The Functional Machine Calculus II: Semantics

Chris Barrett
Department of Computer Science, University of Bath, UK

Willem Heijltjes
Department of Computer Science, University of Bath, UK

Guy McCusker
Department of Computer Science, University of Bath, UK

Abstract

The Functional Machine Calculus (FMC), recently introduced by the second author, is a generalization of the lambda-calculus which may faithfully encode the effects of higher-order mutable store, I/O and probabilistic/non-deterministic input. Significantly, it remains confluent and can be simply typed in the presence of these effects.

In this paper, we explore the denotational semantics of the FMC. We have three main contributions: first, we argue that its syntax – in which both effects and lambda-calculus are realised using the same syntactic constructs – is semantically natural, corresponding closely to the structure of a Scott-style domain theoretic semantics. Second, we show that simple types confer strong normalization by extending Gandy's proof for the lambda-calculus, including a small simplification of the technique. Finally, we show that the typed FMC (without considering the specifics of encoded effects), modulo an appropriate equational theory, is a complete language for Cartesian closed categories.

2012 ACM Subject Classification Theory of computation

Keywords and phrases lambda-calculus, computational effects, denotational semantics, strong normalization

Digital Object Identifier 10.4230/LIPIcs.CSL.2023.10

Related Version *Full Version*: `https://arxiv.org/abs/2211.13140`

Funding *Willem Heijltjes*: EPSRC Grant EP/R029121/1 *Typed lambda-calculi with sharing and unsharing.*

Acknowledgements Pierre Clairambault, Ugo Dal Lago, Anupam Das, Giulio Guerrieri, Jim Laird, Paul Levy, Simon Peyton Jones, John Power, Alex Simpson, Vincent van Oostrom.

1 Introduction

Almost without exception, modern programming languages support a combination of computational effects and higher-order mechanisms. Programmers and programming language theorists recognise that the effects and higher-order mechanisms are fundamentally different constructs, with radically different syntax and semantics: compare assignment and dereferencing operations with function definition and invocation, for example. In both operational and denotational accounts, the higher-order mechanism – typically expressed in some variant of λ-calculus – and the effects mechanisms are treated using distinct approaches, and indeed the combination of the two, not to mention the combination of multiple kinds of effects, requires careful handling.

In a previous paper [6] the second author introduced the Functional Machine Calculus (FMC), a compact programming language which eliminates these distinctions and supports higher-order effectful programming with a streamlined yet natural syntax and operational semantics. In this paper, we reprise the definition of the FMC and attempt to explain and explore its construction and behaviour from a denotational perspective. Beginning with a domain-theoretic analysis of computation with stacks, we discover that the λ-calculus and effectful aspects of programming can be viewed as being of exactly the same kind, in fact entirely interchangeable. The syntax and operational semantics of the FMC embodies the operations naturally supported by the denotational semantics, while remaining programmer-friendly: the previous paper [6] demonstrates by example the wide range of effectful programs that can straightforwardly be expressed in this syntax.

The connection we exploit between domain-theoretic semantics of λ-calculus and an operational semantics based on stacks is not new. In [19], Streicher and Reus show that Scott's well-known D_∞ models motivate and explain the definition of Krivine's abstract machine for evaluating λ-terms. Scott's construction takes a domain D and builds a model of the λ-calculus as an appropriate limit of a sequence of domains, shown on the left below. Streicher and Reus observed that this construction may alternatively be regarded as taking the limit of the sequence on the right, where each D_n is recovered as $C_n \to D$, and D_∞ is $C_\infty \to D$.

$$\begin{array}{rcl} D_0 &=& D \\ D_{n+1} &=& D_n \to D_n \end{array} \qquad\qquad \begin{array}{rcl} C_0 &=& 1 \\ C_{n+1} &=& C_n \times (C_n \to D) \end{array}$$

With this view, the limit C_∞ is a stream or (potentially infinite) stack of elements of D_∞, i.e. denotations of λ-terms; and such terms consume a stack. The equations defining the interpretations of terms in this model show that the operations of application and abstraction correspond directly to pushing and popping from the stack. The return domain D plays a very minor role in the semantics of λ-terms: in Streicher and Reus's view, it is the result type of continuations, and ordinary λ-terms never return. Our development in this paper adapts this in two ways. First, we choose the return domain D to be the domain of stacks. Thus a term can be regarded as a *stack transformer*, which immediately supports an operation of sequencing between terms, leading to a sublanguage of the FMC called the *sequential λ-calculus*. Second, we enrich the domain equation with the familiar monad for global state, and observe that – if states are also stacks – the resulting domain equation is one of multiple-stack transformers, with *no special status* afforded to the stack that implements λ-abstraction and application. Thus we arrive at the promised semantic explanation of the FMC, treating reader–writer style effects and higher-order mechanisms identically, and giving a denotational motivation for the stack-machine semantics.

We go on to study this calculus from a type-theoretic and category-theoretic perspective. The domain-theoretic model is untyped but, generalising the usual development for λ-calculus, a simple type system can be imposed which describes the shapes of the stacks being operated upon. Adapting Gandy's proof technique for λ-calculus, we show that the well-typed terms of the FMC are strongly normalizing. Further, we study the categorical properties of the calculus, viewing terms as morphisms between types, and discover that, up to a notion of contextual equivalence, well-typed programs form a Cartesian closed category. An equational theory is presented which refines this equivalence, and well-typed terms modulo this theory are shown to be a sound and complete language for CCCs.

The properties identified above – strong normalization, Cartesian closure, and the operational property of confluence established in [6] – are entirely expected of languages which are variants of typed λ-calculus, but perhaps surprising, or even shocking, in the setting of the effectful FMC. How can the combination of higher-order programming and effects, including state, input/output, nondeterminism and probability, retain properties of confluence and referential transparency?

We offer the following explanation. These properties are of the FMC as a general calculus, which remains close to the λ-calculus, independent of the encoding of effects. As explored in [6], the FMC is confluent, and for encoded reader/writer effects this manifests as reduction following the algebraic laws of Plotkin and Power [13]. The CCC semantics presented in this paper pertains to the FMC itself, and, we emphasize, is *not* a semantics of effects: when particular properties of the encoded effects are taken into account, the semantics will no longer be a CCC. We explore this in more detail in Section 6.

2 The Functional Machine Calculus

The Functional Machine Calculus (FMC) arises from an operational perspective on the λ-calculus, taking a simple call–by–name stack machine in the style of Krivine [7] as primary. The machine evaluates a term in the context of a stack, where application is a *push* (of the argument) and abstraction is a *pop*, binding the popped value to the abstracted variable. Since the stack machine is intended as an operational semantics, and not for implementation, for simplicity we use substitution rather than an environment. The FMC then introduces two natural generalizations.

Locations. The machine is generalized from one to multiple stacks or streams, indexed by a global set of *locations* A. In the calculus, abstraction and application are parameterized in A as pop- and push-actions on the associated stack. Stacks are homogeneous, but may be used to encode different reader/writer effects: an input stream (which may be non-deterministic or probabilistic), an output stream, or a global mutable variable.

Sequencing. The calculus is extended with sequential composition of terms, which gives their consecutive execution on the machine, and an identity term as the unit to composition, which is the empty instruction sequence on the machine, analogous to imperative *skip*. This generalizes the calculus from one of stack *consumers* to stack *transformers*, where a term may return multiple outputs to the stack, and gives control over execution, as demonstrated by the encoding of various reduction strategies including call–by–value and call–by–push–value (see [6]).

Locations are an innovation of the FMC [6], while sequencing is familiar from Hasegawa's κ-calculus [5, 17] and higher-order stack programming (see e.g. [10]). In the latter case, there are also certain similarities with Milner's action calculi [9]. These two innovations are implemented technically as follows. To emphasize the operational intuition as *push* and *pop*, application $M\,N$ will be written as $[N].\,M$, and abstraction $\lambda x.\,M$ as $\langle x\rangle.\,M$. These are subsequently parameterized in *locations* $a, b, c \in A$. *Sequencing* introduces a *nil* or *skip* term $\star$ and makes the variable construct a prefix $x.\,M$; sequential composition $M\,;N$ will be a defined operation.

▶ **Definition 1.** *The* Functional Machine Calculus *(*FMC*) is given by the grammar*

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

where from left to right the term *constructors are* 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 M. Terms are considered modulo α-equivalence.*

We may omit the trailing $.\,\star$ from terms for readability. We will use a *main* location λ, omitted from the term notation, as the computation stack (as opposed to the stacks to interpret effects), on which (call–by–name) λ-terms embed. We will use constants as free variables; constant operators such as addition $+$ and conditionals if will operate on λ as well.

▶ **Example 2.** Consider the following example terms.

$$\mathsf{rnd}\langle x\rangle.\,[x].\,c\langle y\rangle.\,[y].\,+\,.\,\langle z\rangle.\,[z]c \qquad\qquad [\langle x\rangle.\,[x]\mathsf{out}.\,[x].\,[1].\,+].\,\langle f\rangle.\,[0].\,f.\,f.\,f$$

The first term increments a memory cell c with a random number. It pops x from the random generator stream rnd and pushes it to the main stack; pops y from the cell c and pushes that to the main stack as well; then $+$ adds the top two elements of the main stack x and y pushing back the result $x+y$; and this is popped as z and pushed back onto the cell c.

The second term counts up from zero to three, outputting $0,1,2$ and leaving 3 on the main stack. The subterm $\langle x\rangle.\,[x]\mathsf{out}.\,[x].\,[1].\,+$ pops x from the main stack and sends it to the output location out, and then $[x].\,[1].\,+$ leaves $x+1$ on the main stack. In the example, this term is pushed, popped as f, and called three times on zero.

▶ **Definition 3.** Composition $N\,;M$ *and* substitution $\{N/x\}M$ *are given by*

$$\begin{array}{rclrcll}
\star\,;M &=& M & [P]a.\,N\,;M &=& [P]a.\,(N\,;M) & \\
x.\,N\,;M &=& x.\,(N\,;M) & a\langle x\rangle.\,N\,;M &=& a\langle x\rangle.\,(N\,;M) & (x\notin \mathsf{fv}(M))
\end{array}$$

$$\begin{array}{rclrcll}
\{P/x\}\star &=& \star & \{P/x\}[N]a.\,M &=& [\{P/x\}N]a.\,\{P/x\}M & \\
\{P/x\}x.\,M &=& P\,;\{P/x\}M & \{P/x\}a\langle x\rangle.\,M &=& a\langle x\rangle.\,M & \\
\{P/x\}y.\,M &=& y.\,\{P/x\}M & \{P/x\}a\langle y\rangle.\,M &=& a\langle y\rangle.\,\{P/x\}M & (y\notin \mathsf{fv}(P))
\end{array}$$

where, in the last two cases, $x\neq y$*; both are capture-avoiding by the conditions* $x\notin\mathsf{fv}(M)$ *and* $y\notin\mathsf{fv}(P)$*, which can be satisfied by* α*-conversion.*

▶ **Lemma 4.** *Composition is associative and has unit* $\star$*.*

▶ **Definition 5.** *The* functional abstract machine *is given by the following data. A* stack *of terms S is defined below left, and written with the top element to the right. A* memory S_A *is a family of stacks or streams in A, defined below right.*

$$S \;::=\; \varepsilon \;\mid\; S\cdot M \qquad\quad S_A \;::=\; \{\,S_a \mid a\in A\}$$

The notation $S_A;S_a$ identifies the stack S_a within S_A. A state *is a pair (S_A, M) of a memory and a term. The* transitions *or* steps *of the machine are given below left as top–to–bottom rules. A* run *of the machine is a sequence of steps, written as $(S_A, M)\Downarrow(T_A, N)$ or with a double line as below right.*

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

Constant operations such as addition $+$ and conditional if pop the required number of items from the main stack and reinstate their result, as per the rule given below left. The FMC then operates as a standard stack calculus: e.g. an arithmetic expression $1+((2+3)\times 4)$ is given as a term $[4].\,[3].\,[2].\,+\,.\,\times\,.\,[1].\,+$ which indeed returns 21. Formally, a constant operator $c^{n,m}$ of arity n,m is defined by a (partial) function $c^{n,m}$ from n input terms to m output terms, which generates the machine rule schema below right, where the output terms are put on the stack.

$$\frac{(\;S_A;S_\lambda\cdot 2\cdot 3\;,\;+\,.\,M\;)}{(\;S_A;S_\lambda\cdot 5\quad\;,\quad M\;)}\qquad \frac{(\;S_A;S_\lambda\cdot N_n\cdots N_1\qquad\quad,\;c^{n,m}.\,M\;)}{(\;S_A;S_\lambda\cdot c^{n,m}(N_1,\ldots,N_n)\;,\quad M\;)}$$

▶ **Example 6.** The first term of Example 2 has the following run of the machine, where the rnd location is initialized with a stream with 3 at the head, and c with the value 5.

$$\begin{array}{lr}
(\ S_{\mathsf{rnd}}\cdot 3\ ;\ \varepsilon_c\cdot 5\ ;\ \varepsilon_\lambda\ , & \mathsf{rnd}\langle x\rangle.\,[x].\,c\langle y\rangle.\,[y].\,+\,.\,\langle z\rangle.\,[z]c\ ) \\ \hline
(\ S_{\mathsf{rnd}}\ ;\ \varepsilon_c\cdot 5\ ;\ \varepsilon_\lambda\ , & [3].\,c\langle y\rangle.\,[y].\,+\,.\,\langle z\rangle.\,[z]c\ ) \\ \hline
(\ S_{\mathsf{rnd}}\ ;\ \varepsilon_c\cdot 5\ ;\ \varepsilon_\lambda\cdot 3\ , & c\langle y\rangle.\,[y].\,+\,.\,\langle z\rangle.\,[z]c\ ) \\ \hline
(\ S_{\mathsf{rnd}}\ ;\ \varepsilon_c\ ;\ \varepsilon_\lambda\cdot 3\ , & [5].\,+\,.\,\langle z\rangle.\,[z]c\ ) \\ \hline
(\ S_{\mathsf{rnd}}\ ;\ \varepsilon_c\ ;\ \varepsilon_\lambda\cdot 3\cdot 5\ , & +\,.\,\langle z\rangle.\,[z]c\ ) \\ \hline
(\ S_{\mathsf{rnd}}\ ;\ \varepsilon_c\ ;\ \varepsilon_\lambda\cdot 8\ , & \langle z\rangle.\,[z]c\ ) \\ \hline
(\ S_{\mathsf{rnd}}\ ;\ \varepsilon_c\ ;\ \varepsilon_\lambda\ , & [8]c\ ) \\ \hline
(\ S_{\mathsf{rnd}}\ ;\ \varepsilon_c\cdot 8\ ;\ \varepsilon_\lambda\ , & \star\ )
\end{array}$$

Beta-reduction in the λ-calculus, from the perspective of the machine, lets consecutive push and pop actions interact directly. With multiple stacks, these must be actions on the same location, while other locations may be accessed in-between. Informally, the reduction step will then be as follows, where the argument $[N]a$ and abstraction $a\langle x\rangle$ may be separated by actions $[P]b$ and $b\langle y\rangle$ with $a \neq b$: $[N]a \dots a\langle x\rangle.\,M \ \rightarrow_\beta\ \dots \{N/x\}M$. In addition, it must be avoided that any intervening $b\langle y\rangle$ captures y in N.

▶ **Definition 7.** *A* head context H *is a sequence of applications and abstractions terminating in a hole:*

$$H \ ::=\ \{\} \ \mid\ [M]a.\,H \ \mid\ a\langle x\rangle.\,H$$

The term denoted $H.\,M$ *is given by replacing the hole* $\{\}$ *in* H *with* M*, where a binder* $a\langle x\rangle$ *in* H *captures in* M*. The* binding variables $\mathsf{bv}(H)$ *of* H *are those variables* x *where* H *is constructed over* $a\langle x\rangle$*. The set of* locations *used in a term or context is denoted* $\mathsf{loc}(M)$ *respectively* $\mathsf{loc}(H)$*. Then* beta-reduction *and* eta-reduction *are defined respectively by the rewrite rule schemas below, where* $a \notin \mathsf{loc}(H)$ *for both reduction rules,* $\mathsf{bv}(H) \cap \mathsf{fv}(N) = \varnothing$ *for the* β*-rule, and* $x \notin \mathsf{bv}(H)$ *for the* η*-rule. Both reductions are closed under all contexts.*

$$[N]a.\,H.\,a\langle x\rangle.\,M \ \rightarrow_\beta\ H.\,\{N/x\}M \qquad\qquad a\langle x\rangle.\,H.\,[x]a.\,M \ \rightarrow_\eta\ H.\,M \quad (x \notin \mathsf{fv}(M))$$

We now clarify the relationship between beta reduction and the machine. Evaluation of a term M on the machine, given sufficient inputs in the form of a stack of terms $N_1 \cdots N_n$, begins in the state $(N_1 \cdots N_n, M)$. The machine then implements a particular (weak) reduction strategy, with each *pop* transition corresponding to a beta-reduction of the term $[N_1] \dots [N_n].\,M$ corresponding to the machine state under evaluation.

In Section 6, we further introduce a notion of observational equivalence based on the machine, dubbed *machine equivalence*, where terms are considered equivalent if they send equivalent inputs to equivalent outputs. This is shown to validate the beta and eta equations in general.

The two generalizations *locations* and *sequencing* are independent, and the two calculi that have one but not the other are of independent interest.

- The *poly λ-calculus* has only *locations*, and is given by the fragment below.

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

- The *sequential λ-calculus* has only *sequencing*, and is given by the fragment below.

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

▶ **Example 8.** To demonstrate how the FMC encodes both effects and evaluation strategies, we consider the following (standard) call–by–value λ-calculus with reader/writer effects. (We assume familiarity with the operational semantics of effects and call–by–value λ-calculus; for an introduction see Winskel [20].)

$$\begin{array}{rlll}
M,N,P ::= & x \;\mid\; M\,N \;\mid\; \lambda x.M & & \textit{λ-calculus}\\
\mid & \mathsf{read} \;\mid\; \mathsf{write}\ N;M & & \textit{input/output}\\
\mid & c := N;M \;\mid\; !c & & \textit{state update and lookup}\\
\mid & N \oplus M \;\mid\; N + M & & \textit{probabilistic and non-deterministic sum}
\end{array}$$

The full calculus is encoded into the FMC as follows. We let A comprise the main location λ, a location in for input, out for output, rnd and nd for probabilistically and non-deterministically generated streams of boolean values $(\top,\bot)$, and one location for each global memory cell. A term M encodes as M_v below, where $N+M$ encodes like $N \oplus M$ but with the stream nd.

$$\begin{array}{rlcrlcrl}
x_v &= [x] & & & & & (\mathsf{write}\ N\,;M)_v &= N_v.\,\langle x\rangle.\,[x]\mathsf{out}.\,M_v\\
(\lambda x.\,M)_v &= [\langle x\rangle.\,M_v] & & \mathsf{read}_v &= \mathsf{in}\langle x\rangle.\,[x] & & (c := N\,;M)_v &= N_v.\,\langle x\rangle.\,c\langle\_\rangle.\,[x]c.\,M_v\\
(M\,N)_v &= N_v\,;M_v\,;\langle x\rangle.\,x & & !c_v &= c\langle x\rangle.\,[x]c.\,[c] & & (N\oplus M)_v &= \mathsf{rnd}\langle x\rangle.\,[N].\,[M].\,[x].\,\mathsf{if}
\end{array}$$

We leave it to the reader to confirm that the interpretation generates the correct evaluation behaviour, and conclude the example with the encoding and reduction of the following term.

$$\begin{array}{rcl}
((\lambda f.f\,(f\,0))\,(\lambda x.\mathsf{write}\ x;!c))_v & = & [\langle x\rangle.\,\underline{[x].\,\langle v\rangle}.\,[v]\mathsf{out}.\,c\langle y\rangle.\,[y]c.\,[y]].\,\langle f\rangle.\,[0].\,[f].\,\langle z\rangle.\,z.\,[f].\,\langle w\rangle.\,w\\
& \rightharpoonup & [\langle x\rangle.\,[x]\mathsf{out}.\,c\langle y\rangle.\,[y]c.\,[y]].\,\langle f\rangle.\,[0].\,[f].\,\langle z\rangle.\,z.\,\underline{[f].\,\langle w\rangle}.\,w\\
& \rightharpoonup & [\langle x\rangle.\,[x]\mathsf{out}.\,c\langle y\rangle.\,[y]c.\,[y]].\,\langle f\rangle.\,[0].\,\underline{[f].\,\langle z\rangle}.\,z.\,f\\
& \rightharpoonup & \underline{[\langle x\rangle.\,[x]\mathsf{out}.\,c\langle y\rangle.\,[y]c.\,[y]].\,\langle f\rangle}.\,[0].\,f.\,f\\
& \rightharpoonup & [0].\,\langle x\rangle.\,[x]\mathsf{out}.\,c\langle y\rangle.\,\underline{[y]c}.\,[y].\,\langle z\rangle.\,[z]\mathsf{out}.\,\underline{c\langle w\rangle}.\,[w]c.\,[w]\\
& \rightharpoonup & [0].\,\langle x\rangle.\,[x]\mathsf{out}.\,c\langle y\rangle.\,\underline{[y].\,\langle z\rangle}.\,[z]\mathsf{out}.\,[y]c.\,[y]\\
& \rightharpoonup & \underline{[0].\,\langle x\rangle}.\,[x]\mathsf{out}.\,c\langle y\rangle.\,[y]\mathsf{out}.\,[y]c.\,[y]\\
& \rightharpoonup & [0]\mathsf{out}.\,c\langle y\rangle.\,[y]\mathsf{out}.\,[y]c.\,[y]
\end{array}$$

3 Domain-theoretic semantics

Our aim in this section is to show that the syntax and stack-machine semantics of the FMC may be further justified by consideration of a simple domain-theoretic semantics. We work in the category **Cppo** of complete partial orders (with least-element) and continuous functions. We show that a domain equation for stack-transformers directly supports the operations of the sequential λ-calculus, directly extending a Scott-style semantics of λ-calculus. The step from sequential lambda-calculus to FMC is then nothing more than the incorporation of the state monad in the original domain equation. Our development has much in common with Streicher and Reus's work [19]; the key step in the move to the FMC is to allow computations to return a result – a new stack – which may be further operated upon by later computations, yielding the sequencing operation of the FMC.

We begin by constructing a domain D to interpret terms. A stack can be seen as an element of $D^{\mathbb{N}}$. A term takes a stack and, after computation, returns a new stack as its result. We suppose that computations are modelled using an (unspecified) strong monad T on **Cppo**; for now think of T as the lifting monad. Then a term would be an element of a domain satisfying $D \cong D^{\mathbb{N}} \to TD^{\mathbb{N}}$. This domain equation can be solved by standard techniques. Kleisli function composition gives rise to a sequencing operation $D \times D \to D$ which is associative, and forms a monoid with unit element given by the unit of the monad. We will write $(d_1.d_2)$ for the composition of two elements of D using this operation.

Observe that D is a *reflexive object* in **Cppo** and hence provides a model of the λ-calculus:

$$D \;\cong\; D^{\mathbb{N}} \to TD^{\mathbb{N}} \;\cong\; (D^{\mathbb{N}} \times D) \to TD^{\mathbb{N}} \;\cong\; D \to (D^{\mathbb{N}} \to TD^{\mathbb{N}}) \;\cong\; D \to D.$$

We briefly spell out the semantics of λ-calculus induced by this model. Let ρ range over *environments*: functions from the set of variables to D. We use s to range over $D^{\mathbb{N}}$; $(s \cdot d)$ denotes the stack resulting from pushing d onto s. We shall elide the isomorphism $D \cong D^{\mathbb{N}} \to TD^{\mathbb{N}}$. For any term M and environment ρ we define $[\![M]\!]\rho \in D$ (equivalently, a function $D^{\mathbb{N}} \to TD^{\mathbb{N}}$) as follows.

$$[\![x.\star]\!]\rho = \rho x \qquad [\![[N].\,M]\!]\rho\ s = [\![M]\!]\rho\ (s \cdot [\![N]\!]\rho) \qquad [\![\langle x\rangle.\,M]\!]\rho\ (s \cdot d) = [\![M]\!]\rho'\ s$$

where $\rho'(x) = d$ and $\rho'(y) = \rho(y)$ for $y \neq x$.

These definitions show immediately that application is interpreted by pushing the argument onto the stack, and abstraction by popping a term from the stack. Thus this standard λ-calculus model directly justifies the machine semantics. It extends to the sequential λ-calculus by defining

$$[\![\star]\!]\rho = \eta \qquad [\![x.\,M]\!]\rho = (\rho(x)\ .\ [\![M]\!]\rho),$$

where η is the unit of the monad (and of the monoid).

Thanks to the compositionality of the semantics we can readily prove:

▶ **Lemma 9.** $[\![M]\!]\rho = [\![M]\!]\rho'$ *if* $\forall x \in \mathsf{fv}(M)\ \rho(x) = \rho'(x)$.

As a consequence of this Lemma, we may speak of $[\![M]\!]$, independent of ρ, when M is closed.

We extend the semantics to stacks of closed terms. Suppose the monad T is lifting. A stack S denotes an element of $D^{\mathbb{N}}$:

$$[\![\varepsilon]\!] = \bot \qquad [\![S \cdot M]\!] = \langle [\![S]\!], [\![M]\!] \rangle$$

where we elide the isomorphism $D^{\mathbb{N}} \cong D^{\mathbb{N}} \times D$.

Thanks to the direct correspondence between the semantic equations and the machine transitions, we have:

▶ **Lemma 10** (Soundness). *Whenever* $(S, M) \Downarrow (T, N)$ *(with all terms closed),* $[\![M]\!]([\![S]\!]) = [\![N]\!]([\![T]\!])$.

▶ **Theorem 11** (Adequacy). *If* $[\![M]\!]([\![S]\!]) \neq \bot$ *then there exists a* T *such that* $(S, M) \Downarrow (T, \star)$.

Proof. The proof of this statement follows readily from the existence of three relations: a relation between elements of D and closed terms; a relation between semantic streams in $D^{\mathbb{N}}$ and streams S; and a relation between computations in TD and machine states (S, M). We write $\lhd$ for each of these relations, relying on the types to disambiguate. The relations are required to satisfy the following conditions:

$$\begin{aligned} d \lhd M \quad &\text{iff} \quad \forall \sigma \in D^{\mathbb{N}}, \sigma \lhd S \Rightarrow d(\sigma) \lhd (S, M) \\ \sigma \lhd S \quad &\text{iff} \quad \forall i.\sigma_i \lhd S_i \\ k \lhd (S, M) \quad &\text{iff} \quad k = \mathsf{lift}(\sigma) \Rightarrow (S, M) \Downarrow (T, \star) \text{ and } \sigma \lhd T. \end{aligned}$$

$$\frac{}{\Gamma \vdash \star : \overleftarrow{\tau}_A \Rightarrow \overrightarrow{\tau}_A}\ \star \qquad \frac{}{\Gamma, x : \alpha \vdash x : \alpha}\ \textsf{base} \qquad \frac{\Gamma, x : \rho \vdash M : \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A}{\Gamma \vdash a\langle x\rangle.\, M : a(\rho)\, \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A}\ \textsf{abs}$$

$$\frac{\Gamma \vdash N : \rho \qquad \Gamma \vdash M : a(\rho)\, \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A}{\Gamma \vdash [N]a.\, M : \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A}\ \textsf{app} \qquad \frac{\Gamma, x : \overleftarrow{\rho}_A \Rightarrow \overrightarrow{\sigma}_A \vdash \quad M : \overleftarrow{\sigma}_A\, \overleftarrow{\tau}_A \Rightarrow \overrightarrow{\upsilon}_A}{\Gamma, x : \overleftarrow{\rho}_A \Rightarrow \overrightarrow{\sigma}_A \vdash x.\, M : \overleftarrow{\rho}_A\, \overleftarrow{\tau}_A \Rightarrow \overrightarrow{\upsilon}_A}\ \textsf{var}$$

Figure 1 Typing rules for the Functional Machine Calculus.

These conditions cannot be used as a definition of the relations $\lhd$, for example as a fixed point of an operator on such relations, because the first clause contains a negatively-occurring usage of $\lhd$. Nevertheless the existence of such relations can be established using standard techniques of denotational semantics. Pitts's work [11] gives an elegant general theory which enables the construction of such relations.

Once $\lhd$ has been shown to exist, a straightforward induction on syntax establishes that for any term M, and any finite stream S, we have

$$[\![M]\!] \lhd M \qquad [\![S]\!] \lhd S \qquad [\![M]\!]([\![S]\!]) \lhd (S, M)$$

from which computational adequacy immediately follows. ◀

Note that our soundness and adequacy results are expressed in terms of the stack-machine evaluation mechanism. It is also the case that the denotational semantics validates the beta- and eta-laws of Definition 7, but our point in this section is to emphasise that the stack-machine semantics can be seen as an implementation of a natural denotational model.

Our denotational semantics so far gives an account of the sequential λ-calculus. To extend to the FMC, we replace the lifting monad with the *state monad* $TX = St \to (St \times X)_\perp$, where St is a domain of states. Our domain equation becomes

$$D \cong D^{\mathbb{N}} \to (St \to (St \times D^{\mathbb{N}})_\perp) \cong St \times D^{\mathbb{N}} \to (St \times D^{\mathbb{N}})_\perp$$

If we let $St = D^{\mathbb{N}}$, so that the values in the state are stacks, this is a domain equation for "two-stack transformers". As above, this is a reflexive object, now in *two distinct ways* depending on which stack is used to interpret the arguments. This is exactly the FMC with two locations; extension to any finite set of locations is handled similarly, and the soundness and adequacy results may be proved in the same way. As we emphasized in the introduction, this semantics has the remarkable property that the stack used to interpret the operations of the λ-calculus has exactly the same status as that used to interpret state, and it is merely convention that distinguishes the two. This is precisely the point of view embodied by the novel syntax and operational semantics of the FMC.

4 Simple types

Simple types for the FMC [6] describe the input/output behaviour of the stack machine. The type system has three levels, mirroring the syntactic categories of the machine: types τ for terms M, type vectors $\vec{\tau}$ (or *stack types*) for stacks S, and location-indexed families of type vectors $\vec{\tau}_A$ (or *memory types*) for memories S_A. A function type is then an implication between an input memory type and an output memory type.

▶ **Definition 12.** FMC-types ρ, σ, τ, υ *over a set of* base types Σ *are given by:*

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

Equivalently, one may view a function type as an implication between two vectors of location-indexed types, considered modulo the permutation of types on different locations.

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

We introduce the following notation, which will enable us to write types also in the manner above. The *empty* type vector is ε, and the empty memory type ε_A. A *singleton* memory type $a(\vec{\tau})$ is empty at every location except a, where it has $\vec{\tau}$: $a(\vec{\tau})_a = \vec{\tau}$ and $a(\vec{\tau})_b = \varepsilon$ for $a \neq b$. A singleton $\lambda(\vec{\tau})$ on the main location λ may be written as $\vec{\tau}$. *Concatenation* of type vectors is denoted by juxtaposition and the *reverse* of a type vector $\vec{\tau} = \tau_1 \dots \tau_n$ is written $\overleftarrow{\tau} = \tau_n \dots \tau_1$. This extends point-wise to families, so $\vec{\sigma}_A \vec{\tau}_A = \{\vec{\sigma}_a \vec{\tau}_a \mid a \in A\}$ and $\overleftarrow{\tau}_A = \{\overleftarrow{\tau}_a \mid a \in A\}$.

▶ **Definition 13.** *A* judgement $\Gamma \vdash M : \tau$ *is a typed term in a* context $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$, *a finite function from variables to types. The typing rules for the FMC are given in Figure 1.*

▶ **Example 14.** The terms from Example 2 can be typed as follows, where $\mathbb{Z}$ is a base type of integers. Recall that the first term adds a random number to the cell c, and the second sends the numbers from zero to two to output, leaving the number three on the main stack.

$$\mathsf{rnd}\langle x\rangle.\,[x].\,c\langle y\rangle.\,[y].\,+\,.\,\langle z\rangle.\,[z]c : \mathsf{rnd}(\mathbb{Z})\, c(\mathbb{Z}) \Rightarrow c(\mathbb{Z})$$
$$[\langle x\rangle.\,[x]\mathsf{out}.\,[x].\,[1].\,+].\,\langle f\rangle.\,[0].\,f.\,f.\,f \ : \ \Rightarrow \mathsf{out}(\mathbb{Z}\,\mathbb{Z}\,\mathbb{Z})\,\mathbb{Z}$$

Note, there are two typing rules for variables: one for variables of base type, and one for variables of higher type. The simply-typed FMC satisfies the subject reduction property [6], which is implicitly used in the following section.

5 Strong normalization

We will show that reduction for the simply-typed FMC is strongly normalizing (SN). Our proof is a variant of Gandy's for the simply-typed λ-calculus [3]. Gandy's proof interprets terms in domains of strictly ordered, strict monotone functionals: the base domain is $\mathbb{N}^< = (\mathbb{N}, <_\mathbb{N})$, and if $X = (|X|, <_X)$ and $Y = (|Y|, <_Y)$ are domains then so is $X \to Y$ where

$$\begin{array}{rcl} |X \to Y| & = & \{f \in Y^X \mid \forall x, x' \in X.\ x <_X x' \implies f(x) <_Y f(x')\} \\ f <_{X \to Y} g & \iff & \forall x \in X.\ f(x) < g(x)\ . \end{array}$$

The interpretation takes types to domains and terms of a given type to elements of that domain. The domains are well-founded and the interpretation of terms is such that it decreases on reduction, giving SN. One may further collapse a functional to a natural number to give an overestimate of the longest reduction sequence of a term. The literature has several variants on this proof, including one by De Vrijer that calculates longest reduction sequences exactly [2]; see also [18, 15, 14].

We introduce a (to the best of our knowledge) new variant, that avoids the domain of *strict* functionals and instead interprets terms in the – more standard – domain of (non-strict) monotone functionals, as above but with $\leq$, generated from $\mathbb{N}^\leq = (\mathbb{N}, \leq_\mathbb{N})$ with $\to$. This domain is not well-founded, but our interpretation of terms ensures that when functionals are collapsed to a natural number, this strictly decreases upon reduction, giving SN.

The technical difference is small and subtle. Gandy's proof originates in ΛI, where abstracted variables must occur, and hence the interpretation of an abstraction $\lambda x.M$ is naturally strict: the argument to x always contributes to the overall interpretation. To generalize to the λ-calculus, where x need not occur in $\lambda x.M$, a construction is introduced to nevertheless measure the argument to x, so that the functional for $\lambda x.M$ remains strictly monotone. The literature has several further such constructions [4].

This solves the challenge of accounting for reduction in terms that will be discarded, common in SN proofs. In our proof, instead we account for such terms when they are supplied as arguments: for a term $M\,N$ we increment the overall measure with that for N, measuring potential reduction in N even if it will be discarded by M. An abstraction $\lambda x.M$ may then be interpreted as a standard monotone functional, avoiding strictness.

To build our domains, we use the $\to$ construction above, as well as the product of domains $X \times Y$ and an indexed product $\Pi_{a\in A}\, X_a$, defined in the expected way, as follows. Note, we will omit to work with base types in this section, so the base case is given by $(\Rightarrow)$.

$$\begin{aligned} |X \times Y| &= |X| \times |Y| & (x,y) \le_{X\times Y} (x',y') &\iff x \le_X x' \wedge y \le_Y y' \\ |\Pi_{a\in A}\, X_a| &= \Pi_{a\in A}\, |X_a| & x \le_{\Pi_{a\in A} X_a} x' &\iff \forall a \in A.\ x_a \le_{X_a} x'_a \end{aligned}$$

▶ **Definition 15.** *The* interpretation *of an FMC-type τ is the domain $[\![\tau]\!]$ given by:*

$$[\![\overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A]\!] = [\![\overrightarrow{\sigma}_A]\!] \ \to \mathbb{N}^{\le} \times [\![\overrightarrow{\tau}_A]\!] \qquad [\![\tau_1 \dots \tau_n]\!] = [\![\tau_1]\!] \times \dots \times [\![\tau_n]\!] \qquad [\![\overrightarrow{\tau}_A]\!] = \Pi_{a\in A}\, [\![\overrightarrow{\tau}_a]\!]$$

It is worth observing that for the simple types of the λ-calculus, as embedded in the FMC, these domains are the natural ones. Briefly (see [6] for details), a simple type $\tau_1 \to \dots \to \tau_n \to o$ embeds as the FMC-type $\tau_1 \dots \tau_n \Rightarrow$ with the domain $[\![\tau_1]\!] \times \dots \times [\![\tau_n]\!] \to \mathbb{N}^{\le}$, which is the expected one modulo Currying.

▶ **Definition 16.** *The least element of a domain $[\![\tau]\!]$ is written 0_τ. The* collapse *function $\lfloor - \rfloor_\tau : [\![\tau]\!] \to \mathbb{N}$ takes a functional to a natural number by providing a least element as input and discarding all other output: $\lfloor f \rfloor_{\overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A} = \pi_1(f(0_{\overrightarrow{\sigma}_A}))$.*

We will interpret terms such that if $M : \tau$ then $[\![M]\!] \in [\![\tau]\!]$, and if $M \longrightarrow N$ at type τ then both $[\![M]\!] \ge_{[\![\tau]\!]} [\![N]\!]$ and $\lfloor [\![M]\!] \rfloor >_{\mathbb{N}} \lfloor [\![N]\!] \rfloor$, to give SN. We introduce the following notation. To interpret terms in a context Γ, let a *valuation* v on Γ be a function assigning to each variable $x : \tau$ in Γ a value $v(x) \in [\![\tau]\!]$. The valuation $v\{x \leftarrow t\}$ assigns t to x and otherwise behaves as v. We write elements of product domains as vectors $(t_1, \dots, t_n)$, and will elide the isomorphisms for associativity and unitality so that concatenation of s and t may be written (s, t). Concatenation lifts to indexed products pointwise: $(s, t)_a = (s_a, t_a)$. For $t \in [\![\tau]\!]$ we have a singleton $a(t) \in [\![a(\tau)]\!]$ where $a(t)_a = t$ and $a(t)_b = ()$ for $b \ne a$.

▶ **Definition 17.** *For a term $\Gamma \vdash M : \tau$ and valuation v on Γ, we inductively define the* interpretation *$[\![\Gamma \vdash M : \tau]\!]_v \in [\![\tau]\!]$ as follows, omitting Γ for compactness.*

$$\begin{aligned} [\![\star : \overleftarrow{\tau}_A \Rightarrow \overrightarrow{\tau}_A]\!]_v(t) &= (0, t) \\ [\![x.\,M : \overleftarrow{\rho}_A\, \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A]\!]_v(s, r) &= (n+m, t) \quad \textit{where } (n, u) = v(x : \overleftarrow{\rho}_A \Rightarrow \overrightarrow{\upsilon}_A)(r) \\ & \qquad\qquad\qquad\qquad\ (m, t) = [\![M : \overleftarrow{\upsilon}_A\, \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A]\!]_v(s, u) \\ [\![a\langle x\rangle.\,M : a(\rho)\, \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A]\!]_v(s, a(r)) &= (1+m, t) \quad \textit{where } (m, t) = [\![M : \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A]\!]_{v\{x \leftarrow r\}}(s) \\ [\![[N]a.\,M : \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A]\!]_v(s) &= (1+m+\lfloor f \rfloor, t) \quad \textit{where } f = [\![N : \rho]\!]_v \\ & \qquad\qquad\qquad\qquad\ (m, t) = [\![M : a(\rho)\, \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A]\!]_v(s, a(f)) \end{aligned}$$

We write $[\![\Gamma \vdash M : \tau]\!]$ for $[\![\Gamma \vdash M : \tau]\!]_v$ with v the least valuation $v(x : \tau) = 0_\tau$, and may abbreviate $[\![\Gamma \vdash M : \tau]\!]_v$ to $[\![M : \tau]\!]_v$ or $[\![M]\!]_v$.

▶ Remark 18. In this definition, the application case $[\![[N]a.\,M]\!]_v(s) = (1+m+\lfloor f \rfloor)$ adds the value $\lfloor f \rfloor$ to account for reduction inside the argument N. Further, both it and the abstraction case $[\![a\langle x\rangle.\,M]\!]_v(s, a(r)) = (1+m, t)$ add 1 to count redexes, so that a reduction step reduces the overall measure by (at least) 2. It would suffice to count only abstractions or only applications, but the choice to count both is so that we count steps of the stack machine. We observe the following: for the alternative interpretation that omits to count $\lfloor f \rfloor$, and instead has $[\![[N]a.\,M]\!]_v(s) = (1+m)$, the collapsed interpretation $\lfloor [\![M]\!] \rfloor$ measures the exact length of machine runs for M. This observation provides the proof with an operational intuition: terms are strongly normalizing because a) types guarantee termination of the machine [6, Theorem 3.12], and b) reduction shortens the length of machine runs.

For the remainder of the proof, we will give an overview by stating the main lemmata. Each follows by a straightforward induction on typing derivations. First, for the interpretation $[\![-]\!]$ to be well-defined, the construction for each term must be shown to preserve monotonicity. We will do so in the following lemma. For valuations v and w over Γ, let $v \leq w$ denote that $v(x) \leq_{[\![\tau]\!]} w(x)$ for all $x\colon\tau$ in Γ.

▶ **Lemma 19.** *For all terms $\Gamma \vdash M\colon \tau$ and valuations $v \leq w$ over Γ, we have that:*

1. $[\![M]\!]_v \in [\![\tau]\!]$
2. $[\![M]\!]_v \leq_{[\![\tau]\!]} [\![M]\!]_w$.

For the next steps, we first need that the interpretation commutes with sequential composition $M\,;N$ and substitution $\{N/x\}M$. Then, we show that reduction (non-strictly) decreases the interpretation, and strictly decreases the collapsed interpretation.

▶ **Lemma 20.** *For terms $\Gamma \vdash M\colon \overleftarrow{\sigma_A}\overleftarrow{\tau_A} \Rightarrow \overrightarrow{\upsilon_A}$ and $\Gamma \vdash N\colon \overleftarrow{\rho_A} \Rightarrow \overrightarrow{\sigma_A}$ and valuation v on Γ,*

$$[\![N\,;M]\!]_v(t, r) = (i+j, u) \quad \textit{where} \quad [\![N]\!]_v(r) = (i, s) \quad \textit{and} \quad [\![M]\!]_v(t, s) = (j, u)\ .$$

▶ **Lemma 21.** *For terms $\Gamma \vdash N\colon \sigma$ and $\Gamma, x\colon\sigma \vdash M\colon\tau$ and valuation v on Γ,*

$$[\![\{N/x\}M]\!]_v = [\![M]\!]_{v\{x \leftarrow [\![N]\!]_v\}}\ .$$

▶ **Lemma 22.** *If $\Gamma \vdash M \longrightarrow N\colon\tau$ then $[\![M]\!]_v \geq_{[\![\tau]\!]} [\![N]\!]_v$ for every valuation v on Γ.*

▶ **Lemma 23.** *If $\Gamma \vdash M \longrightarrow N\colon \overleftarrow{\sigma_A} \Rightarrow \overrightarrow{\tau_A}$ then $\pi_1([\![M]\!]_v(s)) >_{\mathbb{N}} \pi_1([\![N]\!]_v(s))$ for every $s \in [\![\overrightarrow{\sigma_A}]\!]$ and valuation v on Γ.*

The last lemma then immediately gives the strong normalization result.

▶ **Theorem 24** (Strong Normalization)**.** *Simply-typed FMC-terms are strongly normalizing with respect to beta-reduction.*

Proof. By Lemma 23 if $\Gamma \vdash M \longrightarrow N\colon\tau$ then $\lfloor [\![M]\!] \rfloor >_{\mathbb{N}} \lfloor [\![N]\!] \rfloor$, so that $\lfloor [\![M]\!] \rfloor$ gives a bound for the length of any reduction path from M. ◀

Note that it is easy to extend this result to include eta-reduction: since eta-reduction does not increase the measure, and is clearly strongly normalizing by itself (the size of the term decreases), we can interleave each beta-reduction step with an arbitrary number of eta-reduction steps without affecting strong normalization.

6 Categorical semantics

We give the categorical view on the FMC in three layers:

- terms with composition $N\,;M$ and unit $\star$ form a category;
- terms modulo $\beta\eta$-equivalence form a *premonoidal* category [16];
- terms modulo an appropriate equational theory form a complete language for Cartesian closed categories;

we then show that *machine equivalence*, where terms are equivalent if they display the same input/output behaviour on the machine, validates the final equational theory.

The idea that a calculus with effects should semantically be a CCC may be surprising, so we will first motivate what the semantics does and does not capture. Firstly, and most importantly, the semantics we give here is one of the pure FMC, and emphatically *not* a semantics *of effects*: it ignores that, for instance, *input* only has a *pop* operation but no *push*, and that *state* locations would be restricted to a stack of depth one (at most). Imposing these constraints will cause the CCC semantics to break down, as we will demonstrate later in the section.

Secondly, the situation is analogous to the encoding of monadic effects in simply-typed λ-calculus, where for instance *state* encodes as the monad $S \to (- \times S)$. In that case, too, the semantics remains a CCC, despite the possibility of encoding effects.

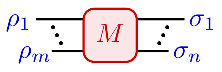
Thirdly, the two generalizations of the FMC, *locations* and *sequencing*, remain close enough to the λ-calculus that simple types allow to collapse them back onto a CCC semantics. *Locations* give multiple copies of the λ-calculus, but because the types give the entire memory, the semantics may combine the different indexed stacks into one, projecting the multiple copies onto a single λ-calculus. *Sequencing* gives control over evaluation and allows us to encode various reduction strategies, but the point of the denotational perspective is precisely to collapse any computational behaviour, and only consider the input/output behaviour of a term.

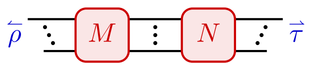
The purpose of our CCC semantics is to demonstrate that the simply-typed FMC is an *operational* refinement of the lambda-calculus, but not a *denotational* one. The FMC allows to express *how* computation takes place: what reduction strategy is used, whether inputs are passed as function arguments or via mutable store, when the random generator is consulted, *etc.* The denotational perspective then collapses these distinctions, demonstrating that we remain firmly in the domain of higher-order functional computation, despite the ability to encode effects.

The plain category

For simplicity we will work in the sequential λ-calculus. The arguments generalize straightforwardly to the case of the FMC, and the details of this case are to be given in the first author's Ph.D. thesis. The objects are then type vectors $\vec{\tau}$ and morphisms in $\vec{\sigma} \longrightarrow \vec{\tau}$ will be *closed* terms $M : \overleftarrow{\sigma} \Rightarrow \vec{\tau}$ modulo the given equivalence.

A term $M : \rho_1 \dots \rho_m \Rightarrow \sigma_n \dots \sigma_1$ may be represented by a string diagram as below, left. The wires represent the input and output stacks, with the first element at the top. Strict composition of terms $M : \overleftarrow{\rho} \Rightarrow \vec{\sigma}$ and $N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}$ into $M\,;N : \overleftarrow{\rho} \Rightarrow \vec{\tau}$, given below, right.



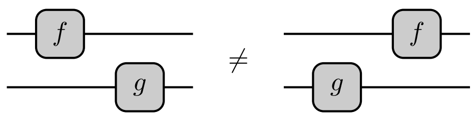


Analogous to the notation for type vectors, we introduce the following notation. We use vector notation for variables, $\vec{x} = x_1 \dots x_n$, and reverse a vector by a left-pointing arrow: if $\vec{x}$ is as before, then $\overleftarrow{x} = x_n \dots x_1$. Concatenation of vectors is by juxtaposition. We lift the notation to sequences of abstractions and applications, but only for variables: if $\vec{x}$ is as above, then $\langle\overleftarrow{x}\rangle.\,N = \langle x_n\rangle\dots\langle x_1\rangle.\,N$ and $[\vec{x}].\,N = [x_1]\dots[x_n].\,N$. Vector notation is extended to contexts as $x_1 : \tau_1, \dots, x_n : \tau_n \;=\; \vec{x} : \vec{\tau}$ and simultaneous substitutions as $\{S/\vec{x}\} = \{M_1/x_1, \dots, M_n/x_n\}$, where $S = \varepsilon \cdot M_1 \cdots M_n$.

The first, plain category is then given by Lemma 4.

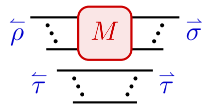
The premonoidal category

A *premonoidal* category [16], like a monoidal category, describes string diagrams, but with a *sequential* element: a premonoidal product $\otimes$ has no *parallel* composition $f \otimes g$, while $(f \otimes \mathsf{id});(\mathsf{id} \otimes g)$ and $(\mathsf{id} \otimes g);(f \otimes \mathsf{id})$ are distinct.

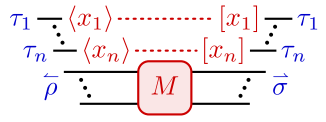


Formally, a premonoidal product is a binary operation on objects $X \otimes Y$ that is a functor in each argument, $- \otimes X$ and $X \otimes -$, but need not be a bifunctor $- \otimes -$. In the FMC, the action on objects is concatenation, $\vec{\sigma} \otimes \vec{\tau} \;\triangleq\; \vec{\sigma}\vec{\tau}$, with the first element at the top of the stack, and the unit given by ε. Both actions on morphisms are given below for $M : \overleftarrow{\rho} \Rightarrow \vec{\sigma}$, where $\overleftarrow{x} : \overleftarrow{\tau}$.

$$M \otimes \vec{\tau} : \; \vec{\tau} \otimes \vec{\rho} \longrightarrow \vec{\tau} \otimes \vec{\sigma} \quad \triangleq \quad M : \overleftarrow{\rho}\overleftarrow{\tau} \Rightarrow \vec{\tau}\vec{\sigma}$$



$$\vec{\tau} \otimes M : \; \vec{\rho} \otimes \vec{\tau} \longrightarrow \vec{\sigma} \otimes \vec{\tau} \quad \triangleq \quad \langle\overleftarrow{x}\rangle.\,M.\,[\vec{x}] : \overleftarrow{\tau}\overleftarrow{\rho} \Rightarrow \vec{\sigma}\vec{\tau}$$



The first is *expansion* (see Property 3.9 of the previous paper [6]). The second lifts the arguments for $\overleftarrow{\tau}$ from the stack as the variables $\overleftarrow{x}$, to restore them after evaluating M. We illustrate these above. A premonoidal product further has an *associator* and a *unitor*, and is called *strict* if these are identities, which they are here. The category is then formed by terms modulo $\beta\eta$-equivalence, where η-equivalence is generated by:

$$M : \rho\overleftarrow{\sigma} \Rightarrow \vec{\tau} \quad =_\eta \quad \langle x\rangle.\,[x].\,M : \rho\overleftarrow{\sigma} \Rightarrow \vec{\tau} \qquad \text{where } x \notin \mathsf{fv}(M)$$

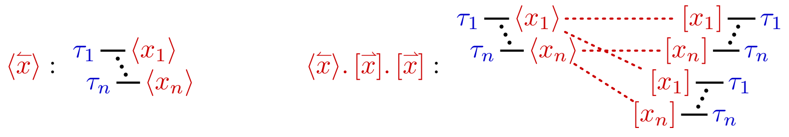
▶ **Proposition 25.** *Terms modulo $\beta\eta$-equivalence form a strict premonoidal category.*

We remark that terms modulo $\beta\eta$-equivalence do *not* form a *symmetric* pre-monoidal category, due to the failure of naturality of symmetry. Of course, one can add further equations to remedy this. In the sequel, we develop an extended equational theory which in fact makes the category of terms Cartesian closed.

Premonoidal structure forces a notion of sequentiality, which has previously been employed to capture that of effects, as in the closed Freyd categories of Power, Thielecke, and Levy [17, 8], which are the premonoidal equivalent of Cartesian closed categories. However, this imposed sequentiality is only necessary if interactions through effects (such as state) are hidden from the type system. Because the FMC makes these explicit, they can instead be accounted for in the semantics, which then reverts to a Cartesian closed category.

The Cartesian closed category

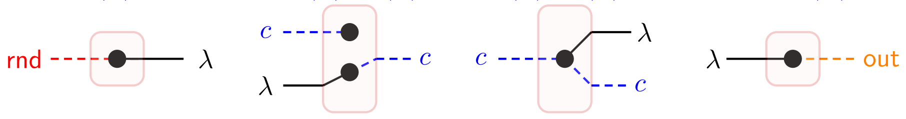
We now give an example illustrating why we would expect the FMC to form a Cartesian (closed) category, despite its ability to encode effects. For this example, but not for the rest of the section, we will then consider terms *with* locations. Note, first, how the two following terms are illustrated in string diagrams below.



$\langle\vec{x}\rangle$:
τ_1 $\langle x_1\rangle$
τ_n $\langle x_n\rangle$
$\langle\vec{x}\rangle.\,[\vec{x}].\,[\vec{x}]$:
τ_1 $\langle x_1\rangle$ $[x_1]$ τ_1
τ_n $\langle x_n\rangle$ $[x_n]$ τ_n
$[x_1]$ τ_1
$[x_n]$ τ_n

▶ **Example 26.** We introduce the following effectful operations as defined constructs ("sugar") into the FMC: reading from a stream of random integers ($\mathbb{Z}$), rand, a memory cell c with get and set operations, and writing to output, write. We give the definitions and types of these operations and illustrate these as as (nominal) string diagrams below, using colours and dashed lines to indicate non-main locations: red for rnd, blue for c and yellow for out. We use the black dot here to depict a transition from one location to another[1] (as in rnd, set and write), as well as to depict the terminal ! (as in set) and diagonal Δ (as in get), as is standard.

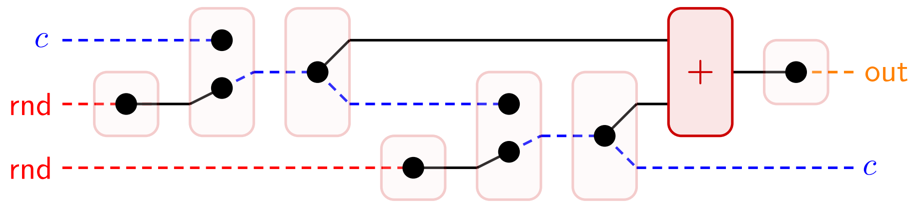
rand	set	get	write
$\mathsf{rnd}\langle x\rangle.\,[x]$	$\langle x\rangle.\,c\langle\_\rangle.\,[x]c$	$c\langle x\rangle.\,[x]c.\,[x]$	$\langle x\rangle.\,[x]\mathsf{out}$
$\mathsf{rnd}(\mathbb{Z})\Rightarrow\mathbb{Z}$	$\mathbb{Z}c(\mathbb{Z})\Rightarrow c(\mathbb{Z})$	$c(\mathbb{Z})\Rightarrow c(\mathbb{Z})\mathbb{Z}$	$\mathbb{Z}\Rightarrow\mathsf{out}(Z)$



Note that, modulo renaming of wires, these effectful operations are encoded by the diagonal and terminal operations of a Cartesian category.

Consider the following term (reprised from Examples 4.4 and 4.8 of the previous paper [6]) and its string diagrammatic representation. Note that we give the diagram modulo symmetry.[2]

rand ; set ; get ; rand ; set ; get ; + ; write



[1] This corresponds to the *renaming* a wire in the formalism of nominal string diagrams, *i.e.*, string diagrams where wires additionally have an associated *name*.

[2] Note that, because operations acting on different locations permute, we have, for example, no need to lift (and restore) the arguments from the stacks at locations λ and c prior to (and subsequent to) applying the second instances of rand, set or get – or, equivalently, doing so has the same result as not doing so. Technically, the term corresponding to the diagram would lift the result of the first instance of get off the main λ stack before applying the second instance of get, and then restore it afterwards; however, since $+$ is symmetric in its inputs, we omit this for simplicity.

Due to the strong type system, we see all the dependencies between operations. For example, the second call to rand may safely be made before the first calls to set and get. This would be illustrated by "sliding" the rand operation along the wire – something which is forbidden in general in a pre-monoidal category, but is permissible in a monoidal setting. We can also see that the second set is dependent on the first get. Indeed, their composition forms a beta-redex, corresponding to the expected interaction of ! with Δ in a Cartesian category.

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

The following data will make $\otimes$ a Cartesian product $\times$, when considered modulo the following equational theory. The *exponent* $\overrightarrow{\sigma}\to\overrightarrow{\tau} \triangleq \overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau}$ will then give Cartesian closure.

$$\begin{array}{rcl}
! : \overrightarrow{\tau} \longrightarrow 1 & = & \langle\overleftarrow{x}\rangle : \overleftarrow{\tau} \Rightarrow \\
\delta : \overrightarrow{\tau} \longrightarrow \overrightarrow{\tau}\times\overrightarrow{\tau} & = & \langle\overleftarrow{x}\rangle.\,[\overrightarrow{x}].\,[\overrightarrow{x}] : \overleftarrow{\tau}\Rightarrow\overrightarrow{\tau}\overrightarrow{\tau} \\
\pi_1 : \overrightarrow{\upsilon}\times\overrightarrow{\tau} \longrightarrow \overrightarrow{\tau} & = & \langle\overleftarrow{x}\rangle.\,\langle\overleftarrow{y}\rangle.\,[\overrightarrow{x}] : \overleftarrow{\tau}\overleftarrow{\upsilon}\Rightarrow\overrightarrow{\tau} \\
\pi_2 : \overrightarrow{\upsilon}\times\overrightarrow{\tau} \longrightarrow \overrightarrow{\upsilon} & = & \langle\overleftarrow{x}\rangle.\,\langle\overleftarrow{y}\rangle.\,[\overrightarrow{y}] : \overleftarrow{\tau}\overleftarrow{\upsilon}\Rightarrow\overrightarrow{\upsilon} \\
\epsilon : \overrightarrow{\sigma}\times(\overrightarrow{\sigma}\to\overrightarrow{\tau}) \longrightarrow \overrightarrow{\tau} & = & \langle z\rangle.\,z : (\overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau})\overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau} \\
\eta : \overrightarrow{\tau} \longrightarrow (\overrightarrow{\sigma}\to\overrightarrow{\sigma}\times\overrightarrow{\tau}) & = & \langle\overleftarrow{x}\rangle.\,[[\overrightarrow{x}]] : \overleftarrow{\tau}\Rightarrow(\overleftarrow{\sigma}\Rightarrow\overrightarrow{\sigma}\overrightarrow{\tau}) \\
M\to N : (\overrightarrow{\sigma}\to\overrightarrow{\tau}) \longrightarrow (\overrightarrow{\rho}\to\overrightarrow{\upsilon}) & = & \langle z\rangle.\,[M.\,z.\,N] : (\overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau})\Rightarrow(\overleftarrow{\rho}\Rightarrow\overrightarrow{\upsilon})
\end{array}$$

where $\overrightarrow{x} : \overrightarrow{\tau}$, $\overrightarrow{y} : \overrightarrow{\upsilon}$, $z : \overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau}$

▶ **Definition 27.** *We define the equational theory $=_{\mathsf{eqn}}$ of the FMC to be the least equivalence generated by the following laws, closed under all contexts.*

$$\begin{array}{lcl}
\text{Beta:} & [N].\,\langle x\rangle.\,M =_\beta M\{N/x\} & \overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau} \\
\text{Interchange:} & \langle\overleftarrow{x}\rangle.\,N.\,[\overrightarrow{x}].\,M =_\iota M.\,\langle\overleftarrow{y}\rangle.\,N.\,[\overrightarrow{y}] & \overleftarrow{\sigma}\overleftarrow{\rho}\Rightarrow\overrightarrow{\upsilon}\overrightarrow{\tau} \\
\text{Diagonal:} & M\,\langle\overleftarrow{y}\rangle.\,[\overrightarrow{y}].\,[\overrightarrow{y}] =_\Delta \langle\overleftarrow{x}\rangle.\,[\overrightarrow{x}].\,M.\,[\overrightarrow{x}].\,M & \overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau}\overrightarrow{\tau} \\
\text{Terminal:} & M.\,\langle\overleftarrow{y}\rangle =_! \langle\overleftarrow{x}\rangle & \overleftarrow{\sigma}\Rightarrow \\
\text{Eta (First-order):} & \star =_\eta \langle a\rangle.\,[a] & \alpha\Rightarrow\alpha \\
\text{Eta (Higher-order):} & P =_\epsilon \langle\overleftarrow{x}\rangle.\,[[\overrightarrow{x}].\,P.\,\langle z\rangle.\,z] & \overleftarrow{\rho}\Rightarrow(\overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau})
\end{array}$$

where $a : \alpha$, $\overrightarrow{x} : \overrightarrow{\sigma}, \overrightarrow{y} : \overrightarrow{\tau}, M : \overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau}$, $N : \overleftarrow{\rho}\Rightarrow\overrightarrow{\upsilon}$, $z : \overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau}$ and $P : \overleftarrow{\rho}\Rightarrow(\overleftarrow{\sigma}\Rightarrow\overrightarrow{\tau})$, and we do not allow abstractions to capture in M, N, or P.

Note that this theory includes beta- and eta-reduction. To see it includes eta-reduction at higher-type, consider the higher-order eta equation with $P = \star$.[3]

▶ **Theorem 28.** *Terms modulo $=_{\mathsf{eqn}}$ form a strict Cartesian closed category.*

[3] Following the definition of substitution, given a context $\{-\}.\,M$ with hole $\{-\}$, the substitution of a term N into the hole is given by $N\,;M$, in particular with N not binding in M. This means the eta equations together give $\langle x\rangle.\,[x].\,M = M$, where $x \notin \mathsf{fv}(M)$.

The proof of the theorem above provides a canonical functor from the free Cartesian closed category generated over a set of base types Σ, denoted $\mathsf{CCC}(\Sigma)$, to the category of FMC terms generated over th same signature, denoted $\Lambda S/\mathsf{eqn}$. We construct a left-inverse CCC-functor interpreting FMC-terms into λ-terms (with products and patterns), thus proving completeness. In general, all constructions also work with also with constants drawn from a monoidal signature, as well as simply with a signature given by a set of base types.

The interpretation $[\![-]\!] : \Lambda S/\mathsf{eqn} \to \mathsf{CCC}(\Sigma)$ preserves types. The top-level arrow $\Rightarrow$ of FMC-types becomes sequent entailment $\vdash$: the type of the input stack becomes the type of the λ-context and the type of the output stack becomes the type of the λ-term.

A *valuation* v is a function assigning to each FMC-variable $x \colon \tau$ a λ-term $v(x) \in [\![\tau]\!]$. Given a valuation v, let $v\{x \leftarrow t\}$ denote the valuation which assigns t to x and otherwise behaves as v. We write contexts and products as vectors and elide the isomorphisms for associativity and unitality so that concatenation of s and t may be written as $s \cdot t$.

▶ **Definition 29.** *For each valuation v, define on the type derivation of $\Gamma \vdash M \colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}$ an open λ-term $[\![\Gamma \vdash M \colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}]\!]_v$, given by its action on contexts:*

$$\begin{aligned}
[\![\Gamma \vdash \star \colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\sigma}]\!]_v(s) &= s \\
[\![\Gamma \vdash x \colon \alpha]\!]_v &= v(x) \\
[\![\Gamma \vdash \langle x\rangle.M \colon \rho\overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}]\!]_v(s \cdot r) &= [\![\Gamma, x \colon \rho \vdash M \colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}]\!]_{v\{x \leftarrow r\}}(s) \\
[\![\Gamma \vdash [N].M \colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}]\!]_v(s) &= [\![\Gamma \vdash M \colon \rho\overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}]\!]_v(s \cdot [\![\Gamma \vdash N \colon \rho]\!]_v) \\
[\![\Gamma, x \colon \overleftarrow{\rho} \Rightarrow \overrightarrow{\upsilon} \vdash x.M \colon \overleftarrow{\rho}\overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}]\!]_v(s \cdot r) &= [\![\Gamma, x \colon \overleftarrow{\rho} \Rightarrow \overrightarrow{\upsilon} \vdash M \colon \overleftarrow{\upsilon}\overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}]\!]_v(s \cdot v(x)(r))
\end{aligned}$$

▶ **Theorem 30.** *Terms modulo $=_{\mathsf{eqn}}$ form a complete language for Cartesian closed categories.*

Machine Equivalence

A natural contextual equivalence on terms is given by *machine equivalence*, defined inductively on types below. It resembles the logical relation for program equivalence of Pitts and Stark [12]. We write $(S, M)\Downarrow$ for T if $(S, M) \Downarrow (T, \star)$ and take here the only constants to be of base type.

▶ **Definition 31.** *Closed terms $M \colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}$ and $M' \colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}$ are* machine equivalent *at type $\overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau}$ if for equivalent inputs the machine gives equivalent outputs,*

$$M \sim M' \colon \overleftarrow{\sigma} \Rightarrow \overrightarrow{\tau} \;\triangleq\; \forall\, S{\sim}S' : \overrightarrow{\sigma}.\; (S, M)\Downarrow \,\sim\, (S', M')\Downarrow \,:\, \overrightarrow{\tau}$$

where two terms of base type are equivalent if they are equal, and two stacks are equivalent if their terms are pairwise equivalent. Equivalence extends to open terms $\Gamma \vdash M \colon \tau$ and $\Gamma \vdash M' \colon \tau$ as follows: $\overrightarrow{w} \colon \overrightarrow{\omega} \vdash M \sim M' \colon \tau$ if and only if

$$\forall\, W{\sim}W' : \overrightarrow{\omega}.\; \{W/\overrightarrow{w}\}M \,\sim\, \{W'/\overrightarrow{w}\}M' : \tau \;.$$

Machine equivalence validates the equational theory (and in particular the beta and eta equations). Thus we have the following result.

▶ **Theorem 32.** *Terms modulo machine equivalence form a Cartesian closed category.*

In fact, the category given by terms modulo machine equivalence is just the extensional collapse of the category of terms modulo the equational theory. Note that machine equivalence is strictly coarser than the equational theory: a situation analogous to that of the simply-typed λ-calculus with products, considered modulo an appropriate contextual equivalence.

7 Further work

The current type system for the FMC is *too strong* for practical programming: it captures such intensional (and unobservable) aspects of computation as the number of elements read from a random stream. We aim to investigate more abstract type systems, including dealing with the particular properties of effectful locations. The results in this paper concerning the type system, which is essentially a presentation of intuitionistic logic, the operational intuition and the close denotational relationship with the λ-calculus make a strong basis for future refinements which account properly for effects. There are several ways in which we already know how to weaken the type system: introducing a recursor, stream types τ^*, which type a stream of terms of type τ, and ignoring types on non-main locations. A close link with string diagrams is evident from the results presented, including with the recently introduced higher-order string diagrams for CCCs [1]. This is another avenue for investigation.

References

1 Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. *Computer Science Logic (CSL) 2023*, 2023. arXiv:2107.13433.

2 Roel de Vrijer. Exactly estimating functionals and strong normalization. *Indagationes Mathematicae (Proceedings)*, 90(4):479–493, 1987.

3 Robin Gandy. Proofs of strong normalization. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press, 1980.

4 Inge Gørtz, Signe Reuss, and Morten Sørensen. Strong normalization from weak normalization by translation into the Lambda-I-calculus. *Higher-Order and Symbolic Computation*, 16:253–285, 2003. doi:10.1023/A:1025693307470.

5 Masahito Hasegawa. Decomposing typed lambda-calculus into a couple of categorical programming languages. In *International Conference on Category Theory and Computer Science*, 1995.

6 Willem Heijltjes. The functional machine calculus. To appear in Mathematical Foundations of Programming Semantics (MFPS 2022). Preprint available at http://people.bath.ac.uk/wbh22/index.html#FMC2022, 2022.

7 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20:199–207, 2007. doi:10.1007/s10990-007-9018-9.

8 Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call–by–value programming languages. *Information and Computation*, 185:182–210, 2003.

9 Robin Milner. Action calculi, or syntactic action structures. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*, volume 711 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 1993. doi:10.1007/3-540-57182-5_7.

10 Slava Pestov, Daniel Ehrenberg, and Joe Groff. Factor: A dynamic stack-based programming language. *ACM SIGPLAN Notices*, 45(12):43–58, 2010.

11 Andrew Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.

12 Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher order operational techniques in semantics*, pages 227–273. Isaac Newton Institute for Mathematical Sciences, 1998.

13 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.

14 Jaco van de Pol. Two different strong normalization proofs? In *Selected Papers from the Second International Workshop on Higher Order Algebra, Logic, and Term Rewriting (HOA '95)*, volume 1074 of *LNCS*, pages 201–220, 1995. doi:10.1007/3-540-61254-8_27.

15 Jaco van de Pol and Helmut Schwichtenberg. Strict functionals for termination proofs. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications (TLCA '95)*, pages 350–364. Springer-Verlag, 1995.

16 John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7:453–468, 1997.

17 John 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.

18 Helmut Schwichtenberg. Complexity of normalization in the pure typed lambda-calculus. In Anne Sjerp Troelstra and Dirk van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 453–457. Elsevier, 1982.

19 Thomas Streicher and Bernhard Reus. Classical logic, continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6):543–572, 1998.

20 Glynn Winskel. *The formal semantics of programming languages: An introduction.* MIT Press, Cambridge, Massachusetts, 1993.