] &= \forall \alpha.\, [\![\tau]\!]_\alpha \\ [\![\overleftarrow{\sigma} \Rightarrow \vec{\tau}\,]\!]_\alpha &= [\![\vec{\tau}]\!]_\alpha \to [\![\overleftarrow{\sigma}]\!]_\alpha \\ [\![\tau_1 \dots \tau_n]\!]_\alpha &= [\![\tau_n]\!] \to \dots [\![\tau_1]\!] \to \alpha\,. \end{aligned}$$

These translations are extended to contexts Γ and sequents as follows.

$$\begin{aligned} [\![x_1 : \tau_1, \dots, x_n : \tau_n]\!] &= x_1 : [\![\tau_1]\!], \dots, x_n : [\![\tau_n]\!] \\ [\![\Gamma \vdash N : \tau\,]\!] &= [\![\Gamma]\!] \vdash [\![N]\!] : [\![\tau]\!] \\ [\![\Gamma \vdash N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}\,]\!]_{\alpha,k} &= [\![\Gamma]\!], k : [\![\vec{\tau}]\!]_\alpha \vdash [\![N]\!]_k : [\![\overleftarrow{\sigma}]\!]_\alpha \end{aligned}$$

The translations $[\![-]\!]_{\alpha,k}$ and $[\![-]\!]$ on a given sequent are related by a *closing* derivation cls, consisting of a ${\to}\mathsf{R}$ rule closing k and a $\forall\mathsf{R}$ rule closing α:

$$\frac{[\![\Gamma, N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}\,]\!]_{\alpha,k}}{[\![\Gamma, N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}\,]\!]}\,\mathsf{cls} \;:=\; \frac{\dfrac{[\![\Gamma]\!], k : [\![\vec{\tau}]\!]_\alpha \vdash [\![N]\!]_k : [\![\overleftarrow{\sigma}]\!]_\alpha}{[\![\Gamma]\!] \vdash \lambda k.[\![N]\!]_k : [\![\vec{\tau}]\!]_\alpha \to [\![\overleftarrow{\sigma}]\!]_\alpha}\,{\to}\mathsf{R}}{[\![\Gamma]\!] \vdash \lambda k.[\![N]\!]_k : \forall \alpha.\, [\![\vec{\tau}]\!]_\alpha \to [\![\overleftarrow{\sigma}]\!]_\alpha}\,\forall\mathsf{R}$$

$$[\![\star]\!]_{\alpha,k} \;=\; \frac{}{[\![\Gamma]\!], k\colon [\![\overleftarrow{\tau}]\!]_\alpha \vdash k\colon [\![\overleftarrow{\tau}]\!]_\alpha}\,\mathsf{ax}$$

$$[\![\mathsf{var}]\!]_{\alpha,k} \;=\; \frac{\dfrac{\dfrac{}{[\![\Gamma]\!], k\colon [\![\overleftarrow{\tau}\,\overleftarrow{\sigma}]\!]_\alpha \vdash k\colon [\![\overleftarrow{\tau}\,\overleftarrow{\sigma}]\!]_\alpha}\,\mathsf{ax} \qquad \dfrac{}{[\![\Gamma]\!], j\colon [\![\overleftarrow{\rho}\,\overleftarrow{\sigma}]\!]_\alpha \vdash j\colon [\![\overleftarrow{\rho}\,\overleftarrow{\sigma}]\!]_\alpha}\,\mathsf{ax}}{[\![\Gamma]\!], x\colon [\![\overleftarrow{\tau}\,\overleftarrow{\sigma}]\!]_\alpha \to [\![\overleftarrow{\rho}\,\overleftarrow{\sigma}]\!]_\alpha, k\colon [\![\overleftarrow{\tau}\,\overleftarrow{\sigma}]\!]_\alpha \vdash x\,k\colon [\![\overleftarrow{\rho}\,\overleftarrow{\sigma}]\!]_\alpha}\,{\to}\mathsf{L}}{[\![\Gamma]\!], x\colon \forall\beta.\,[\![\overleftarrow{\tau}]\!]_\beta \to [\![\overleftarrow{\rho}]\!]_\beta, k\colon [\![\overleftarrow{\tau}\,\overleftarrow{\sigma}]\!]_\alpha \vdash x\,k\colon [\![\overleftarrow{\rho}\,\overleftarrow{\sigma}]\!]_\alpha}\,\forall\mathsf{L}$$

$$[\![\mathsf{app}]\!]_{\alpha,k} \;=\; \frac{[\![\Gamma]\!] \vdash [\![M]\!]\colon [\![\tau]\!] \qquad [\![\Gamma]\!], j\colon [\![\overleftarrow{\sigma}]\!]_\alpha \vdash [\![N]\!]_j\colon [\![\overleftarrow{\rho}]\!]_\alpha}{[\![\Gamma]\!], k\colon [\![\tau]\!] \to [\![\overleftarrow{\sigma}]\!]_\alpha \vdash \{k\,[\![M]\!]/j\}[\![N]\!]_j\colon [\![\overleftarrow{\rho}]\!]_\alpha}\,{\to}\mathsf{L}$$

$$[\![\mathsf{abs}]\!]_{\alpha,k} \;=\; \frac{[\![\Gamma]\!], k\colon [\![\overleftarrow{\tau}]\!]_\alpha, x\colon [\![\rho]\!] \vdash [\![N]\!]_k\colon [\![\overleftarrow{\sigma}]\!]_\alpha}{[\![\Gamma]\!], k\colon [\![\overleftarrow{\tau}]\!]_\alpha \vdash \lambda x.[\![N]\!]_k\colon [\![\rho]\!] \to [\![\overleftarrow{\sigma}]\!]_\alpha}\,{\to}\mathsf{R}$$

$$[\![\mathsf{cut}]\!]_{\alpha,k} \;=\; \frac{[\![\Gamma]\!], k\colon [\![\overleftarrow{\tau}]\!]_\alpha \vdash [\![M]\!]_k\colon [\![\overleftarrow{\sigma}]\!]_\alpha \qquad [\![\Gamma]\!], x\colon [\![\overleftarrow{\sigma}]\!]_\alpha \vdash [\![N]\!]_x\colon [\![\overleftarrow{\rho}]\!]_\alpha}{[\![\Gamma]\!], k\colon [\![\overleftarrow{\tau}]\!]_\alpha \vdash \{[\![M]\!]_k/x\}[\![N]\!]_x\colon [\![\overleftarrow{\rho}]\!]_\alpha}\,\mathsf{cut}$$

Figure 5: Context interpretation of typing rules

Proposition 41 (Interpretation preserves types)**.** *If* $\Gamma \vdash N\colon \tau$ *then* $[\![\Gamma \vdash N\colon \tau]\!]$.

Proof. By the derivations in Figure 5, which translate the typing rules of the sequential λ-calculus of Figure 2 into second-order λ-calculus. □

Proposition 42 (Reduction commutes with typed interpretations)**.** *If* $\Gamma \vdash N\colon \tau$ *and* $N \longrightarrow M$ *then* $[\![\Gamma \vdash N\colon \tau]\!] \twoheadrightarrow [\![\Gamma \vdash M\colon \tau]\!]$.

Proof. First, to show that the reduction $\{[\![M]\!]/x\}[\![N]\!]_k \twoheadrightarrow [\![\{M/x\}N]\!]_k$ preserves types, the following must be shown.

$$\frac{[\![\Gamma]\!] \vdash [\![M]\!]\colon [\![\tau]\!] \qquad [\![\Gamma]\!], k\colon [\![\overleftarrow{v}]\!]_\alpha, x\colon [\![\tau]\!] \vdash [\![N]\!]_k\colon [\![\overleftarrow{\sigma}]\!]_\alpha}{[\![\Gamma]\!], k\colon [\![\overleftarrow{v}]\!]_\alpha \vdash \{[\![M]\!]/x\}[\![N]\!]_k\colon [\![\overleftarrow{\sigma}]\!]_\alpha}\,\mathsf{cut}$$

$$\twoheadrightarrow$$

$$[\![\Gamma]\!], k\colon [\![\overleftarrow{v}]\!]_\alpha \vdash [\![\{M/x\}N]\!]_k\colon [\![\overleftarrow{\sigma}]\!]_\alpha$$

This is proved by induction on N. The base case $N = x$ is given in Figure 6 as the top reduction; the remaining cases are by permutation steps.

Then to prove the proposition, consider the typed version of the reduction step $N.\,[M].\,\langle x\rangle.\,P \longrightarrow N.\,\{M/x\}P$, below.

$$\frac{\Gamma \vdash M\colon \rho \qquad \dfrac{\Gamma \vdash N\colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}}{\Gamma \vdash \langle x\rangle.\,N\colon \rho\,\overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}}}{\Gamma \vdash [M].\,\langle x\rangle.\,N\colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}}$$

$$\longrightarrow$$

$$\Gamma \vdash \{M/x\}N\colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}$$

□

The continuation interpretation of this reduction sequence is given in Figure 6 as the bottom reduction, using the typed reduction $\{[\![M]\!]/x\}[\![N]\!]_k \twoheadrightarrow [\![\{M/x\}N]\!]_k$ in the second step.

Theorem 43. *Typed sequential λ-terms are strongly normalizing.*

Proof. If $\Gamma \vdash N : \tau$ had an infinite reduction, then by Proposition 42, so would $[\![\Gamma]\!] \vdash [\![N]\!] : [\![\tau]\!]$. But this is a contradiction, since second-order λ-calculus is strongly normalizing. □

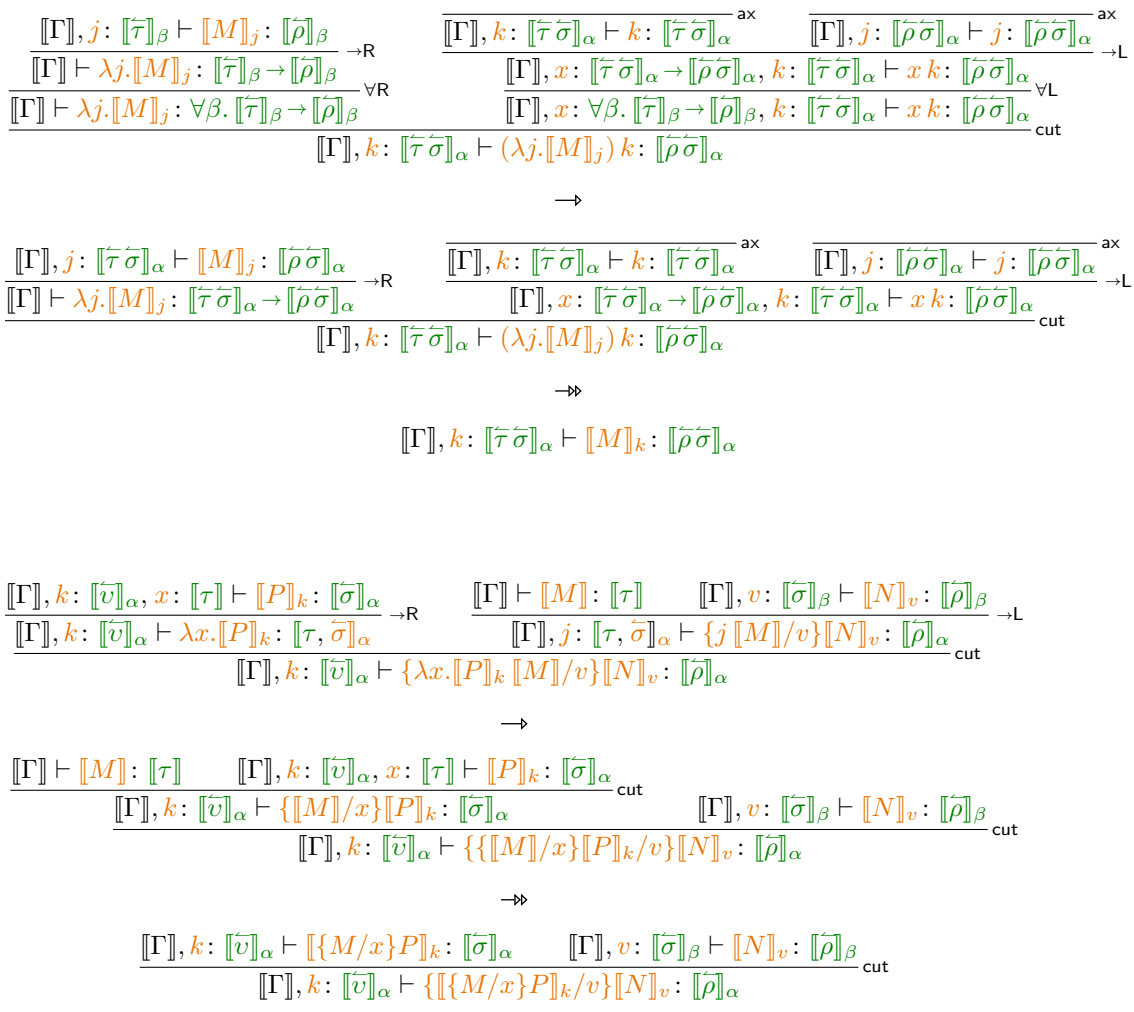

Figure 6: Reductions for the proof of Proposition 42