Electronic Notes in
Theoretical Informatics
and Computer Science

ENTICS
https://entics.episciences.org

Volume 5
Proceedings of
MFPS 2025

The Functional Machine Calculus III: Control

Willem Heijltjes

Department of Computer Science
University of Bath
United Kingdom

Abstract

The Functional Machine Calculus (Heijltjes 2022) is a new approach to unifying the imperative and functional programming paradigms. It extends the lambda-calculus, preserving the key features of confluent reduction and typed termination, to embed computational effects, evaluation strategies, and control flow operations. The first instalment modelled sequential higher-order computation with global store, input/output, probabilities, and non-determinism, and embedded both the call–by–name and call–by–value lambda-calculus, as well as Moggi's computational metalanguage and Levy's call–by–push–value. The present paper extends the calculus from sequential to branching and looping control flow. This allows the faithful embedding of a minimal but complete imperative language, including conditionals, exception handling, and iteration, as well as constants and algebraic data types.

The calculus is defined through a simple operational semantics, extending the (simplified) Krivine machine for the lambda-calculus with multiple operand stacks to model effects and a continuation stack to model sequential, branching, and looping computation. It features a confluent reduction relation and a system of simple types that guarantees termination of the machine and strong normalization of reduction (in the absence of iteration). These properties carry over to the embedded imperative language, providing a unified functional–imperative model of computation that supports simple types, a direct and intuitive operational semantics, and a confluent reduction semantics.

Keywords: lambda-calculus, computational effects, exception handling, simple types

1 Introduction

An interesting challenge in programming language theory is to find good ways of extending the λ-calculus with computational effects. The λ-calculus, established by Landin's seminal work [34,35] as the basis of the *functional* programming paradigm, gives a canonical treatment of higher-order functions that sets the standard for good semantic properties such as compositionality, referential transparency, and type safety. However, these do not readily extend to effects. The *imperative* paradigm, meanwhile, sets expectations for a treatment of effects: a clear syntax with a direct, intuitive operational meaning and the seamless integration of multiple effects.

The overarching aim of the challenge is thus to reconcile both paradigms. This requires a unified model of computation that naturally accommodates both higher-order functions and sequential operations, while supporting types, a confluent reduction semantics, and other means of reasoning. In line with the

[1] Email: w.b.heijltjes@bath.ac.uk

[2] Thanks to Ohad Kammar, Guy McCusker, Olivier Laurent, and Nicolas Wu for discussions and feedback.

Rec'd Apr 3,2025; Pub'd Dec 15,2025
10.46298/entics.16682

Proceedings Available Online at
ENTICS MFPS 41 Proceedings

ambition of these requirements, approaches to address this challenge include some of the most influential developments in the field, such as Moggi's account of effects as *monads* [44], Levy's *call–by–push–value* (cbpv) [38], and Plotkin and Pretnar's *effect handlers* [51].

The Functional Machine Calculus (FMC) is a new approach to this challenge. The idea is to take the Krivine Machine [32] seriously as the notion of *sequentiality* in the λ-calculus. This provides a direct operational semantics in a simple abstract machine with a single stack, where *application* is *push*, *abstraction* is *pop*, and *variable* is *execute*. The design principle is then to extend the machine in natural and minimal ways to capture a wide range of imperative features, maintaining the calculus simultaneously as a direct instruction language for the machine, and as a calculus that supports confluent reduction and simple types: a *"machine calculus"*, a machine language that is also a calculus. Previously, the following two extensions were introduced [3,20].

Sequencing The λ-calculus is extended with imperative *sequencing* and *skip*, its unit, as *composition* and *identity* for stack operations. This gives a model of higher-order sequential computation that encompasses both call–by–name and call–by–value behaviour. It faithfully embeds Plotkin's call–by–value (cbv) λ-calculus [52], Moggi's computational metalanguage [44], and Levy's cbpv [38]. Sequencing is implemented on the machine in standard fashion with a continuation stack (elided in the original presentation but introduced in [2,23]).

Locations The machine is generalized from one to many operand stacks, each named by a *location* and directly accessed through push- and pop-instructions. These may then model various computational effects: *mutable higher-order store* as stacks of depth one; *input/output* as pop-only and push-only streams; and *probabilities* and *non-determinism* as probabilistically, respectively non-deterministically generated streams.

This paper presents the following third extension.

Control The machine is generalised from strictly sequential to *branching* and *looping* control flow. The *skip* command is replaced with a set of *choice* labels, each indicating a branch of the computation, while sequencing becomes conditional on a choice, composing on the given branch only. A *loop* construct is introduced that repeats on a given choice and exits otherwise. Computationally, choice labels represent *constants*, *exceptions*, and *data constructors*. Various notions of control flow then embed into the FMC: *exception handling*, *conditionals* and more generally *algebraic data types*, and *iteration* with *escape*.

What these control constructs have in common is that, semantically, they are modelled by *sums* or *coproducts*. The exception monad $TX = E + X$ is the coproduct of the value type X with a type E for exceptions. The data type of booleans is the sum $1+1$ and a conditional is a co-diagonal $[f, g] : 1+1 \to X$, which algebraic data types generalise to sums of products indexed by named constructors. Iteration is modelled by taking a morphism $f : A \to A+B$ to one $\mathsf{iter}\, f : A \to B$, which loops on A and exits on B [7] (the construction is semantically dual to recursion [58]).

The purpose of this paper is to capture these constructs in the FMC with a minimal and natural extension to the machine and the calculus. The concrete contributions are the calculus itself, its small-step operational semantics in the extended stack machine, a big-step operational semantics, a confluent reduction relation, and a notion of simple types that (in the absence of loops) guarantees successful termination of the machine and strong normalization of reduction. The proofs of these properties moreover employ standard techniques, in straightforward but, in some cases, novel ways. Confluence (Section 6) is by *parallel reduction* [63]. Machine termination (Section 8) is by a reducibility argument [62] with a direct induction on typing derivations, without the combinatorial reasoning typical of strong normalization proofs. Strong normalization (Section 9) is by extending big-step evaluation to count transitions on the machine, and then demonstrating that reduction shortens machine evaluation, thus separating the logical content (again by reducibility) from the combinatorics (comparing reduction against the machine).

The FMC itself is a minimal language of six syntactic constructs, each representing a natural instruction on a simple machine with multiple operand stacks and one continuation stack. It nevertheless faithfully embeds a complete imperative language with store, exception handling, and loops, as well as the cbv λ-calculus [52], the computational metalanguage [44], and cbpv [38]. Moreover, these embeddings are by *macro-expansion*, where the encoded constructs are given as definitions (or *syntactic sugar*) over FMC terms. As a consequence, confluent reduction and simple types are directly conferred on imperative constructs. This suggests that, in Levy's terminology [37], these embeddings may be conjectured to be *subsuming*: the semantics of the encoded constructs is expected to arise as that of their FMC encoding. This is immediate for the operational semantics, as presented in this paper. It was recently confirmed for *intersection types*: established notions of intersection typing for store [10,12] arise naturally via the embedding of store in the intersection-typed FMC with *sequencing* and *locations* [22]. Other notions of semantics will be investigated in future.

More specifically, the FMC is a new solution to the problem of combining multiple effects. Both user-defined monads such as *state* $TX = S \to (S \times X)$ and user-defined effect handlers support confluence and types, by encoding effects into the λ-calculus with inductive data types. But monads do not compose, and require monad transformers to combine multiple effects [40] (their limitations are discussed in [30]). Effect handlers create a two-layered system, separating the *handlers* from the *effect operations*, in particular complicating the treatment of exceptions since *raising* is an effect operation while *catching* is a handler. As a consequence, translating imperative constructs into monads or handlers is more involved. The main contribution of the FMC is to allow a direct embedding of a minimal but complete imperative language, seamlessly combining multiple effects within a minimal, typed functional calculus.

While monads are universal for effects, and handlers for algebraic effects [50], the FMC as presented here covers a substantial but fixed set of effects only. This is being addressed in a present line of enquiry by introducing effect handlers into the FMC, which appears a very natural combination. Independent of the present *control* extension, the sequential FMC has been extended with *non-deterministic* branching for relational computation [2], and *probabilistic* branching to capture probabilistic choice in the λ-calculus [23]. This generalises to an *"algebraically"* branching FMC that fits neatly into the effect handlers paradigm: branching is given by abstract effect operators, which are interpreted computationally by handlers [51]. Interesting questions then arise regarding the relation between the different notions of branching, the "algebraic" approach versus the present *control* extension, and how each encodes effects. These will be considered in future work.

The next three sections will introduce the terms and types of the FMC informally and explore the connections with related work, while the remaining sections will present the formal development. Starting from the λ-calculus, the three extensions *sequencing* (Section 2), *locations* (Section 3), and *control* (Section 4) will be covered in turn, while building up the interpretation of an imperative language complete with store, conditionals, exceptions, and loops. The calculus will be illustrated and motivated through an informal presentation of its operational semantics; the abstract machine itself is given in Section 5 at the start of the formal development. The formulation of *sequencing* and *locations* has matured since the original presentation [20], mainly due to the requirements of the present *control* extension. A preliminary version was presented at MFPS 2024 [21].

2 Sequencing

The Krivine Abstract Machine (KAM) [32], here in a form that uses substitution rather than environments, gives a natural perspective of λ-terms as stack operations, where application *pushes* and abstraction *pops*. To emphasize this view and to accommodate the extensions of the FMC, application $M\,N$ is written $[N].\,M$ ("push N", c.f. Levy's $V`M$ [38]) and abstraction $\lambda x.M$ is written $\langle x\rangle.\,M$ ("*pop* and bind to x"). The operational semantics of a term M acting on a stack S will be depicted as follows.

$$S \overset{M}{\longmapsto\!\!\!\!\twoheadrightarrow}$$

Writing a stack with the head to the right, as $S\,M$, below left is the evaluation of $[N].\,M$, which pushes N to the stack S and continues by evaluating M. On the right, $\langle x\rangle.\,M$ pops the first term N off the stack and substitutes it for x in the remaining computation, as $\{N/x\}M$.

$$[N].\,M : \quad S \xmapsto{[N]} S\,N \overset{M}{\mapsto\!\!\!\twoheadrightarrow} \qquad\qquad \langle x\rangle.\,M : \quad S\,N \xmapsto{\langle x\rangle} S \overset{\{N/x\}M}{\mapsto\!\!\!\twoheadrightarrow}$$

Note that the result of the computation is left implicit. This is because the KAM returns the first term without a transition, which is either a (free) variable or an abstraction with an empty stack, while the FMC takes a different approach: FMC terms return the stack. This is effected by the *sequencing* extension, which introduces imperative *skip* ($\star$) and *sequencing* ($M\,;N$) as the *identity* and *composition* of terms as functions on stacks (or rather *partial* functions, because of non-termination).

$$\star : \quad S \overset{\star}{\mapsto\!\!\!\twoheadrightarrow} S \qquad\qquad M\,;N : \quad R \overset{M}{\mapsto\!\!\!\twoheadrightarrow} S \overset{N}{\mapsto\!\!\!\twoheadrightarrow} T$$

While natural from the perspective of the machine, this notion of sequentiality represents a significant conceptual departure from the traditional λ-calculus. *Skip* indicates *successful termination*, as it does in imperative models, and returns the stack, enabling the view of terms as partial functions on stacks. Sequential composition ($M\,;N$) is then the natural notion of composition, and likewise has its standard imperative meaning, first evaluate M then evaluate N. Both represent a departure from traditional notions of termination and composition in λ-calculus, but are commensurate with sequentiality in cbpv and the computational metalanguage, as will be explored below.

Introducing both constructs into the λ-calculus gives the *sequential λ-calculus*, a fragment of the FMC.

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

The extension gives terms for natural stack operations: e.g. $\langle x\rangle.\,[x].\,[x].\,\star$ duplicates the head of the stack, $\langle x\rangle.\,\star$ deletes it, $\langle x\rangle.\,\langle y\rangle.\,[x].\,[y].\,\star$ exchanges the top two elements, and $\langle x\rangle.\,x$ (the traditional identity term) pops and executes the first stack element. Note that since stacks are last–in–first–out, terms of the form $\langle x_1\rangle.\,.\,\langle x_n\rangle.\,[x_n].\,.\,[x_1].\,\star$, with the order of variables reversed between popping and pushing, return an input stack (of depth at least n) unchanged. By way of example, below is the evaluation of $\langle f\rangle.\,\langle g\rangle.\,(f\,;g)$ which pops and executes the first two stack elements.

$$\langle f\rangle.\,\langle g\rangle.\,(f\,;g) : \quad R\,N\,M \xmapsto{\langle f\rangle} R\,N \xmapsto{\langle g\rangle} R \overset{M}{\mapsto\!\!\!\twoheadrightarrow} S \overset{N}{\mapsto\!\!\!\twoheadrightarrow} T$$

The present formulation of the calculus differs from the original [20,3]. This had the variable construct as a prefix, $x.\,M$, and composition $M\,;N$ as a defined operation, promoting a view of terms as sequences of pushes, pops, and variables. Taking *skip* and *sequence* as primitives is now preferred, for its clear and direct combination of λ-calculus with sequential computation, and as a prerequisite for the *control* extension of this paper. Both formulations are essentially interchangeable: the previous sequential variable $x.\,M$ is a restriction of sequential composition as $x\,;M$, and the previous sequencing operation is implemented here through the reduction relation (see Section 5).

Higher-order stack calculi similar to the sequential λ-calculus have appeared before. Notable are Hasegawa's κ-calculus [19], revisited by Power and Thielecke [53], a study of compiler languages by Douence and Fradet [13], and the *concatenative* programming paradigm [24], meaning higher-order stack languages such as λ-Forth, Joy, and Factor [48]. Typing, explored next, is familiar from several of the above calculi and languages, and from other stack languages like WebAssembly [56].

2.1 Stack typing

Types for the sequential λ-calculus follow the view of terms as functions on stacks. A type is an implication $\overline{\sigma} \Rightarrow \overline{\tau}$ between type vectors $\overline{\sigma}$ and $\overline{\tau}$, which represent the types of the input and output stacks, as below. The empty type vector is ε. The antecedent vector $\overline{\sigma}$ of a type $\overline{\sigma} \Rightarrow \overline{\tau}$ is implicitly reversed, so that identity types $\overline{\sigma} \Rightarrow \overline{\sigma}$ follow the shape of identity terms, $\sigma_1 \dots \sigma_n \Rightarrow \sigma_n \dots \sigma_1$.

$$\sigma,\,\tau \;::=\; \overline{\sigma} \Rightarrow \overline{\tau} \qquad \overline{\sigma},\,\overline{\tau} \;::=\; \tau_1 \dots \tau_n$$

The intuitive meaning of a type assignment $M : \overline{\sigma} \Rightarrow \overline{\tau}$ is then as follows: for any stack S typed by $\overline{\sigma}$ the term M will evaluate successfully and return a stack T typed by $\overline{\tau}$. This is formalized as a reducibility predicate in Section 8 to prove termination of the machine.

$$M : \overline{\sigma} \Rightarrow \overline{\tau} \quad\Longrightarrow\quad \forall S : \overline{\sigma}.\, \exists T : \overline{\tau}.\, S \overset{M}{\longmapsto\!\!\!\!\rightarrow} T$$

Typical of stack typing is further the notion of *stack expansion*: a term $M : \overline{\sigma} \Rightarrow \overline{\tau}$ may also be typed $M : \overline{\sigma}\,\overline{\rho} \Rightarrow \overline{\rho}\,\overline{\tau}$, since if M evaluates with a stack S, it also evaluates with a larger stack $R\,S$ (using juxtaposition for concatenation), leaving R untouched.

$$S \overset{M}{\longmapsto\!\!\!\!\rightarrow} T \quad\Longrightarrow\quad R\,S \overset{M}{\longmapsto\!\!\!\!\rightarrow} R\,T \qquad\qquad M : \overline{\sigma} \Rightarrow \overline{\tau} \quad\Longrightarrow\quad M : \overline{\sigma}\,\overline{\rho} \Rightarrow \overline{\rho}\,\overline{\tau}$$

Semantically, stacks are products of terms, and the calculus forms a strict Cartesian closed category [3], where morphisms are closed terms, identity is $\star : \overline{\tau} \Rightarrow \overline{\tau}$, and composition is $M\,;N : \overline{\rho} \Rightarrow \overline{\tau}$ for $M : \overline{\rho} \Rightarrow \overline{\sigma}$ and $N : \overline{\sigma} \Rightarrow \overline{\tau}$. Other example morphisms are a projection $\langle x\rangle.\, \langle y\rangle.\, [x].\, \star\, : \sigma\tau \Rightarrow \sigma$, a diagonal $\langle x\rangle.\, [x].\, [x].\, \star\, : \tau \Rightarrow \tau\tau$, and a terminal map $\langle x\rangle.\, \star\, : \tau \Rightarrow \varepsilon$. Interestingly, in this interpretation the traditional identity term $\lambda x.x$ becomes *eta/eval* $\langle x\rangle.\, x : (\overline{\sigma} \Rightarrow \overline{\tau})\,\overline{\sigma} \Rightarrow \overline{\tau}$.

Typing is conservative over simple types for the λ-calculus. These embed as *input-only* types $\overline{\tau} \Rightarrow \varepsilon$, with empty output vector, in accordance with the observation that *skip* is needed to yield a return stack. The base type o is interpreted as $\varepsilon \Rightarrow \varepsilon$, and the arrow type $\sigma \to \tau$ adds a further input type σ to the antecedent vector of τ, as given below. The overall picture is given by $\tau_1 \to \dots \to \tau_n \to o = \tau_1 \dots \tau_n \Rightarrow \varepsilon$.

$$o \;=\; \varepsilon \Rightarrow \varepsilon \qquad \sigma \to (\overline{\tau} \Rightarrow \varepsilon) \;=\; \sigma\,\overline{\tau} \Rightarrow \varepsilon$$

Two further observations will be made that set the sequential λ-calculus apart from the λ-calculus. Firstly, all types are inhabited (see [20, Remark 3.7] and Proposition 7.6). In particular, the embedded base type $o = \varepsilon \Rightarrow \varepsilon$ is inhabited by $\star$. Secondly, the calculus has a natural first-order restriction, an internal language for Cartesian categories, as follows (see also [19]). This is expanded to a model of *relational* computation in [2].

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

2.2 Expressing evaluation strategies

Along the same lines as previous higher-order stack calculi [13,53], the sequential λ-calculus embeds Plotkin's cbv λ-calculus [52] and Moggi's computational metalanguage [44]. The key idea in both interpretations is that *values*, respectively *return values*, are returned on the stack. The former then embeds as below left, using subscript $-_{\mathsf{v}}$ to distinguish it from the call–by–name calculus, with $@_{\mathsf{v}}$ for application. This is a typed embedding, with types for values embedded as $\sigma \to_{\mathsf{v}} \tau = \sigma \Rightarrow \tau$ and for computations as $\tau_{\mathsf{v}} = \varepsilon \Rightarrow \tau$. The latter embeds as below right, likewise passing a single return value on the stack, and again this is typed, with $\sigma_1 \to \dots \to \sigma_n \to T(\tau)$ for a monad T embedding as $\sigma_1 \dots \sigma_n \Rightarrow \tau$.

$$\begin{aligned} x_{\mathsf{v}} &= [x].\star & \qquad \mathsf{return}\,M &= [M].\star \\ \lambda_{\mathsf{v}}x.M &= [\langle x\rangle.\, M].\star & \mathsf{let}\,x = M\,\mathsf{in}\,N &= M\,;\langle x\rangle.\, N \\ @_{\mathsf{v}}\,M\,N &= N\,;M\,;\langle x\rangle.\, x \end{aligned}$$

The embedding of $@_{\mathsf{v}}$ evaluates as illustrated below, where N returns the value V and M returns $\lambda y.P$. First, N and M are evaluated, pushing V and $\langle y\rangle.\, P$ to the stack; then $\langle y\rangle.\, P$ is popped, and executed with V as its first argument on the stack, popping V as y and finally evaluating $\{V/y\}P$ to some value

W. Note that this evaluates the argument before the function; the other way around is by the translation taking $@_{\mathsf{v}}\,M\,N$ to $M\,;N\,;\langle y\rangle.\,\langle x\rangle.\,[y].\,x$.

$$N\,;M\,;\langle x\rangle.\,x: \quad S \overset{N}{\mapstochar\twoheadrightarrow} S\,V \overset{M}{\mapstochar\twoheadrightarrow} S\,V\,(\langle y\rangle.\,P) \xmapsto{\langle x\rangle} S\,V \overset{\langle y\rangle}{\mapstochar\twoheadrightarrow} S \overset{\{V/y\}P}{\mapstochar\twoheadrightarrow} S\,W$$

The sequential λ-calculus implicitly features the value/computation distinction of Levy's cbpv [38], with the operand stack holding values. Making these explicit yields the variant of the calculus below, into which cbpv embeds in the same way as the computational metalanguage. It distinguishes *value* terms V and *computation* terms M, mediated by *thunk* and *force* constructs $!M$ and $?V$. The first-order calculus then arises by removing thunks $!M$ as values, leaving only variables, and forced values $?V$ as computations.

$$V,W \;::=\; x \;\mid\; !M \qquad\qquad M,N \;::=\; ?V \;\mid\; \langle x\rangle.\,M \;\mid\; [V].\,M \;\mid\; \star \;\mid\; M\,;N$$

In the monadic setting and cbpv, sequencing is modelled by the *return*- and *let*-constructs, which pass a single return value. The translation interprets these as *skip* and *sequence*, with the return value passed on the stack. All three calculi thus express essentially the same notion of *sequentiality*. This is made clearer still with the monadic *bind* notation of Haskell, which is interpreted directly as $M \gg\!= N = M\,;N$. The sequential λ-calculus may thus be viewed as a generalization of cbpv where sequencing passes the entire argument stack instead of a single value. This interpretation, that $\star$ passes all return values rather than no values, is expressed in the reduction semantics by the following rules.

$$([M].\,N)\,;P \rightarrow [M].\,(N\,;P) \qquad\qquad (\langle x\rangle.\,M)\,;N \rightarrow \langle x\rangle.\,(M\,;N) \quad (x \notin \mathsf{fv}(N))$$

$$R \xmapsto{[M]} R\,M \overset{N}{\mapstochar\twoheadrightarrow} S \overset{P}{\mapstochar\twoheadrightarrow} T \qquad\qquad R\,P \xmapsto{\langle x\rangle} R \overset{\{P/x\}M}{\mapstochar\twoheadrightarrow} S \overset{N}{\mapstochar\twoheadrightarrow} T$$

These implement the idea that sequencing and prefixing in push- and pop-actions express the same notion of sequentiality, in accordance with the operational semantics. Together with standard β-reduction $[M].\,\langle x\rangle.\,N \rightarrow \{M/x\}N$ and the rule $\star\,;M \rightarrow M$, the interpretation of a *let*-redex then reduces as follows. This demonstrates how values pushed before $\star$ are passed on to the next computation.

$$\mathsf{let}\,x = \mathsf{return}\,M\,\mathsf{in}\,N \quad = \quad ([M].\,\star)\,;\langle x\rangle.\,N \;\rightarrow\; [M].\,(\star\,;\langle x\rangle.\,N) \;\rightarrow\; [M].\,\langle x\rangle.\,N \;\rightarrow\; \{M/x\}N$$

Like the computational metalanguage and cbpv, the sequential λ-calculus may be may be extended with effect operators as well as constants, primitive operations, exceptions, etc. to give a model of higher-order computation with effects. The FMC however takes a different approach. Instead of introducing new primitives for effects, the machine and the existing constructs are generalised in subtle and minimal ways, so that effects and control flow may be modelled while preserving confluent reduction and typeability. The two generalisations, *locations* and *control*, are explored next.

3 Locations

To capture effects, the machine is generalised from one operand stack to multiple, named stacks, indexed in a global set of *locations* $A = \{\lambda, a, b, c, \dots\}$, with λ indicating the original or *default* stack. Pop- and push-actions are parameterised in a location to operate on the designated stack, as $a\langle x\rangle.\,M$ and $[N]a.\,M$. The default location λ will be omitted from the notation to retain the previous $\langle x\rangle.\,M$ and $[N].\,M$. This constitutes the FMC as previously published [20] (modulo the choice of primitives for sequencing).

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

Terms now operate on a family of stacks $S_A = \{S_a \mid a \in A\}$ called a *memory*, where only a finite number of stacks are non-empty. Since a given term uses only a fixed, finite number of locations, in each particular case A itself may be considered finite. For easier manipulation, a memory will be written as a sequence of terms indexed by locations, $a(M)$, where terms on different locations may permute:

$$S_A \ ::= \ a_1(M_1)\dots a_n(M_n) \qquad \text{where} \ \ a(M)\,b(N) = b(N)\,a(M) \ \ \text{if} \ \ a \neq b$$

Pushing M to the stack S_a in the memory S_A is then written $S_A\,a(M)$. The operational semantics of the indexed push- and pop-actions is as follows.

$$[N]a.\,M : \quad S_A \xmapsto{[N]a} S_A\,a(N) \overset{M}{\mapsto\!\!\!\twoheadrightarrow} T_A \qquad\qquad a\langle x\rangle.\,M : \quad S_A\,a(N) \xmapsto{a\langle x\rangle} S_A \overset{\{N/x\}M}{\mapsto\!\!\!\twoheadrightarrow} T_A$$

The reduction semantics expresses the independence of stacks on different locations by the following two rules. The first is the regular β-step from λ-calculus, when a push meets a pop on the same location. The second, *passage* rule gives the case when the locations are distinct, resolved by permuting both operations past each other, in search of a counterpart that does match. (Note that the positioning of the location label indicates on which side the operation interacts.)

$$[N]a.\,a\langle x\rangle.\,M \ \rightharpoonup \ \{N/x\}M \qquad\qquad [N]b.\,a\langle x\rangle.\,M \ \rightharpoonup \ a\langle x\rangle.\,[N]b.\,M \quad (a \neq b,\ x \notin \mathsf{fv}(N))$$

Instead of the above two rules, the original presentation of the FMC [20] featured a single rule that allowed matching push- and pop-actions to interact *at a distance*, ignoring other actions in between. The present formulation is now preferred: the rules are simpler, they give better normal forms where pop-actions precede push-actions, and they give a better equational theory, as follows. With η-equivalence $M \sim a\langle x\rangle.\,[x]a.\,M$ where $x \notin \mathsf{fv}(M)$, reduction equivalence gives the following equations, completing the intuitive idea that push- and pop-actions on distinct stacks are interchangeable.

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

This equational theory then captures the algebraic theory for global store of Plotkin and Power [50], via the encoding of store given below (see [20]).

Typing generalises along the same lines as memories, from one to many type vectors for input and output, indexed in the set of locations A. A *memory type*, written $\overline{\overline{\tau}}$, is a family of type vectors in A, and like a memory, may be written as a sequence modulo permutation on distinct locations, as below. A type is then an implication between memory types, $\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}$. The empty memory type is written ε, and as with terms, the main location λ may be omitted, so that the notation is conservative over that for the sequential λ-calculus.

$$\sigma,\tau \ ::= \ \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}} \qquad\qquad \overline{\overline{\tau}} \ ::= \ a_1(\tau_1)\dots a_n(\tau_n) \quad \text{where} \ \ a(\sigma)\,b(\tau) = b(\tau)\,a(\sigma) \ \ \text{if} \ \ a \neq b$$

3.1 Modelling effects

Stacks model higher-order store, input/output, and probablistic choice as follows.

$$\begin{aligned} c \coloneqq M \ &= \ M\,;\langle x\rangle.\,c\langle \_\rangle.\,[x]c.\,\star :\ c(\tau) \Rightarrow c(\tau) &\qquad \mathsf{print}\,M \ &= \ M\,;\langle x\rangle.\,[x]\mathsf{out}.\,\star :\ \varepsilon \Rightarrow \mathsf{out}(\tau) \\ !c \ &= \ c\langle x\rangle.\,[x]c.\,[x].\,\star :\ c(\tau) \Rightarrow c(\tau)\,\tau & \mathsf{read} \ &= \ \mathsf{in}\langle x\rangle.\,[x].\,\star :\ \mathsf{in}(\tau) \Rightarrow \tau \\ & & \mathsf{sample} \ &= \ \mathsf{rnd}\langle x\rangle.\,[x].\,\star :\ \mathsf{rnd}(\tau) \Rightarrow \tau \end{aligned}$$

For higher-order store, a mutable variable c is represented by a location, with *update* $c \coloneqq M$ and *lookup* $!c$ as below left. Updating evaluates M, leaving the result N on the main stack; this is popped as x; then c is cleared by popping and discarding the value P, where the underscore ($\_$) represents a variable that may not occur; and finally N (substituted for x) is pushed to c.

$$S_A\,c(P) \overset{M}{\mapsto\!\!\!\twoheadrightarrow} S_A\,c(P)\,\lambda(N) \xmapsto{\langle x\rangle} S_A\,c(P) \xmapsto{c\langle \_\rangle} S_A \xmapsto{[N]c} S_A\,c(N)$$

Lookup $!c$ pops the value from c as x; restores it by pushing; and returns it on the main stack. Note that these encodings are familiar from Concurrent Haskell's MVars [49]. The stack for c is assumed to hold at most one term, but this need not be enforced explicitly since the encoding of the operations preserves it.

Input/output is modelled by two dedicated locations in and out, with *read* and *print* operations as above. The in stack should allow only to *pop* but not *push*, and out only to push, which is again maintained by the encoding. Probabilistic choice is similar to input, with a location rnd representing a random number generator. The embeddings give a call–by–value semantics, returning a value to the stack in the case of lookup, read, and sample, and in the case of update and print first evaluating the argument $M : \varepsilon \Rightarrow \tau$, which leaves a single value of type τ on the main stack.

Untyped, the FMC provides an operational semantics to effect operators, and is able to express their call–by–name as well as their call–by–value interpretation. The simple types imposed by the translation are an innovation of the FMC, and agree with known *intersection types* for store [10,12,22]. Typing for store is further similar to the state monad $T(\tau) = \sigma \to (\sigma \times \tau)$, but parameterized in a location c, with the type for lookup as $c(\sigma) \Rightarrow c(\sigma)\,\tau$ and that for update as $c(\sigma) \Rightarrow c(\sigma)$, corresponding to $T(1) = \sigma \to (\sigma \times 1)$. Notable however is that extending a computation with a store c in the FMC is not given by a transformation from τ to $c(\sigma) \Rightarrow c(\sigma)\,\tau$, but from $\overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}$ to $c(\sigma)\,\overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}\,c(\sigma)$. It is thus not a state monad in the traditional sense, which is one reason why the FMC avoids the compositionality problem of the monadic approach. A final observation here is that for higher-order store, typing mutable variables themselves, as $c : \sigma$, does not guarantee termination, since it allows Landin's Knot [34], the fixed-point combinator $c \coloneqq F(!c)\,;\,!c$, to be typed. The above type systems (intersection types, the state monad, and FMC typing) avoid this by typing both the input and output effect of store operations, which does enforce termination properties.

The locations of the FMC are akin to the *channels* of process calculi [43], as demonstrated by the encoding of input/output, and indeed channels are used in concurrency theory to encode store [26,55]. Future work will consider locations as channels for a message-passing concurrent FMC. Locations are also similar to the μ-variables of $\lambda\mu$-calculus [47], which likewise are names for stacks [60]. However, $\lambda\mu$-calculus abstracts over these, reifing stacks as values, a sufficiently different mechanism to the FMC that neither calculus readily encodes the other.

4 Control

The contribution of the present paper is to extend the FMC to branching and looping computation, enabling the embedding of control flow operations including conditionals, exception handling, and loops. *Skip* ($\star$), signifying successful termination, is generalized to a set of *choices* $\{\star, i, j, k, \dots\}$ which name the potential branches of a computation, similar to *exceptions* or *exit codes*. Sequential composition ($M\,;N$) is generalized accordingly, to a *case* $M\,;i\!\to\!N$ which composes only on the chosen branch i: it first evaluates M, and if this exits with i it continues with N, discarding it otherwise. Again for syntactic conservativity, sequencing is defined as $M\,;N = M\,;\,\star\to\!N$. Finally, a *loop* construct M^i is introduced, which repeats M for as long as it exits with i, and terminates otherwise. The full calculus is then as follows.

$$M, N \;::=\; x \;\mid\; [N]a.\,M \;\mid\; a\langle x\rangle.\,M \;\mid\; i \;\mid\; M\,;i\!\to\!N \;\mid\; M^i$$

In the operational semantics computations return a memory together with a choice. *Choice*, *case*, and *loop* terms evaluate as follows, where $i \neq j$.

$$i :\quad S_A \overset{i}{\longmapsto\!\!\!\!\twoheadrightarrow} S_A, i \qquad M\,;i\!\to\!N : \begin{cases} R_A \overset{M}{\longmapsto\!\!\!\!\twoheadrightarrow} S_A, i \overset{N}{\longmapsto\!\!\!\!\twoheadrightarrow} T_A, k \\ R_A \overset{M}{\longmapsto\!\!\!\!\twoheadrightarrow} S_A, j \end{cases} \qquad M^i : \begin{cases} R_A \overset{M}{\longmapsto\!\!\!\!\twoheadrightarrow} S_A, i \overset{M^i}{\longmapsto\!\!\!\!\twoheadrightarrow} T_A, k \\ R_A \overset{M}{\longmapsto\!\!\!\!\twoheadrightarrow} S_A, j \end{cases}$$

4.1 Expressing control flow

Constructions for control flow are embedded as follows. Below left, the *choice* and *case* terms themselves are the standard *throw* and *try/catch* of exception handling, with an exception e as a choice label. Below right, the boolean constants are encoded as choice terms pushed onto the stack. A conditional evaluates the condition, pops the (expected) boolean from the stack, and uses it as a choice to select the matching branch. Note that *case* associates left, $M\,;i\to N\,;j\to P = (M\,;i\to N)\,;j\to P$.

$$\begin{array}{rclrcl} \mathsf{throw}\,e &=& e & \top\,,\,\bot &=& [\top].\star\;,\;\;[\bot].\star \\ \mathsf{try}\,M\,\mathsf{catch}\,e\,N &=& M\,;e\to N & \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N &=& B\;;\;\langle x\rangle.\,x\;;\;\top\to M\;;\;\bot\to N \end{array}$$

The encoding of the booleans gives the general pattern for modelling constants and primitive operations. Primitive data such as bounded integers are modelled by a finite set of choices $\{c_1,\dots,c_n\}$, each representing a constant. A constant as an expression is encoded by pushing the choice to the stack, and functions on contants such as addition and multiplication are constructed from case switches, interpreted as below.

$$c_i \;=\; [c_i].\star \qquad \mathsf{case}\,M\,\mathsf{of}\,\{c_1\mapsto N_1,\dots,c_n\mapsto N_n\} \;=\; M\,;\langle x\rangle.\,x\,;c_1\to N_1\,;\,\dots\,;c_n\to N_n$$

These interpretations impose a call–by–value semantics on the encoded constructs, which follows the principle that *exceptions* are choices as *computations*, while *constants* are choices as *values*. The operational behaviour of the encoding is correct as long as constants and exceptions are modelled by distinct choice labels; then in the above interpretation of the case switch, an exeption in some N_k cannot match a later constant c_m.

The *loop* construct M^i encodes iteration, as expressed by its operational semantics and its rewrite rule, $M^i \twoheadrightarrow M\,;i\to M^i$. The use of the *case* construct to model iteration means that if M exits with any other choice $j\neq i$ this terminates the loop. Exceptions as choices in M then behave as expected. Escape clauses such as *break* and *return* may likewise be modelled by choices, to be caught outside the loop. Where M^i is naturally a *"do M while i"* loop, the escape mechanism allows easy *while-do* loops as well. In the encodings below right, the *do–while* loop repeats for the $\top$-choice of the boolean B, while the *while–do* loop repeats if M terminates correctly with $\star$. Both catch the $\bot$-choice of B and a break-choice as correct loop escapes (but omit to catch a *return* clause).

$$\begin{array}{rclrcl} \mathsf{break} &=& \mathsf{break} & \mathsf{do}\;M\;\mathsf{while}\;B &=& (M\;;\;B\;;\;\langle x\rangle.\,x)^{\top}\;;\;\bot\to\star\;\;;\;\mathsf{break}\to\star \\ \mathsf{return}\;M &=& [M].\,\mathsf{return} & \mathsf{while}\;B\;\mathsf{do}\;M &=& (B\;;\;\langle x\rangle.\,x\;;\;\top\to M)^{\star}\;;\;\bot\to\star\;\;;\;\mathsf{break}\to\star \end{array}$$

4.2 Control types

The static semantics of the new constructs is that of a choice-indexed sum: computation may follow a finite number of possible branches, each labelled with a choice and returning a different memory. Typing is adjusted by changing output types to sums of memory types, as follows. The notation mirrors the syntax of terms ending in a choice, and sum types are considered moduly symmetry.

$$\sigma,\tau \;::=\; \overline{\overline{\sigma}}\Rightarrow\overline{\overline{\tau}}_I \qquad\qquad \overline{\overline{\tau}}_I \;::=\; \overline{\overline{\tau}}_{i_1}.i_1+\dots+\overline{\overline{\tau}}_{i_n}.i_n \quad\text{where}\quad I=\{i_1,\dots,i_n\}$$

The meaning of a typing judgement $M:\overline{\overline{\sigma}}\Rightarrow\overline{\overline{\tau}}_I$ is that given an input memory $S_A:\overline{\overline{\sigma}}$ evaluation returns a memory $T_A:\overline{\overline{\tau}}_i$ for some $i\in I$ — *if* it terminates, which is guaranteed in the absence of the loop construct, but not in its presence. Types thus guarantee *progress* in the general case (Proposition 7.5) and *termination* in the loop-free case (Theorem 8.4). The interpretation of types supports the following two properties. Below left is *sum expansion*: a term with return type $\overline{\overline{\sigma}}_I$ may also be typed with any larger return type $\overline{\overline{\sigma}}_I+\overline{\overline{\tau}}_J$, where the sum notation implicitly assumes $I\cap J=\varnothing$. Below right is *stack expansion*: as before, a term that evaluates with a memory $\overline{\overline{\rho}}$ may also use a larger memory $\overline{\overline{\sigma}}\,\overline{\overline{\rho}}$, leaving the additional memory $\overline{\overline{\sigma}}$ unchanged in each branch of the computation. Implementing this, the notation $\overline{\overline{\sigma}}\,\overline{\overline{\tau}}_I$ prefixes the memory type $\overline{\overline{\sigma}}$ to each $\overline{\overline{\tau}}_i$ in $\overline{\overline{\tau}}_I$, taking the summand $\overline{\overline{\tau}}_i.i$ to $\overline{\overline{\sigma}}\,\overline{\overline{\tau}}_i.i$. Since stacks are products this represents the familiar distribution law $A\times(B+C)=(A\times B)+(A\times C)$.

$$M:\overline{\overline{\rho}}\Rightarrow\overline{\overline{\sigma}}_I \implies M:\overline{\overline{\rho}}\Rightarrow\overline{\overline{\sigma}}_I+\overline{\overline{\tau}}_J \qquad\qquad M:\overline{\overline{\rho}}\Rightarrow\overline{\overline{\tau}}_I \implies M:\overline{\overline{\rho}}\,\overline{\overline{\sigma}}\Rightarrow\overline{\overline{\sigma}}\,\overline{\overline{\tau}}_I$$

The constructors are typed as follows. Choice terms are injections, typed $i : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\sigma}}.i + \overline{\overline{\tau}}_J$, a type which may be obtained from $i : \varepsilon \Rightarrow \varepsilon.i$ by stack and sum expansion. A case $M \,;\, i \to N$ is typed as follows:

$$M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_J + \overline{\overline{\sigma}}.i \quad \text{and} \quad \left\{ \begin{array}{ll} N : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_J & \text{or} \\ N : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_J + \overline{\overline{\upsilon}}.i & \end{array} \right\} \quad \Longrightarrow \quad \left\{ \begin{array}{ll} M \,;\, i \to N : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_J & \text{or} \\ M \,;\, i \to N : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_J + \overline{\overline{\upsilon}}.i & \end{array} \right.$$

That is, M must exit on i with an appropriate memory $\overline{\overline{\sigma}}$ to serve as input for N; for any other choice $j \in J$ the terms M and N must exit with the same type $\overline{\overline{\tau}}_j$, both matching a potential later case $j \to P$; and N may or may not exit with a choice i and type $\overline{\overline{\upsilon}}$. The merger of both types $\overline{\overline{\tau}}_J$ represents the co-diagonal of the sum, $A + A \to A$.

Composition is made flexible by stack and sum expansion, as follows. First, evaluating M may provide fewer or more arguments on the stack than used by N, with the difference made up with stack expansion for M or N respectively. Second, the return choices for M and N need not coincide, or even overlap: as long as their types agree for the choices they have in common, they can be sum-expanded to match.

A loop M^i, which repeats on i, expects a type $M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\sigma}}.i + \overline{\overline{\tau}}_J$ where the input type coincides with the output type for i. The loop itself is then typed $M^i : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_J$, omitting the choice i, as it exits on the choices in J. This gives the expected typing pattern of iteration, which is to take $f : A \to A + B$ to $\mathsf{iter}\, f : A \to B$ [7]. An interesting consequence is what happens with iteration on a term with only a single branch. This results in a sum type where J is empty, the empty sum or *void* type 0. This is not ruled out in principle, and may happen for example with the loop $i^i : \varepsilon \Rightarrow 0$ for $i : \varepsilon \Rightarrow \varepsilon.i$. Types $\overline{\overline{\sigma}} \Rightarrow 0$ returning void are thus inhabited by terms looping on their only output branch, which are guaranteed non-terminating.

The type system imposes typing on the embedded constructions, including exceptions, constants, case switches, and while-loops. The type for booleans may be defined as $\mathbb{B} = \varepsilon \Rightarrow \varepsilon.\bot + \varepsilon.\top$, which correctly gives $\bot : \mathbb{B}$ and $\top : \mathbb{B}$. The type for the conditional is built up as follows, for a simple case where B returns only a boolean and M and N return a single value of type τ. The term B returns a boolean on the main stack λ for the default choice $\star$. This is picked up and executed by $\langle x \rangle . \, x$, giving $B \,;\, \langle x \rangle . \, x$ the type $\mathbb{B}$. Then the case $\top \to M$ composes on the $\top$-branch, returning τ on $\star$ instead, and $\bot \to N$ composes on the $\bot$-branch, also returning τ for $\star$ and merging both branches.

$$B : \varepsilon \Rightarrow \mathbb{B}.\star \qquad \langle x \rangle . \, x : \mathbb{B} \Rightarrow \varepsilon.\bot + \varepsilon.\top \qquad M : \varepsilon \Rightarrow \tau.\star \qquad N : \varepsilon \Rightarrow \tau.\star$$

$$\mathsf{if}\, B \,\mathsf{then}\, M \,\mathsf{else}\, N \;=\; B \,;\, \langle x \rangle . \, x \,;\, \top \to M \,;\, \bot \to N : \varepsilon \Rightarrow \tau.\star$$

Types for constants such as bounded integers use this general pattern, as $\varepsilon \Rightarrow \varepsilon.i_1 + \ldots + \varepsilon.i_n$ for constants $\{i_1, \ldots, i_n\}$. Continuing the above example, if B may throw an exception e, with the type $\varepsilon \Rightarrow \mathbb{B}.\star + \varepsilon.e$, or one or both of M or N with type $\varepsilon \Rightarrow \tau.\star + \varepsilon.e$, this is seamlessly integrated into the types to give the conditional the type $\varepsilon \Rightarrow \tau.\star + \varepsilon.e$ as well. The following example will demonstrate how store integrates with these constructs, and how a *while-do* loop is typed.

Example 4.1 Below is the interpretation and typing of the following imperative program.

$$\mathsf{while}\; !a < 5 \;\mathsf{do}\; a \coloneqq !a + 1$$

It is broken up as $\mathsf{while}\; B \;\mathsf{do}\; N$ where $B = !a < 5$, $N = a \coloneqq M$ and $M = !a + 1$. Let $\mathbb{I}$ be a type for bounded integers and assume terms for addition $+ : \mathbb{I}\,\mathbb{I} \Rightarrow \mathbb{I}.\star$ and inequality $< \; : \mathbb{I}\,\mathbb{I} \Rightarrow \mathbb{B}.\star$ modelled by case switches. The interpretation of an expression $P + Q$ is its translation into reverse Polish notation, $P \,;\, Q \,;\, +$, standard for a stack machine implementation. Then M is interpreted as below, where $[1]. \star : \varepsilon \Rightarrow \mathbb{I}.\star$ is stack-expanded with the memory type $a(\mathbb{I})\,\mathbb{I}$ to allow the composition, and $+ : \mathbb{I}\,\mathbb{I} \Rightarrow \mathbb{I}.\star$ with $a(\mathbb{I})$. Note that this implicit expansion is the equivalent of lifting into the state monad in the monadic approach.

$$a\langle x \rangle . \, [x]a . \, [x] . \, \star \, : a(\mathbb{I}) \Rightarrow a(\mathbb{I})\,\mathbb{I}.\star \qquad [1] . \, \star \, : \mathbb{I}\, a(\mathbb{I}) \Rightarrow a(\mathbb{I})\,\mathbb{I}\,\mathbb{I}.\star \qquad + : \mathbb{I}\,\mathbb{I}\, a(\mathbb{I}) \Rightarrow a(\mathbb{I})\,\mathbb{I}.\star$$

$$M \;=\; !a + 1 \;=\; a\langle x \rangle . \, [x]a . \, [x] . \, \star \,;\, [1] . \, \star \,;\, + \; : a(\mathbb{I}) \Rightarrow a(\mathbb{I})\,\mathbb{I}.\star$$

Next, the term N composes this with inputs on a and λ, returning on a, and B is similar to M.

$$\begin{array}{rcrcl} N & = a \mathrel{:=} !a+1 & = & M\,;\langle y\rangle.\,a\langle\_\rangle.\,[y]a.\star : & a(\mathbb{I}) \Rightarrow a(\mathbb{I}).\star \\ B & = \quad !a < 5 & = & a\langle x\rangle.\,[x]a.\,[x].\,\star\,;[5].\,\star\,;\,<\, : & a(\mathbb{I}) \Rightarrow a(\mathbb{I})\,\mathbb{B}.\star \end{array}$$

The loop $\mathsf{while}\ B\ \mathsf{do}\ N$ translates to $(B\,;\langle x\rangle.\,x\,;\top\!\to\! N)^\star\,;\bot\!\to\!\star$, which is built up below. First, to compose with B, the term $\langle x\rangle.\,x : \mathbb{B} \Rightarrow \varepsilon.\bot + \varepsilon.\top$ is lifted with $a(\mathbb{I})$. The second line composes B with $\top\!\to\! N$, and the third line gives the loop on the default choice $\star$, which is well typed since the output type $a(\mathbb{I})$ matches the input. The final step adds the catching of $\bot$, giving the final type of the program as an update on a.

$$\begin{array}{rrcl} & \langle x\rangle.\,x & : & \mathbb{B}\,a(\mathbb{I}) \Rightarrow a(\mathbb{I}).\bot + a(\mathbb{I}).\top \\ & B\,;\langle x\rangle.\,x\,;\top\!\to\! N & : & a(\mathbb{I}) \Rightarrow a(\mathbb{I}).\bot + a(\mathbb{I}).\star \\ & (B\,;\langle x\rangle.\,x\,;\top\!\to\! N)^\star & : & a(\mathbb{I}) \Rightarrow a(\mathbb{I}).\bot \\ \mathsf{while}\ !a < 5\ \mathsf{do}\ a \mathrel{:=} !a+1 & = (B\,;\langle x\rangle.\,x\,;\top\!\to\! N)^\star\,;\bot\!\to\!\star & : & a(\mathbb{I}) \Rightarrow a(\mathbb{I}).\star \end{array}$$

The above demonstrates how the type system of the FMC carries over to the embedded imperative programming language, complete with higher-order store, input/output, probabilistic choice, constants, conditionals, exceptions, and iteration with escape. The properties of the type system, *progress* (Proposition 7.5) and *termination* in the absence of loops (Theorem 8.4), apply. Concretely, since types make all potential branches of a computation explicit, *progress* guarantees that exceptions are caught and loop escapes handled correctly. The embedding is naturally compatible with that of the call–by–value λ-calculus, which likewise returns values to the main stack, giving the embedding of an ML-like language.

As a final consideration it will be shown how algebraic (non-inductive) datatypes with a call–by–name semantics embed. With the cbn λ-calculus and recursive definitions, the latter omitted here, this forms the core of the Haskell programming language. Data constructors embed as choice terms, but not returned on the stack as in the call–by–value interpretation. Following the standard cbn interpretation, a fully applied constructor i thus leaves its data on the stack and exits with choice i, as below. A case switch on a datatype consists of a series of *pattern matching* cases $i\,x_1\ \ldots\ x_m \to N$ which bind the arguments M_k of the datatype to the variables x_k for use in N; the notation below uses vectors of variables $\overline{x}$ for the parameters of each case. In the FMC interpretation the arguments are passed on the stack, so that pattern-matching may be given by abstractions, where the notation uses $\langle\overline{x}\rangle$ for $\langle x_1\rangle.\,.\langle x_n\rangle$. The computation N_i for the matching case is then pushed to the stack, since evaluating it might incorrectly trigger a later case, and is popped and evaluated by $\langle x\rangle.\,x$ after the case switch is completed.

$$\begin{array}{rcl} i\,M_1\ \ldots\ M_m & = & [M_m]\ldots[M_1].\,i \\ \mathsf{case}\,M\,\mathsf{of}\ \{i_1\,\overline{x}_1 \to N_1,\ldots,i_n\,\overline{x}_n \to N_n\} & = & M\ ;\ i_1\!\to\!\langle\overline{x}_1\rangle.\,[N_1].\,\star\ ;\ \ldots\ ;\ i_n\!\to\!\langle\overline{x}_n\rangle.\,[N_n].\,\star\ ;\ \langle x\rangle.\,x \end{array}$$

Typing is as follows. A datatype definition as below left, using Haskell's `data` keyword, creates a new type σ as a sum–of–products indexed by constructors. The FMC interprets the type σ directly as a sum type.

$$\mathtt{data}\ \sigma\ =\ i_1\,\overline{\tau}_1 \mid \cdots \mid i_n\,\overline{\tau}_n \qquad \Longrightarrow \qquad \sigma\ =\ \varepsilon \Rightarrow \overline{\tau}_1.i_1 + \ldots + \overline{\tau}_n.i_n$$

The embedding of datatypes is then a typed embedding, extending that of the call–by–name λ-calculus. A typed constructor is interpreted as below, where i is some i_k of the above datatype with $\overline{\tau}_k = \tau_1 \ldots \tau_m$, so that it has type σ when fully applied.

$$i : \tau_1 \to \ldots \to \tau_m \to \sigma\ =\ i : \tau_1 \ldots \tau_m \Rightarrow \overline{\tau}_1.i_1 + \ldots + \overline{\tau}_n.i_n \qquad [M_m]\ldots[M_1].\,i : \sigma$$

The case switch on $M : \sigma$ requires all terms N_i to share the same type ρ, and will then return ρ itself.

The overall type ρ for the case switch is composed in the following way.

$$M : \sigma \qquad \overline{x}_i : \overline{\tau}_i \qquad N_i : \rho \qquad \langle \overline{x}_i \rangle . [N_i] . \star : \overline{\tau}_i \Rightarrow \rho.\star$$

$$M \; ; \; i_1 \to \langle \overline{x}_1 \rangle . [N_1] . \star \; ; \; \dots \; ; \; i_n \to \langle \overline{x}_n \rangle . [N_n] . \star \; ; \; \langle x \rangle . x : \rho$$

4.3 Discussion and related work

The present discussion will compare with related type systems and semantics for branching computation, in particular with typed exceptions. These have been studied in various forms, with subtle syntactic and semantic differences that the discussion will attempt to clarify.

The FMC takes the *exceptions–as–values* approach [1,27,59,66], which follows the idea that computations return either a value or an exception. This casts exceptions as *coproducts*, characterized by the exception monad $TX = E + X$ for a set of exceptions E. Going back to Gentzen's natural deduction [17], coproducts in λ-calculus have been handled with a case distinction on each summand, as below.

$$\dfrac{M : A + B \qquad \begin{array}{c}[x : A] \\ \vdots \\ N : C\end{array} \qquad \begin{array}{c}[y : B] \\ \vdots \\ P : C\end{array}}{\mathsf{case}\, M\, \mathsf{of}\, \{\mathsf{inl}\, x \to N, \mathsf{inr}\, y \to P\} : C}$$

Benton and Kennedy's *exception handlers* [4] adapt case switches to be *fall-through*, capturing the return value and some (but not necessarily all) exceptions, with uncaught exceptions being passed through. This appears crucial to a compositional treatment of exceptions, by macro expansion. Otherwise, the encoding of a catch-construct $\mathsf{try}\, M\, \mathsf{catch}\, e\, N$ depends on the potential exceptions $e_1 \dots e_n$ that M might throw:

$$\mathsf{case}\, M\, \mathsf{of}\, \{\mathsf{return}\ x \to \mathsf{return}\ x,\ e_1 \to e_1,\ \dots,\ e_n \to e_n,\ e \to N\}$$

Exception handlers feature a clear notion of *sequentiality*, and are implemented through a *continuation stack* similar to the one that will be used for the FMC in Definition 5.2. They are introduced into cbpv in [39], and later generalized by Plotkin and Pretnar to *effect handlers* [51], a general model of branching computation that will be discussed below.

The FMC features two key adaptations to the above. First, pattern-matching is avoided by passing values on the stack. Second, by unifying *sequencing* and *exceptions*, case switches may become not only *fall-through* but also *single-case*, since multiple cases can now be handled sequentially. Both changes are made possible, at least conceptually, by the model of the FMC as a sequential stack-based calculus.

A different approach to exceptions uses classical *continuations* [6,9,11,28,36,46,64], which give a typed λ-calculus with continuation operators such as call/cc [18] via the correspondence with classical logic. Continuation operators have a natural interpretation also on the Krivine machine [8,60], where they *reify* the stack, allowing to pass an entire stack as an argument. This is a different mechanism to how the FMC extends the KAM, not easily simulated in either direction. From an operational perspective, exceptions and continuations are incomparable constructs: neither can macro-express the other, not even in the presence of types or state [33,54].

A separate but related notion are *jumps*, which like the present *control* extension, model branching and looping computation. Low-level languages, where jumps are a basic construct, may be typed [57]. In a more general setting, Fiore and Staton model jumps through *explicit substitutions* [14], and Maurer, Downen, Ariola, and Peyton Jones introduce jumps as explicit substitutions to optimize case switches in the Haskell compiler [42]. Syntactically, an explicit substitution $M[N/x]$ is very similar to a *case* $M \,;\, i \to N$, with the only difference that the former uses a *variable* and the latter a *choice* (a constant). Maurer et al. demonstrate their appeal as a *single-case* construct, and further allow loops. This was a major inspiration for the present approach, and earlier drafts of this work adopted their terminology of *jump* and *join* [21], though this was changed to *choice* and *case* for the following considerations.

There are key distinctions between explicit substitutions, jumps, and exceptions. First, the continuation N in $M[N/x]$ may be duplicated, where both forward program jumps and exception handling are *affine*: the continuation is used or discarded, but not duplicated. Fiore and Staton model jumps by making explicit substitutions affine through syntactic restrictions, as do Maurer et al., who restrict the target variables of an explicit substitution to *head* position. Second, explicit substitutions and jumps are *static*: the connection between the jump (the variable) and the continuation is determined by the term structure, before evaluation. Exceptions however are *dynamic*: the choice whether a continuation is used or discarded is made only during evaluation. This is the key distinction between variables and constants, and is reflected also in the types for both constructs, where jumps and explicit substitutions are generally typed with type arrows as intuitionistic continuations, but exceptions as coproducts.

Plotkin and Pretnar's *effect handlers* [29,41,51,67] are a general model of branching computation. Built over two layers, cbpv as a model of sequential computation is extended with *effect operator symbols*, that create symbolic branching points in the computation tree. A layer of *effect handlers* then interprets these branching points operationally to model effects. This treatment of branching is sufficiently different from the present *control* extension that it is not directly obvious which concept in one model maps onto which concept in the other.

Following the origin of effect handlers in exception handlers, one view would be of operators as a generalisation of *raising* an exception, analogous to an FMC *choice*, and handlers as *catching* it, analogous to *case*. Handlers and cases have in common that they are matched *dynamically*, but there are three key distinctions. First, handlers are generally defined globally, separate from computation terms, where the FMC has cases within the term language. Second, the standard *deep* handlers apply to multiple operators in sequence, and thus are not *affine*, though *shallow* handlers are [25]. Third, like exception handlers, effect handlers require a return case as well as catching effects, and are thus *fall-through* but not *single-case*.

Taking a different perspective, like cbpv the sequential λ-calculus is a model of higher-order sequential computation into which effect operators and handlers may be introduced. This is a present line of investigation, but preliminary observations suggest that this forms a natural combination: the presence of sequential composition as a primitive, enabled by passing arguments on the stack, means the use of continuations may be avoided. This not only simplifies the model but moreover allows the characteristic algebraic laws to be enforced by the syntax, rather than imposed externally. A further line of enquiry is how effect handlers relate to *locations*, noting that both model specific effects, in particular *store*. These observations serve to underline that both models, the FMC and handlers, complement each other in interesting ways.

To summarise, the overall technical contributions are as follows. First, the new *choice* and *case* constructs of the FMC, in the termilogy used here, combine the *single-case* nature of explicit substitutions with the *affine*, *dynamic* coproduct semantics of the exceptions–as–values paradigm. Second, inherited from the first iteration of the FMC, is the passing of values through (multiple, independent) stacks, which reduces pattern-matching to case-matching. Third, at a higher level, these two contributions allow the FMC to *unify* (rather than *combine*) the concepts of sequential composition, exception handling, and case switches, and seamlessly integrate them with the effects encoded through *locations*. Fourth, again derived from the previous, is the type system, which gives a natural coproduct semantics to these constructions (to be confirmed formally in future work), where approaches through explicit substitution inherit continuation-style typing. Finally, the treatment of *loops* with escape, enabled by the design of the choice and case constructs, appears novel.

The remaining sections will present the formal development of the calculus: its operational semantics, reduction relation, confluence, types, and machine termination and normalization of the typed calculus. Its denotational semantics will be the subject of future work.

The FMC as presented here covers a complete *theoretical* functional–imperative language. In practice, further features are expected. *Inductive data types* and *recursion* require (standard) syntactic extensions but otherwise seem unproblematic. As the encoding of non-inductive data types above suggests, to allow recursive type aliasing is sufficient to make these inductive. An open question here is whether such a treatment may solve the problem of *extensible data types* [61]. *Local store* would require a construct for introducing new locations, which appears similarly free of complications. *First-class* locations, storing and passing on locations as arguments, would cover things like *pointers* and *file handles*, and appears more challenging to administer in the type system. Even more so *arrays* and *pointer arithmetic*, which call for dependent types (note that adding these untyped is trivial; the challenge again is preserving types and confluence).

5 The Functional Machine Calculus

This section will present the new Functional Machine Calculus and its operational aspects: the abstract machine, a big-step evaluation relation, and the reduction relation. It will demonstrate that these agree: big-step evaluation defines complete runs on the machine (Proposition 5.4), weak head reduction simulates the machine (Proposition 5.6), and evaluation commutes with reduction (Proposition 5.8). Confluence is proved in the next section.

Let x, y, z range over *variables*; a, b, c over a global set of *locations* A with distinguished element λ; and i, j, k over *choices* with distinguished choice $\star$, with I, J denoting finite sets of choices.

Definition 5.1 *Terms* are given by the following grammar.

$$M, N \ ::= \ x \ \mid \ [N]a.\,M \ \mid \ a\langle x\rangle.\,M \ \mid \ i \ \mid \ N\,;i\!\rightarrow\! M \ \mid \ M^i$$

The constructs are: a *variable* x, an *application* or *push* on location a, an *abstraction* or *pop* on a that binds x in M, a *choice* i, a *case* $N\,;i\!\rightarrow\! M$, and a *loop* M^i. The notions of *variable binding*, *free variables* $\mathsf{fv}(-)$, and *capture-avoiding substitution* $\{N/x\}M$ of N for x in M are standard.

Define *argument stacks* S, T as stacks of terms, *memories* S_A as families of stacks in the set of locations A, and *continuation stacks* K, L as stacks of *conditional continuations* $i\!\rightarrow\! M$, as follows.

$$S, T \ ::= \ \varepsilon \ \mid \ S\,M \qquad\qquad S_A \ ::= \ \{S_a \mid a \in A\} \qquad\qquad K, L \ ::= \ (i\!\rightarrow\! M)\,K \ \mid \ \varepsilon$$

Stacks are composed by juxtaposition, ST, lifted to memories pointwise: $S_A T_A = \{S_a T_a \mid a \in A\}$. Write $a(M)$ for the singleton memory with M on the stack a and the empty stack on other locations. Memories may be assumed to have *finite support* (only finitely many stacks are non-empty), since a term uses only a fixed, finite set of locations. *Streams* may be used informally instead of stacks to model certain effects.

Definition 5.2 The *abstract machine* is as follows. *States* are triples (S_A, M, K) of a memory S_A, term M, and continuation stack K. *Transitions* are given by the top-to-bottom rules below, where $i \neq j$.

$$\frac{(\,S_A\ ,\ [N]a.\,M\ ,K\,)}{(\,S_A\,a(N)\ ,\ M\ ,K\,)} \qquad \frac{(\,S_A\ ,\ i\ ,\ (i\!\rightarrow\! M)\,K\,)}{(\,S_A\ ,M\ ,\ K\,)} \qquad \frac{(\,S_A\ ,N\,;i\!\rightarrow\! M\ ,\ K\,)}{(\,S_A\ ,\ N\ ,\ (i\!\rightarrow\! M)\,K\,)}$$

$$\frac{(\,S_A\,a(N)\ ,\ a\langle x\rangle.\,M\ ,K\,)}{(\,S_A\ ,\{N/x\}M\ ,K\,)} \qquad \frac{(\,S_A\ ,\ i\ ,(j\!\rightarrow\! M)\,K\,)}{(\,S_A\ ,\ i\ ,\ K\,)} \qquad \frac{(\,S_A\ ,\ M^i\ ,\ K\,)}{(\,S_A\ ,\ M\ ,(i\!\rightarrow\! M^i)\,K\,)}$$

A *run* of the machine is a sequence of steps written with a double line as below. A *final* state is of the form (S_A, i, ε) and a *failure* state of the form (S_A, x, K), or $(S_A, a\langle x\rangle.\,M, K)$ where $S_a = \varepsilon$. A run is *successful* if it terminates in a final state.

$$\begin{array}{c}(\,S_A\ ,M\ ,K\,)\\ \hline\hline (\,T_A\ ,\ N\ ,\ L\,)\end{array}$$

Observe that every state is either final, a failure state, or has a transition. The machine gives the small-step operational semantics of the FMC. The following big-step evaluation relation ($\Downarrow$) describes the overall behaviour of successful runs of the machine, and formalizes the intuitive version of the introduction:

$$S_A \overset{M}{\longmapsto\!\!\!\!\twoheadrightarrow} T_A, i \qquad \iff \qquad S_A, M \Downarrow T_A, i$$

Like the machine, it is deterministic, i.e. it is a partial function.

Definition 5.3 The *evaluation* relation $S_A, M \Downarrow T_A, i$ is defined inductively as follows, where $i \neq j$.

$$\frac{}{S_A,\ i \Downarrow S_A,\ i} \qquad \frac{S_A\, a(N),\ M \Downarrow T_A,\ i}{S_A,\ [N]a.\, M \Downarrow T_A,\ i} \qquad \frac{R_A,\ M \Downarrow S_A,\ i \quad S_A,\ N \Downarrow T_A,\ j}{R_A,\ M\,;i{\to}N \Downarrow T_A,\ j} \qquad \frac{R_A,\ M \Downarrow T_A,\ i}{R_A,\ M\,;j{\to}N \Downarrow T_A,\ i}$$

$$\frac{S_A,\ \{N/x\}M \Downarrow T_A,\ i}{S_A\, a(N),\ a\langle x\rangle.\, M \Downarrow T_A,\ i} \qquad \frac{R_A,\ M \Downarrow S_A,\ i \quad S_A,\ M^i \Downarrow T_A,\ j}{R_A,\ M^i \Downarrow T_A,\ j} \qquad \frac{R_A,\ M \Downarrow T_A,\ i}{R_A,\ M^j \Downarrow T_A,\ i}$$

Proposition 5.4 *Small-step and big-step semantics agree:*

$$\begin{array}{c} (\,S_A\,,M\,,\varepsilon\,) \\ \hline\hline (\,T_A\,,\quad i\,,\varepsilon\,) \end{array} \quad\iff\quad S_A, M \Downarrow T_A, i$$

Proof. $\Longrightarrow$ By induction on the run of the machine. $\Longleftarrow$ By induction on $\Downarrow$. □

Definition 5.5 The *reduction* relation $\rightarrowtriangle$ is given by closing the following rules under any context, where $a \neq b$, $i \neq j$, in the *passage* rule $x \notin \mathsf{fv}(N)$, and in the *prefix (pop)* rule $x \notin \mathsf{fv}(M)$.

$$\begin{array}{lrcl@{\qquad}rcll}
\text{beta} & [N]a.\,a\langle x\rangle.\,M & \rightarrowtriangle & \{N/x\}M & (a\langle x\rangle.\,N)\,;i{\to}M & \rightarrowtriangle & a\langle x\rangle.\,(N\,;i{\to}M) & \text{prefix (pop)} \\
\text{passage} & [N]b.\,a\langle x\rangle.\,M & \rightarrowtriangle & a\langle x\rangle.\,[N]b.\,M & ([P]a.\,N)\,;i{\to}M & \rightarrowtriangle & [P]a.\,(N\,;i{\to}M) & \text{prefix (push)} \\
\text{select} & i\,;i{\to}M & \rightarrowtriangle & M & P\,;i{\to}N\,;i{\to}M & \rightarrowtriangle & P\,;i{\to}(N\,;i{\to}M) & \text{associate} \\
\text{reject} & i\,;j{\to}M & \rightarrowtriangle & i & M^i & \rightarrowtriangle & M\,;i{\to}M^i & \text{unroll}
\end{array}$$

Weak head reduction $\rightarrowtriangle_{\mathsf{wh}}$ is given by closing the reduction rules under application contexts only: if $M \rightarrowtriangle_{\mathsf{wh}} N$ then $[P]a.\,M \rightarrowtriangle_{\mathsf{wh}} [P]a.\,N$. The reflexive–transitive closure of a reduction relation $\rightarrowtriangle$ is written $\twoheadrightarrow$, and reduction to normal form $\twoheadrightarrow\!\!\!\mid$.

Weak head reduction operates under a sequence of applications corresponding to a memory on the machine. To relate the machine and reduction, define the *readback* relation $(\mapsto)$ from states to terms by the exhaustive application of the following steps, reversing the *push* and *sequence* rules of the machine:

$$(\varepsilon, M, \varepsilon) \mapsto M \qquad (S_A\, a(N), M, K) \mapsto (S_A, [N]a.\,M, K) \qquad (S_A, M, (i{\to}N)\,K) \mapsto (S_A, M\,;i{\to}N, K)$$

The following then shows that weak head reduction simulates the machine.

Proposition 5.6 *If a state* (S_A, M, K) *reads back to* M' *and evaluates to a state* (T_A, N, L), *then the latter state reads back to a term* N' *such that* $M' \twoheadrightarrow_{\mathsf{wh}} N'$.

$$\begin{tikzcd}
(S_A, M, K) \arrow[r, mapsto] \arrow[d, phantom, "="] & M' \arrow[d, dashed, two heads, "\mathsf{wh}"] \\
(T_A, N, L) \arrow[r, dashed, mapsto] & N'
\end{tikzcd}$$

Proof. By induction on the machine run. □

In the other direction, weak head reduction preserves and reflects the evaluation behaviour of terms.

Proposition 5.7 *If* $M \to_{\mathsf{wh}} N$ *then* $S_A, M \Downarrow T_A, i$ *if and only if* $S_A, M \Downarrow T_A, i$.

$$\begin{tikzcd} S_A, M \arrow[rr, "\mathsf{wh}"] \arrow[dr, Rightarrow] & & S_A, N \arrow[dl, Rightarrow] \\ & T_A, i & \end{tikzcd}$$

Proof. By induction on $\to_{\mathsf{wh}}$. □

Combining both directions, evaluation with an empty memory $\varepsilon, M \Downarrow S_A, i$ coincides with weak head reduction to a term of the form $[N_1]a_1 \dots [N_n]a_n.\, i$ that represents the memory S_A with the choice i.

The following establishes that reduction in any context commutes with evaluation, demonstrating, in essence, that the evaluation semantics and reduction semantics of the FMC are compatible. Reduction may then be viewed as compile-time optimization, preserving the behaviour of evaluation. To state this formally reduction $\to$ is extended to memories: if $M \to N$ then $S_A\, a(M) \to S_A\, a(N)$, and if $S_A \to T_A$ then $S_A\, a(M) \to T_A\, a(M)$. Note that this does not extend the machine itself, but enables to compare evaluation of terms before and after reduction, modelling the optimization of stored functions.

Proposition 5.8 *If* $R_A \twoheadrightarrow S_A$, $M \twoheadrightarrow N$, *and* $R_A, M \Downarrow T_A, i$ *then there is a memory* U_A *such that* $T_A \twoheadrightarrow U_A$ *and* $S_A, N \Downarrow U_A, i$.

$$\begin{tikzcd} R_A, M \arrow[r, two heads] \arrow[d, Rightarrow] & S_A, N \arrow[d, Rightarrow] \\ T_A, i \arrow[r, dashed, two heads] & U_A, i \end{tikzcd}$$

Proof. By induction on the measure (m, n) where m is the size of the derivation for $R_A, M \Downarrow T_A, i$ and n is the number of reduction steps in $M \twoheadrightarrow N$, strengthening the statement with the assertion that the size of the derivation for $S_A, N \Downarrow U_A, i$ is at most m. The proof is similar to that of Lemma 9.5. □

The above propositions serve to demonstrate that the reduction relation is the correct one for the calculus, given its operational semantics. The reduction rules are *sound* for evaluation, in the sense of Proposition 5.8, and *complete* in the sense that weak head reduction implements the machine, Proposition 5.6. The next section will show that the reduction semantics is *consistent* by demonstrating confluence.

6 Confluence

The confluence proof follows the standard *parallel reduction* technique [63]. Reduction is split into *duplicating* reduction $\to_{\mathsf{d}}$, comprising *beta* and *unroll*, and *affine* reduction $\to_{\mathsf{a}}$, consisting of the remaining rules. The former is shown to be confluent by parallel reduction, while the latter is shown confluent and terminating by Newman's Lemma. The two relations are then shown to commute.

Lemma 6.1 *Affine reduction* $\to_{\mathsf{a}}$ *is terminating and confluent.*

Proof. For termination, define a measure on terms as the pair (n, m) where:

- n is the sum over the size of M for every subterm $M\,;i \to N$, and
- m is the sum over the size of M for every subterm $[N]a.\, M$.

The first component is invariant under the *passage* rewrite rule, and strictly reduces for the other rules (*select*, *reject*, both *prefix* rules, and *associate*), while the second component strictly reduces for *passage*, proving termination.

Confluence follows by Newman's Lemma from local confluence. There are the following critical pairs.

- *Passage–prefix*:

$$[N]b.\, a\langle x\rangle.\, M\,;i \to P \to_{\mathsf{a}} \begin{cases} Q = a\langle x\rangle.\, [N]b.\, M\,;i \to P \\ Q' = [N]b.\, (a\langle x\rangle.\, M\,;i \to P) \end{cases}$$

This is closed as follows.

$$\left.\begin{array}{r} Q \rightarrow_{\mathsf{a}} a\langle x\rangle.\,([N]b.\,M\,;i{\to}P) \\ Q' \rightarrow_{\mathsf{a}} [N]b.\,a\langle x\rangle.\,(M\,;i{\to}P) \end{array}\right\} \rightarrow_{\mathsf{a}} a\langle x\rangle.\,[N]b.\,(M\,;i{\to}P)$$

- *Select–associate*:

$$i\,;i{\to}M\,;i{\to}N \rightarrow_{\mathsf{a}} \begin{cases} P = M\,;i{\to}N \\ Q = i\,;i{\to}(M\,;i{\to}N) \end{cases}$$

  This is closed by $Q \rightarrow_{\mathsf{a}} P$.

- *Reject–associate*:

$$i\,;j{\to}M\,;j{\to}N \rightarrow_{\mathsf{a}} \begin{cases} P = i\,;j{\to}N \\ Q = i\,;j{\to}(M\,;j{\to}N) \end{cases}$$

  This is closed by $P \rightarrow_{\mathsf{a}} i$ and $Q \rightarrow_{\mathsf{a}} i$.

- *Prefix–associate*: The case for *prefix (pop)* is as follows.

$$a\langle x\rangle.\,M\,;i{\to}N\,;i{\to}M \begin{cases} Q = a\langle x\rangle.\,(M\,;i{\to}N)\,;i{\to}P \\ Q' = a\langle x\rangle.\,M\,;i{\to}(N\,;i{\to}P) \end{cases}$$

  This is closed as follows.

$$\left.\begin{array}{r} Q \rightarrow_{\mathsf{a}} a\langle x\rangle.\,(M\,;i{\to}N\,;i{\to}P) \\ Q' \end{array}\right\} \rightarrow_{\mathsf{a}} a\langle x\rangle.\,(M\,;i{\to}(N\,;i{\to}P))$$

  The case for *prefix (push)* is similar.

□

For duplicating reduction $\rightarrow_{\mathsf{d}}$, parallel reduction is defined by marking selected redexes and reducing these simultaneously by induction on the term.

Definition 6.2 A *marked* term is one equipped with a marking on a selection of *beta*- and *unroll*-redexes, indicated by underlining. The *marked reduct* $\lfloor M\rfloor$ of a marked term M is defined inductively as follows.

$$\begin{array}{cccc} \lfloor \underline{[N]a.\,a\langle x\rangle}.\,M\rfloor = \{\lfloor N\rfloor/x\}\lfloor M\rfloor & \lfloor x\rfloor = x & \lfloor [N]a.\,M\rfloor = [\lfloor N\rfloor]a.\,\lfloor M\rfloor & \lfloor M\,;i{\to}N\rfloor = \lfloor M\rfloor\,;i{\to}\lfloor N\rfloor \\ \lfloor \underline{M^i}\rfloor = \lfloor M\rfloor\,;i{\to}\lfloor M\rfloor^i & \lfloor i\rfloor = i & \lfloor a\langle x\rangle.\,M\rfloor = a\langle x\rangle.\,\lfloor M\rfloor & \lfloor M^i\rfloor = \lfloor M\rfloor^i \end{array}$$

A *parallel reduction step* $M \Rightarrow_{\mathsf{d}} N$ takes M to the marked reduct $N = \lfloor M\rfloor$ for some marking of M. The *complete development* $\lfloor\!\lfloor M\rfloor\!\rfloor$ of M is the marked reduct of marking every duplicating redex in M.

Parallel reduction is in-between single-step and multi-step reduction:

Lemma 6.3 $(\rightarrow_{\mathsf{d}}) \subset (\Rightarrow_{\mathsf{d}}) \subset (\twoheadrightarrow_{\mathsf{d}})$.

Proof. By marking the redex reduced in $M \rightarrow_{\mathsf{d}} N$, respectively by induction on $\lfloor - \rfloor$. □

Then duplicating reduction and parallel reduction are equivalent, $(\twoheadrightarrow_{\mathsf{d}}) = (\Rrightarrow_{\mathsf{d}})$. Next, a parallel step may be completed to a complete development by another parallel step, reducing the remaining redexes.

Lemma 6.4 *If* $M \Rightarrow_{\mathsf{d}} N$ *then* $N \Rightarrow_{\mathsf{d}} \lfloor\!\lfloor M\rfloor\!\rfloor$.

Proof. A marking on N such that $\lfloor N\rfloor = \lfloor\!\lfloor M\rfloor\!\rfloor$ will be given by induction on the marked term M. The two non-trivial cases are when M is an unmarked redex:

$$\begin{array}{c} \lfloor [M_1]a.\,a\langle x\rangle.\,M_2\rfloor = [\lfloor M_1\rfloor]a.\,a\langle x\rangle.\,\lfloor M_2\rfloor \\ \lfloor {M_1}^i\rfloor = \lfloor M_1\rfloor^i \end{array}$$

Let $N_k = \lfloor M_k \rfloor$ for $k = 1, 2$ and mark the above redexes in N. By induction, $\lfloor N_k \rfloor = \lfloor\!\lfloor M_k \rfloor\!\rfloor$. Then for $\lfloor N \rfloor$ and $\lfloor\!\lfloor M \rfloor\!\rfloor$ the cases are completed by:

$$\lfloor \underline{[N_1]a.\,a\langle x\rangle}.\,N_2 \rfloor = \{\lfloor N_1 \rfloor / x\} \lfloor N_2 \rfloor = \lfloor\!\lfloor [M_1]a.\,a\langle x\rangle.\,M_2 \rfloor\!\rfloor$$
$$\lfloor \underline{N_1{}^i} \rfloor = \lfloor N_1 \rfloor \,;\, i \to \lfloor N_1 \rfloor^i = \lfloor\!\lfloor M_1{}^i \rfloor\!\rfloor$$

□

It follows that parallel reduction is diamond, and hence duplicating reduction is confluent. To prove confluence for the full reduction relation $\rightarrow$, terms will be reduced to their affine normal form. First, it is shown how affine reduction commutes with parallel duplicating reduction.

Lemma 6.5 *If $P \;{}_{\mathsf{a}}{\twoheadleftarrow\!|}\; M \Rightarrow_{\mathsf{d}} N$ then $P \Rightarrow_{\mathsf{d}} Q \;{}_{\mathsf{a}}{\twoheadleftarrow}\; N$ for some Q.*

$$\begin{tikzcd}
M \arrow[r, Rightarrow, "\mathsf{d}"] \arrow[d, two heads, "\mathsf{a}"'] & N \arrow[d, dashed, two heads, "\mathsf{a}"] \\
P \arrow[r, dashed, Rightarrow, "\mathsf{d}"'] & Q
\end{tikzcd}$$

Proof. Let M be marked such that $\lfloor M \rfloor = N$. For every affine reduction step $M \rightarrow_{\mathsf{a}} P$ except *prefix (push)* on a marked redex, carrying over the marking from M to P gives $\lfloor M \rfloor \twoheadrightarrow_{\mathsf{a}} \lfloor P \rfloor$. The remaining case is as follows, where the redex becomes separated so that it cannot be marked.

$$\underline{[N]a.\,a\langle x\rangle}.\,M \,;\, i \to P \rightarrow_{\mathsf{a}} [N]a.\,(a\langle x\rangle.\,M \,;\, i \to P)$$

To address this, by confluence and termination the reduction $M \twoheadrightarrow\!|_{\mathsf{a}} P$ may be arranged so that a *prefix (push)*-step on a marked redex is immediately followed by the *prefix (pop)*-step that restores the redex:

$$[N]a.\,(a\langle x\rangle.\,M \,;\, i \to P) \rightarrow_{\mathsf{a}} \underline{[N]a.\,a\langle x\rangle}.\,(M \,;\, i \to P)$$

This give the required commutation (since x is not free in P):

$$\left.\begin{array}{r} \lfloor \underline{[N]a.\,a\langle x\rangle}.\,M \,;\, i \to P \rfloor \\ \lfloor \underline{[N]a.\,a\langle x\rangle}.\,(M \,;\, i \to P) \rfloor \end{array}\right\} = \{\lfloor N \rfloor / x\} \lfloor M \rfloor \,;\, i \to \lfloor P \rfloor$$

By induction on $M \twoheadrightarrow\!|_{\mathsf{a}} P$ it follows that $\lfloor M \rfloor \twoheadrightarrow_{\mathsf{a}} \lfloor P \rfloor$. □

Next, define *complete reduction* $(\Rrightarrow) = (\Rightarrow_{\mathsf{d}}) \cdot (\twoheadrightarrow\!|_{\mathsf{a}})$ as a parallel duplicating step followed by affine normalization.

Lemma 6.6 *Complete reduction is diamond.*

Proof. By the following diagram, where the top left triangles are by Lemma 6.3, the top right and bottom left squares are by Lemma 6.5, and the bottom right square is by confluence and termination of affine reduction (Lemma 6.1).

$$\begin{tikzcd}
M \arrow[r, Rightarrow, "\mathsf{d}"] \arrow[d, Rightarrow, "\mathsf{d}"'] \arrow[dr, Rightarrow, "\mathsf{d}"] & \cdot \arrow[r, two heads, "\mathsf{a}"] \arrow[d, Rightarrow, "\mathsf{d}"] & N \arrow[d, Rightarrow, "\mathsf{d}"] \\
\cdot \arrow[r, Rightarrow, "\mathsf{d}"'] \arrow[d, two heads, "\mathsf{a}"'] & \cdot \arrow[r, two heads, "\mathsf{a}"'] \arrow[d, two heads, "\mathsf{a}"] & \cdot \arrow[d, two heads, "\mathsf{a}"] \\
P \arrow[r, Rightarrow, "\mathsf{d}"'] & \cdot \arrow[r, two heads, "\mathsf{a}"'] & Q
\end{tikzcd}$$

□

To connect reduction to complete reduction, the following lemma will show that the image of a reduction step under affine normalization is a complete step.

Lemma 6.7 *If $M \rightarrow N$ then $M_{\mathsf a} \Rightarrow N_{\mathsf a}$ where $M_{\mathsf a}$ and $N_{\mathsf a}$ are the affine normal forms of M and N.*

Proof. The case of an affine step $M \rightarrow_{\mathsf a} N$ is immediate since $M_{\mathsf a} = N_{\mathsf a}$ by confluence and termination of affine reduction (Lemma 6.1). In the case of a duplicating step $M \rightarrow_{\mathsf d} N$, Lemma 6.4 gives a parallel step $M \Rightarrow_{\mathsf d} N$, for which Lemma 6.5 gives reductions $M_{\mathsf a} \Rightarrow_{\mathsf d} P \;{}_{\mathsf a}{\twoheadleftarrow}\; N$ for some term P. Then $P \twoheadrightarrow\!|_{\mathsf a} N_{\mathsf a}$ again by Lemma 6.1, giving the required reduction $M_{\mathsf a} \Rightarrow_{\mathsf d} P \twoheadrightarrow\!|_{\mathsf a} N_{\mathsf a}$.

$$\begin{tikzcd}
M \arrow[r, "\mathsf a"] \arrow[d, "\mathsf a"', ->>|] & N \arrow[d, "\mathsf a", ->>|] \\
M_{\mathsf a} \arrow[r, phantom, "="] & N_{\mathsf a}
\end{tikzcd}
\qquad
\begin{tikzcd}
M \arrow[r, "\mathsf d"] \arrow[d, "\mathsf a"', ->>|] & N \arrow[d, "\mathsf a", two heads] \arrow[dr, "\mathsf a", ->>|] & \\
M_{\mathsf a} \arrow[r, Rightarrow, "\mathsf d"'] & P \arrow[r, "\mathsf a"', ->>|] & N_{\mathsf a}
\end{tikzcd}$$

□

The confluence proof puts everything together: affine normalization maps reduction to complete reduction, which is confluent, and which is included in reduction.

Theorem 6.8 *Reduction $\rightarrow$ is confluent.*

Proof. Let $P \twoheadleftarrow M \twoheadrightarrow N$. By Lemma 6.7 there are complete reductions $P_{\mathsf a} \Leftarrow\!\!\Leftarrow M_{\mathsf a} \Rightarrow\!\!\Rightarrow N_{\mathsf a}$ where $M_{\mathsf a}$, $N_{\mathsf a}$, and $P_{\mathsf a}$ are the affine normal forms of respectively M, N, and P. The diamond property for complete reduction (Lemma 6.6) gives reductions $P_{\mathsf a} \Rightarrow\!\!\Rightarrow Q \Leftarrow\!\!\Leftarrow N_{\mathsf a}$ for some Q. Since a parallel step $\Rightarrow_{\mathsf d}$ corresponds to a duplicating reduction $\twoheadrightarrow_{\mathsf d}$ (Lemma 6.3), a complete step $\Rightarrow$ corresponds to a reduction $\twoheadrightarrow$, which gives the desired converging reductions $P \twoheadrightarrow\!|_{\mathsf a} P_{\mathsf a} \twoheadrightarrow Q \twoheadleftarrow N_{\mathsf a} \;{}_{\mathsf a}|\!{\twoheadleftarrow}\; N$.

$$\begin{tikzcd}
M \arrow[rr, two heads] \arrow[dr, "\mathsf a", ->>|] \arrow[dd, two heads] & & N \arrow[d, "\mathsf a", ->>|] \\
& M_{\mathsf a} \arrow[r, Rightarrow, two heads] \arrow[d, Rightarrow, two heads] & N_{\mathsf a} \arrow[d, Rightarrow, two heads] \\
P \arrow[r, "\mathsf a", ->>|] & P_{\mathsf a} \arrow[r, Rightarrow, two heads] & Q
\end{tikzcd}$$

□

7 Types

This section formally introduces the simply typed FMC with control. Types are stratified into four layers: types for terms, vectors of types for stacks, location-indexed families of vectors for memories, and choice-indexed families of memory types as return types. The formal definitions use indexed families directly, while the notation of the informal introduction is given as operations on types or as syntactic sugar.

$$\begin{array}{llll}
\text{Types:} & \rho,\,\sigma,\,\tau \;::=\; \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\overline{\tau}}}_I & \qquad \text{Memory types:} & \overline{\overline{\tau}} \;::=\; \{\overline{\tau}_a \mid a \in A\} \\
\text{Stack types:} & \overline{\tau} \;::=\; \tau_1 \ldots \tau_n & \qquad \text{Sum types:} & \overline{\overline{\overline{\tau}}}_I \;::=\; \{\overline{\overline{\overline{\tau}}}_i \mid i \in I\}
\end{array}$$

The base cases are given by empty stack types and memory types, both written ε, and the empty sum type, written 0. Vectors are composed by juxtaposition $\overline{\sigma}\,\overline{\tau}$, lifted to families point-wise: $\overline{\overline{\sigma}}\,\overline{\overline{\tau}} = \{\overline{\sigma}_a\,\overline{\tau}_a \mid a \in A\}$. The singleton family holding $\overline{\tau}$ at location a and empty elsewhere is written $a(\overline{\tau})$. This retrieves the

$$\frac{}{\Gamma,\, x : \tau \vdash x : \tau}\,\textsf{var} \qquad \frac{}{\Gamma \vdash i : \varepsilon \Rightarrow \varepsilon.i}\,\textsf{chc}$$

$$\frac{\Gamma \vdash N : \rho \quad \Gamma \vdash M : a(\rho)\,\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}{\Gamma \vdash [N]a.\,M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}\,\textsf{push} \qquad \frac{\Gamma,\, x : \rho \vdash M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}{\Gamma \vdash a\langle x\rangle.\,M : a(\rho)\,\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}\,\textsf{pop} \qquad \frac{\Gamma \vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_I}{\Gamma \vdash M : \overline{\overline{\rho}}\,\overline{\overline{\sigma}} \Rightarrow (\overline{\overline{\sigma}}\,\overline{\overline{\tau}})_I}\,\textsf{exp}$$

$$\frac{\Gamma \vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_{I\setminus i} + \overline{\overline{\sigma}}.i \quad \Gamma \vdash N : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}{\Gamma \vdash M\,;i\!\to\! N : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_I}\,\textsf{case} \qquad \frac{\Gamma \vdash M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I + \overline{\overline{\sigma}}.i}{\Gamma \vdash M^i : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}\,\textsf{loop} \qquad \frac{\Gamma \vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\sigma}}_I}{\Gamma \vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\sigma}}_I + \overline{\overline{\tau}}_J}\,\textsf{incl}$$

Fig. 1. The simply-typed FMC with control

notation $a_1(\tau_1)\dots a_n(\tau_n)$ for memory types in the introductory sections, and generalizes it to allow type vectors $a_1(\overline{\tau}_1)\dots a_n(\overline{\tau}_n)$.

Sum types are combined by $\overline{\overline{\sigma}}_I + \overline{\overline{\tau}}_J$ where $I \cap J = \varnothing$. To retrieve the notation from the informal development, the notation $\overline{\overline{\tau}}.i$ indicates the singleton family over $\{i\}$ containing only the memory type $\overline{\overline{\tau}}$ for the choice i; formally, $\overline{\overline{\tau}}.i$ is $\overline{\overline{\tau}}_{\{i\}}$ where $\overline{\overline{\tau}}_i = \overline{\overline{\tau}}$. That is, $\overline{\overline{\tau}}_i$ is a *memory type* as a member of a family, and $\overline{\overline{\tau}}.i$ is a *sum type* that is a singleton family. Sum types may then be written $\overline{\overline{\tau}}_1.i_1 + \ldots + \overline{\overline{\tau}}_n.i_n$ as before. Finally, $\overline{\overline{\tau}}_{I\setminus i}$ (respectively $\overline{\overline{\tau}}_{I\setminus J}$) denotes the family $\overline{\overline{\tau}}_I$ minus the element $\overline{\overline{\tau}}_i$ (respectively the elements $\overline{\overline{\tau}}_j$ for $j \in J$), if present.

Stack types follow the order of terms on the stack, $MNP : \rho\sigma\tau$ for $M : \rho$, $N : \sigma$, and $P : \tau$. Since stacks are last-in-first-out, identity terms are of the form $\langle x\rangle.\,\langle y\rangle.\,\langle z\rangle.\,[z].\,[y].\,[x].\,\star$, with the order of pops reversed relative to a given input stack. The convention is then that stack types on the left of an implication are presented in reverse order, i.e. the type for this term would be $\lambda(\tau\sigma\rho) \Rightarrow \lambda(\rho\sigma\tau).\star$, written $\lambda(\overline{\tau}) \Rightarrow \lambda(\overline{\tau}).\star$ for the stack type $\overline{\tau} = \rho\sigma\tau$.

A *context* Γ is a finite function from variables to types, written as a sequence $x_1 : \tau_1, \dots, x_n : \tau_n$. A *typing judgement* $\Gamma \vdash M : \tau$ assigns the type τ to the term M in the context Γ.

Definition 7.1 The *simply-typed FMC with control* is given by the typing rules in Figure 1.

A few notes on the typing rules. The rules push and pop are the equivalent of the rules for application and abstraction for the simply-typed λ-calculus, since the arrow type $\sigma\!\to\!\tau$ is interpreted as introducing an additional input type σ to the input type vector of τ. The rule exp for *(stack) expansion* extends the input and output memory types of a term by $\overline{\overline{\sigma}}$, on every output branch, reflecting the principle of stack calculi that terms may operate on arbitrarily large stacks, returning any additional part untouched. The rule case for $M\,;i\!\to\!N$ requires the output of M on choice i to match the *input* of N, and on any other choice $I \setminus i$ to match the *output* of N; the output type $\overline{\overline{\tau}}_I$ of N may or may not have a component $\overline{\overline{\tau}}_i$.

Observe that the typing rules as given are not inductive on terms, due to the rules exp (*expansion*) and incl (*inclusion*). This gives a simpler presentation and reduces repetition in proofs. Both rules can however be permuted up past the other rules, which means they may instead be integrated into the *variable* and *choice* rules; or they may permuted down to be integrated into *push* and *case* rules.

Next, the basic properties of preservation of types under substitution and reduction are given.

Lemma 7.2 (Subject substitution) *If* $\Gamma \vdash N : \sigma$ *and* $\Gamma,\, x : \sigma \vdash M : \tau$ *then* $\Gamma \vdash \{N/x\}M : \tau$.

Proof. By induction on the typing derivation for M. □

Proposition 7.3 (Subject reduction) *If* $\Gamma \vdash M : \tau$ *and* $M \twoheadrightarrow N$ *then* $\Gamma \vdash N : \tau$.

Proof. By induction on the typing derivation for M, with top-level reduction steps as base cases, and using the subject substitution lemma (Lemma 7.2) in the case of a *beta* step. □

To demonstrate that machine evaluation preserves types, the type system is extended to stacks, mem-

$$\dfrac{}{\vdash \varepsilon : \varepsilon}\,\textsf{stk0} \qquad\qquad \dfrac{}{\vdash \varepsilon : \overline{\overline{\tau}}_I \Rightarrow \overline{\overline{\tau}}_I}\,\textsf{cnt0}$$

$$\dfrac{\vdash S : \overline{\sigma} \qquad \vdash M : \tau}{\vdash S\,M : \overline{\sigma}\,\tau}\,\textsf{stk1} \qquad\qquad \dfrac{\vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\sigma}}_I + \overline{\overline{\tau}}_{J\setminus I} \qquad \vdash K : \overline{\overline{\sigma}}_I \Rightarrow \overline{\overline{\tau}}_J}{\vdash (i \to M)\,K : \overline{\overline{\rho}}.i + \overline{\overline{\sigma}}_{I\setminus i} \Rightarrow \overline{\overline{\tau}}_J}\,\textsf{cnt1}$$

$$\dfrac{\{\vdash S_a : \overline{\sigma}_a\}_{a\in A}}{\vdash S_A : \{\overline{\sigma}_a \mid a \in A\}}\,\textsf{mem} \qquad\qquad \dfrac{\vdash S_A : \overline{\overline{\rho}} \qquad \vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\sigma}}_I + \overline{\overline{\tau}}_{J\setminus I} \qquad \vdash K : \overline{\overline{\sigma}}_I \Rightarrow \overline{\overline{\tau}}_J}{\vdash (S_A, M, K) : \varepsilon \Rightarrow \overline{\overline{\tau}}_J}\,\textsf{state}$$

Fig. 2. Extended types for stacks, memories, and states

ories, and machine states by the rules in Figure 2. Continuation stacks will have types $\overline{\overline{\sigma}}_I \Rightarrow \overline{\overline{\tau}}_J$, extending the grammar of types.

Proposition 7.4 (Machine evaluation preserves types) *For a typed state $\vdash (S_A, M, K) : \varepsilon \Rightarrow \overline{\overline{\tau}}_I$, if*

$$\dfrac{(\,S_A\,,\,M\,,K\,)}{(\,T_A\,,\,N\,,\;L\,)}$$

then $\vdash (T_A, N, L) : \varepsilon \Rightarrow \overline{\overline{\tau}}_I$.

Proof. By inspection of the machine transitions. □

Proposition 7.5 (Machine progress) *A typed state is either final or has a machine step.*

Proof. Recall that states are either final, have a machine step, or are failure states, which are those with a free variable x as term or with an abstraction $a\langle x\rangle.\,M$ and an empty stack ε_a in the memory. Both cases are ruled out by the type system. □

In the FMC, unlike the λ-calculus, all types are inhabited. A *zero* term for a type $\tau = \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I$ will evaluate on the machine by discarding a memory of $\overline{\overline{\sigma}}$, and for some $i \in I$ returning a memory of zero-terms of type $\overline{\overline{\tau}}_i$. Zero terms are formally defined as follows: if I is non-empty, select $i \in I$ and define:

$$0_\tau = \langle \_{}_{\overline{\overline{\sigma}}}\rangle.\,[0_{\overline{\overline{\tau}}_i}].\,i$$

where $\langle \_{}_{\overline{\overline{\sigma}}}\rangle$ is a sequence of non-binding abstractions $a_1\langle\_\rangle \dots a_n\langle\_\rangle$ matching $\overline{\overline{\sigma}} = a_1(\sigma_1)\dots a_n(\sigma_n)$, and $[0_{\overline{\overline{\tau}}}]$ is a sequence of applications $[0_{\tau_1}]a_1 \dots [0_{\tau_n}]a_n$ matching $\overline{\overline{\tau}}_i = a_1(\tau_1)\dots a_n(\tau_n)$. For zero terms where I is empty, where $\tau = \overline{\overline{\sigma}} \Rightarrow 0$, let $\tau' = \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\sigma}}.\star$ and define the zero term as the loop $0_\tau = (0_{\tau'})^\star$. Note that for a type $\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I$ and a given $i \in I$, zero-terms are equivalent by the permutations $a\langle x\rangle.\,b\langle y\rangle.\,M \sim b\langle y\rangle.\,a\langle x\rangle.\,M$ and $[P]a.\,[N]b.\,M \sim [N]b.\,[P]a.\,M$ where $a \neq b$.

Proposition 7.6 (Type inhabitation) *Every type τ is inhabited by a zero term $\vdash 0_\tau : \tau$.*

Proof. By induction on the type τ. □

8 Machine termination

For terms without loops, types guarantee termination of the machine. By this, the following is meant.

Definition 8.1 A term M is *terminating* if $S_A, M \Downarrow T_A, i$ for some memories S_A, T_A and choice i.

The typed notion, proved below, is stronger. For a term $M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I$, for *any* memory $S_A : \overline{\overline{\sigma}}$ of the correct type there are $i \in I$ and $T_A : \overline{\overline{\tau}}_i$ such that $S_A, M \Downarrow T_A, i$. Type inhabitation guarantees that a suitable input memory $S_A : \overline{\overline{\sigma}}$ exists, so that the typed notion implies the untyped definition above.

This will be proved using the standard Tait reducibility technique [62]. Each type τ is associated with a set $\textsc{run}(\tau)$ of terminating terms, here called the *runnable* terms by analogy to the usual *reducible* terms of strong normalization arguments. It is then shown by induction on typing derivations that every typed term is runnable, and hence terminating.

Definition 8.2 The set $\textsc{run}(\tau)$ of *runnable terms* for a type τ is defined as the set of closed terms

$$\textsc{run}(\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I) = \{M \;\mid\; \forall S_A \in \textsc{run}(\overline{\overline{\sigma}}).\ \exists i \in I.\ \exists T_A \in \textsc{run}(\overline{\overline{\tau}}_i).\ S_A, M \Downarrow T_A, i\ \}$$

where the runnable sets for memory types $\overline{\overline{\tau}}$ and stack types $\overline{\tau}$ are as follows.

$$\textsc{run}(\{\overline{\tau}_a \mid a \in A\}) = \{S_A \mid \forall a \in A.\ S_a \in \textsc{run}(\overline{\tau}_a)\} \qquad \textsc{run}(\tau_1 \dots \tau_n) = \{\varepsilon\, M_1 \dots M_n \mid M_i \in \textsc{run}(\tau_i)\}$$

To work with open terms, a *substitution map* s is a finite function from variables to terms, applied to terms as sM as a simultaneous substitution for the variables in its domain. Denote by $s\{M/x\}$ the map that assigns M to x and is as s for other variables. Then $\textsc{run}(\Gamma)$ associates a context Γ with the substitution maps over runnable terms of the types in Γ:

$$\textsc{run}(x_1 : \tau_1, \dots, x_n : \tau_n) = \{s \mid sx_i \in \textsc{run}(\tau_i)\}$$

The next lemma will show that any loop-free, typed term is runnable. It assumes return types to be non-empty, i.e. types are not of the form $\overline{\overline{\sigma}} \Rightarrow 0$. The proof is then a direct induction on typing derivations.

Lemma 8.3 (Typed terms are runnable) *If* $\Gamma \vdash M : \tau$ *then* $sM \in \textsc{run}(\tau)$ *for any* $s \in \textsc{run}(\Gamma)$.

Proof. By induction on the typing derivation for $\Gamma \vdash M : \tau$.

- *Variable:*

$$\frac{}{\Gamma,\, x : \tau \vdash x : \tau}\,\textsf{var}$$

For any $s \in \textsc{run}(\Gamma,\, x : \tau)$ by definition $sx \in \textsc{run}(\tau)$.

- *Push:*

$$\frac{\Gamma \vdash N : \rho \qquad \Gamma \vdash M : a(\rho)\,\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}{\Gamma \vdash [N]a.\, M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}\,\textsf{push}$$

Let $s \in \textsc{run}(\Gamma)$ and $S_A \in \textsc{run}(\overline{\overline{\sigma}})$. By the inductive hypothesis, $sN \in \textsc{run}(\rho)$, so that $S_A\, a(sN) \in \textsc{run}(\overline{\overline{\sigma}}\, a(\rho))$. Again by the inductive hypothesis, $sM \in \textsc{run}(a(\rho)\,\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I)$, which means it evaluates as $S_A\, a(sN)$, $sM \Downarrow T_A$, i for some $i \in I$ and $T_A \in \textsc{run}(\overline{\overline{\tau}}_i)$. Then S_A, $[sN]a.\, sM \Downarrow T_A$, i by the definition of ($\Downarrow$), which gives $s([N]a.\, M) = [sN]a.\, sM \in \textsc{run}(\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I)$.

- *Pop:*

$$\frac{\Gamma,\, x : \rho \vdash M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}{\Gamma \vdash a\langle x\rangle.\, M : a(\rho)\,\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_J}\,\textsf{pop}$$

Let $s \in \textsc{run}(\Gamma)$ and $S_A\, a(N) \in \textsc{run}(\overline{\overline{\sigma}}\, a(\rho))$, and assume by α-equivalence that x is not in the domain of Γ. Then $S_A \in \textsc{run}(\overline{\overline{\sigma}})$ and $s\{N/x\} \in \textsc{run}(\Gamma,\, x : \rho)$. Since N is closed, $(s\{N/x\})M = s(\{N/x\}M)$. By the inductive hypothesis, $s(\{N/x\}M) \in \textsc{run}(\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I)$ so that S_A, $s(\{N/x\}M) \Downarrow T_A$, i for some $i \in I$ and $T_A \in \textsc{run}(\overline{\overline{\tau}}_i)$. By the definition of ($\Downarrow$) it follows that $S_A\, a(N)$, $s(a\langle x\rangle.\, M) \Downarrow T_A$, i and hence $s(a\langle x\rangle.\, M) \in \textsc{run}(a(\rho)\,\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I)$.

- *Choice:*

$$\frac{}{\Gamma \vdash i : \varepsilon \Rightarrow \varepsilon.i}\,\textsf{chc}$$

Note that the empty memory $\varepsilon \in \textsc{run}(\varepsilon)$ is the only inhabitant of $\textsc{run}(\varepsilon)$, and that $si = i$ for any substitution map s. Since ε, $i \Downarrow \varepsilon$, i it follows that $i \in \textsc{run}(\varepsilon \Rightarrow \varepsilon.i)$.

- *Case:*

$$\frac{\Gamma \vdash N : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_{I\setminus i} + \overline{\overline{\sigma}}.i \qquad \Gamma \vdash M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I}{\Gamma \vdash N\,;i \!\rightarrow\! M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_I}\,\textsf{case}$$

Let $s \in \textsc{run}(\Gamma)$ and $R_A \in \textsc{run}(\overline{\overline{\rho}})$. The inductive hypothesis gives:

$$sN \in \textsc{run}(\overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_{I\setminus i} + \overline{\overline{\sigma}}.i) \qquad \text{and} \qquad sM \in \textsc{run}(\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I)$$

For the former, there are two cases:

$$R_A,\ sN \Downarrow S_A,\ i \qquad \text{or} \qquad R_A,\ sN \Downarrow T_A,\ j \quad \text{where} \quad j \in I,\ i \neq j$$

In the first case, $S_A \in \textsc{run}(\overline{\overline{\sigma}})$ and hence $S_A,\ sM \Downarrow T_A,\ k$ for some $k \in I$ and $T_A \in \textsc{run}(\overline{\overline{\tau}}_k)$. The definition of $(\Downarrow)$ gives the evaluation below left. For the second case there is the evaluation below right. It follows that $s(M\,;i{\to}N) \in \textsc{run}(\overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_I)$.

$$\frac{R_A,\ M \Downarrow S_A,\ i \quad S_A,\ N \Downarrow T_A,\ k}{R_A,\ M\,;i{\to}N \Downarrow T_A,\ k} \qquad\qquad \frac{R_A,\ M \Downarrow T_A,\ j}{R_A,\ M\,;i{\to}N \Downarrow T_A,\ j}$$

- *Expansion*

$$\frac{\Gamma \vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\tau}}_I}{\Gamma \vdash M : \overline{\overline{\rho}}\,\overline{\overline{\sigma}} \Rightarrow (\overline{\overline{\sigma}}\,\overline{\overline{\tau}})_I}\,\mathsf{exp}$$

By induction on $(\Downarrow)$, if $R_A, M \Downarrow T_A, j$ then $S_A R_A, M \Downarrow S_A T_A, i$. The case is then immediate.

- *Inclusion*

$$\frac{\Gamma \vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\sigma}}_I}{\Gamma \vdash M : \overline{\overline{\rho}} \Rightarrow \overline{\overline{\sigma}}_I + \overline{\overline{\tau}}_J}\,\mathsf{incl}$$

Immediate since a return memory $S_A \in \textsc{run}(\overline{\overline{\sigma}}_i)$ for a choice $i \in I$ is also one for $i \in I \cup J$.

□

The theorem is then as follows.

Theorem 8.4 (Machine termination) *Any loop-free, typed term* $\vdash M : \overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I$ *with a zero-free type is terminating.*

Proof. By type inhabitation there is a loop-free memory $S_A : \overline{\overline{\sigma}}$. By Lemma 8.3, $M \in \textsc{run}(\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I)$ and $S_A \in \textsc{run}(\overline{\overline{\sigma}})$. By the definition of runnable terms, $S_A, M \Downarrow T_A, i$ for some i and T_A. □

9 Strong normalization

This section will prove that typed, loop-free terms are strongly normalizing. The proof is centered around the idea that beta-reduction shortens a run of the machine by eliminating consecutive push- and pop-transitions. This gives a proof in two stages. First, a reducibility argument along the lines of that for machine termination establishes that typed terms have a finite evaluation on the machine, for which the *measured evaluation* relation defined below counts the number of pop-transitions. Second, it is shown that *beta*-reduction strictly reduces this measure, while affine reduction does not increase it. Since affine reduction is inherently normalizing, this establishes typed strong normalization.

The first stage needs to overcome the obstacle that terms that are discarded without evaluation must somehow be measured as well. This is familiar from many strong normalization proofs, such as those deriving strong normalization from weak normalization [45,31] and those using perpetual reduction strategies [5,65], and can be distinguished in the combinatoric arguments of many others. Here, it is addressed by requiring machine evaluation of all subterms: for an application $[N]a.\,M$ the argument N is evaluated as well as pushed to the stack, and for a case $M\,;i{\to}N$ the continuation N is evaluated even when M terminates with $j \neq i$. Evaluating such terms requires an input stack, which for typed terms is guaranteed by type inhabitation (Proposition 7.6).

The proof thus comprises a *logical* part, using abstract reducibility, and a *combinatorial* part, measuring abstraction steps in machine evaluation and demonstrating that the measure reduces under *beta*-reduction.

This may be viewed as a decomposition of the proof for the previous iteration of the FMC [3], which followed the style of Gandy's proof [16,15], computing the same measure directly from typing derivations. The key to this decomposition is type inhabitation: Gandy's proof, which maps terms onto domains of monotone functionals over integers, relies on the existence of minimal elements, essentially higher-order versions of the constant zero function, to provide input to any function. Here, because all types are inhabited, zero-terms 0_τ may fulfil that rôle.

This section will again assume loop-free terms (i.e. not containing the loop construct M^i) and non-empty sum types (i.e. $I \neq \varnothing$ for any $\overline{\overline{\tau}}_I$).

Definition 9.1 *Measured evaluation* $S_A, M \Downarrow_n T_A, i$ is defined inductively as follows, where $i \neq j$.

$$\frac{}{S_A,\, i \Downarrow_0 S_A,\, i} \qquad \frac{R_A,\, N \Downarrow_n U_A,\, k \quad S_A\, a(N),\, M \Downarrow_m T_A,\, i}{S_A,\, [N]a.\, M \Downarrow_{n+m} T_A,\, i} \qquad \frac{R_A,\, M \Downarrow_m S_A,\, i \quad S_A,\, N \Downarrow_n T_A,\, k}{R_A,\, M\,;i{\to}N \Downarrow_{m+n} T_A,\, k}$$

$$\frac{S_A,\, \{N/x\}M \Downarrow_n T_A,\, i}{S_A\, a(N),\, a\langle x\rangle.\, M \Downarrow_{n+1} T_A,\, i} \qquad \frac{R_A,\, M \Downarrow_m S_A,\, j \quad T_A,\, N \Downarrow_n U_A,\, k}{R_A,\, M\,;i{\to}N \Downarrow_{m+n} S_A,\, j}$$

The reducibility sets $\textsc{eval}(-)$ interpret types as a guarantee that a measured evaluation exists.

Definition 9.2 The set $\textsc{eval}(\tau)$ for a type τ is defined as the set of closed terms

$$\textsc{eval}(\overline{\overline{\sigma}} \Rightarrow \overline{\overline{\tau}}_I) = \{M \mid \forall S_A \in \textsc{eval}(\overline{\overline{\sigma}}).\ \exists m \in \mathbb{N}.\ \exists i \in I.\ \exists T_A \in \textsc{eval}(\overline{\overline{\tau}}_i).\ S_A, M \Downarrow_m T_A, i\ \}$$

where for memory types $\overline{\overline{\tau}}$ and stack types $\overline{\tau}$:

$$\textsc{eval}(\{\overline{\tau}_a \mid a \in A\}) = \{S_A \mid \forall a \in A.\ S_a \in \textsc{eval}(\overline{\tau}_a)\} \qquad \textsc{eval}(\tau_1 \dots \tau_n) = \{\varepsilon\, M_1 \dots M_n \mid M_i \in \textsc{eval}(\tau_i)\}$$

For contexts, $\textsc{run}(\Gamma)$ is a set of substitution maps s:

$$\textsc{eval}(x_1 : \tau_1, \dots, x_n : \tau_n) = \{s \mid s(x_i) \in \textsc{eval}(\tau_i)\}$$

Evaluation sets are inhabited at least by zero-terms.

Lemma 9.3 *A zero-term* 0_τ *is in* $\textsc{eval}(\tau)$.

Proof. For a type σ, let $\mathsf{src}(\sigma)$ and $\mathsf{tgt}(\sigma)$ be its source (memory) type and target (sum) type, so that $\sigma = \mathsf{src}(\sigma) \Rightarrow \mathsf{tgt}(\sigma)$. By induction on τ it will be shown for all S_A of the dimensions of $\mathsf{src}(\tau)$ that $S_A, 0_\tau \Downarrow_n 0_{\mathsf{tgt}(\tau)_i}, i$ for some n and i. This immediately implies $0_\tau \in \textsc{eval}(\tau)$, since any $S_A \in \textsc{eval}(\mathsf{src}(\tau))$ will be of the right dimensions, and inductively $0_{\mathsf{tgt}(\tau)_i} \in \textsc{eval}(\mathsf{tgt}(\tau)_i)$ for the return memory. Let 0_τ be as follows.

$$0_\tau = \langle_{-\mathsf{src}(\tau)}\rangle \cdot [0_{\mathsf{tgt}(\tau)_i}] \cdot i$$

By induction, for every σ in $\mathsf{tgt}(\tau)_i$ there is the following evaluation for some j_σ and m_σ, since $0_{\mathsf{src}(\sigma)}$ is of the required dimensions.

$$0_{\mathsf{src}(\sigma)},\, 0_\sigma \Downarrow_{m_\sigma} 0_{\mathsf{tgt}(\sigma)_j},\, j_\sigma$$

Then for any memory S_A of the dimensions of $\mathsf{src}(\tau)$ there is the following evaluation, where the double lines indicate multiple evaluation rules, m is the sum over the measures m_σ for each σ in $\mathsf{tgt}(\tau)_i$, and n is the total size of $\mathsf{src}(\tau)$.

$$\begin{array}{r} \{0_{\mathsf{src}(\sigma)},\, 0_\sigma \Downarrow_{m_\sigma} 0_{\mathsf{tgt}(\sigma)_j},\, j_\sigma\}_{\sigma \in \mathsf{tgt}(\tau)_i} \quad \dfrac{}{0_{\mathsf{tgt}(\tau)_i},\, i \Downarrow_0 0_{\mathsf{tgt}(\tau)_i},\, i} \\ \hline\hline \varepsilon,\, [0_{\mathsf{tgt}(\tau)_i}] \cdot i \Downarrow_m 0_{\mathsf{tgt}(\tau)_i},\, i \\ \hline\hline S_A,\, \langle_{-\mathsf{src}(\tau)}\rangle \cdot [0_{\mathsf{tgt}(\tau)_i}] \cdot i \Downarrow_{n+m} 0_{\mathsf{tgt}(\tau)_i},\, i \end{array}$$

□

The main reducibility lemma then shows that types guarantee measured evaluation.

Lemma 9.4 *If* $\Gamma \vdash M : \tau$ *and* $s \in \textsc{eval}(\Gamma)$ *then* $sM \in \textsc{eval}(\tau)$.

Proof. By induction on the typing derivation for M. The proof is similar to that of Lemma 8.3 that typed terms are runnable. □

Reduction ($\rightarrow$) reduces the measure given by measured evaluation, by the following lemma.

Lemma 9.5 *If* $S_A, M \Downarrow_n T_A, i$ *and* $S_A \twoheadrightarrow S'_A$ *and* $M \twoheadrightarrow M'$, *then* $S'_A, M' \Downarrow_{n'} T'_A, i$ *where* $T_A \twoheadrightarrow T'_A$ *and* $n \geq n'$. *If moreover* $M \twoheadrightarrow M'$ *contains beta-steps, then* $n > n'$.

Proof. By an outer induction on the size of the derivation for $\Downarrow_n$, and an inner induction on the length of the reduction $M \twoheadrightarrow M'$.

- *Choice:*

$$\frac{}{S_A,\ i \Downarrow_0 S_A,\ i}$$

For $S_A \twoheadrightarrow S'_A$ it is immediate that $S'_A,\ i \Downarrow_0 S'_A,\ i$.

- *Pop:*

$$\frac{S_A,\ \{N/x\}M \Downarrow_n T_A,\ i}{S_A\, a(N),\ a\langle x\rangle.\, M \Downarrow_{n+1} T_A,\ i}$$

Let $S_A \twoheadrightarrow S'A$ and $N \twoheadrightarrow N'$, and note that the reduction on $a\langle x\rangle.\, M$ must be of the form $a\langle x\rangle.\, M \twoheadrightarrow a\langle x\rangle.\, M'$. Since $\{N/x\}M \twoheadrightarrow \{N'/x\}M'$ the outer inductive hypothesis gives the required T'_A and n'.

- *Push:*

$$\frac{R_A,\ N \Downarrow_n U_A,\ k \quad S_A\, a(N),\ M \Downarrow_m T_A,\ i}{S_A,\ [N]a.\, M \Downarrow_{n+m} T_A,\ i}$$

There are three cases, depending on the first reduction step on $[N]a.\, M$: 1) reduction in either subterm M or N, 2) a top level beta-step, or 3) a top level passage step. For case 1), given $S_A \twoheadrightarrow S'_A$ and $[N]a.\, M \rightarrow [N']a.\, M'$ (where $N = N'$ or $M = M'$), the outer inductive hypothesis for $M \rightarrow M'$ gives the required T'_A and m', or that for $N \rightarrow N'$ gives n'. For the remaining reduction from $[N']a.\, M'$ the statement follows by the inner inductive hypothesis.

In case 2) $M = a\langle x\rangle.\, P$ with a beta-step $[N]a.\, a\langle x\rangle.\, P \rightarrow \{N/x\}P$. Evaluation is as follows.

$$\frac{R_A,\ N \Downarrow_n U_A,\ k \quad \dfrac{S_A,\ \{N/x\}P \Downarrow_m T_A,\ i}{S_A\, a(N),\ a\langle x\rangle.\, P \Downarrow_{m+1} T_A,\ i}}{S_A,\ [N]a.\, a\langle x\rangle.\, P \Downarrow_{n+m+1} T_A,\ i}$$

Given $S_A \twoheadrightarrow S'_A$ and the remaining reduction $\{N/x\}P \twoheadrightarrow M'$, the (outer) inductive hypothesis gives the the evaluation $S'_A, M' \Downarrow_{m'} T'_A, i$ where $T_A \twoheadrightarrow T'_A$ and $m \geq m'$, as required.

In case 3) $M = b\langle x\rangle.\, P$ with a passage step $[N]a.\, b\langle x\rangle.\, P \rightarrow b\langle x\rangle.\, [N]a.\, P$ where $a \neq b$ and $x \notin \mathsf{fv}(N)$. Evaluation for the redex is below left, and for the reduct below right, with a derivation of the same size. The inner inductive hypothesis then gives the required T'_A, m', and n'.

$$\frac{R_A,\ N \Downarrow_n U_A,\ k \quad \dfrac{S_A\, a(N),\ \{Q/x\}P \Downarrow_m T_A,\ i}{S_A\, b(Q)\, a(N),\ b\langle x\rangle.\, P \Downarrow_{m+1} T_A,\ i}}{S_A\, b(Q),\ [N]a.\, b\langle x\rangle.\, P \Downarrow_{n+m+1} T_A,\ i} \qquad \frac{\dfrac{R_A,\ N \Downarrow_n U_A,\ k \quad S_A\, a(N),\ \{Q/x\}P \Downarrow_m T_A,\ i}{S_A,\ [N]a.\, \{Q/x\}P \Downarrow_{n+m} T_A,\ i}}{S_A\, b(Q),\ b\langle x\rangle.\, [N]a.\, P \Downarrow_{n+m+1} T_A,\ i}$$

- *Case:*

$$a)\quad \frac{R_A,\, M \Downarrow_m S_A,\, i \qquad S_A,\, N \Downarrow_n T_A,\, k}{R_A,\, M\,;i{\to}N \Downarrow_{m+n} T_A,\, k} \qquad\qquad b)\quad \frac{R_A,\, M \Downarrow_m S_A,\, j \qquad T_A,\, N \Downarrow_n U_A,\, k}{R_A,\, M\,;i{\to}N \Downarrow_{m+n} S_A,\, j}$$

For each of the derivations a) and b) above there are five sub-cases, depending on the first reduction step: one for reduction inside M or N, and four for a top-level reduction step, as follows.

$$\begin{array}{lrcll}
1) & M\,;i{\to}N & \rightarrowtriangle & M'\,;i{\to}N' & \text{where } M' = M \text{ or } N' = N \\
2) & i\,;i{\to}N & \rightarrowtriangle & N & \text{i.e. } M = i, \text{ or} \\
 & j\,;i{\to}N & \rightarrowtriangle & j & \text{i.e. } M = j \text{ where } i \neq j \\
3) & (Q\,;i{\to}P)\,;i{\to}N & \rightarrowtriangle & Q\,;i{\to}(P\,;i{\to}N) & \text{i.e. } M = Q\,;i{\to}P \\
4) & (a\langle x\rangle.\,P)\,;i{\to}N & \rightarrowtriangle & a\langle x\rangle.\,(P\,;i{\to}N) & \text{i.e. } M = a\langle x\rangle.\,P \text{ where } x \notin \mathsf{fv}(N) \\
5) & ([Q]a.\,P)\,;i{\to}N & \rightarrowtriangle & [Q]a.\,(P\,;i{\to}N) & \text{i.e. } M = [Q]a.\,P
\end{array}$$

Case 1) follows by the outer inductive hypothesis on M' or N' and inner inductive hypothesis on the remaining reduction, similar to the first case for *Push*. For case 2) evaluation is of the following forms respectively; 2a) follows by the outer inductive hypothesis on N, and 2b) is immediate from the evaluation $R_A, j \Downarrow_0 R_A, j$.

$$2a)\quad \frac{S_A,\, i \Downarrow_0 S_A,\, i \qquad S_A,\, N \Downarrow_n T_A,\, k}{R_A,\, i\,;i{\to}N \Downarrow_n T_A,\, k} \qquad\qquad 2b)\quad \frac{R_A,\, j \Downarrow_0 R_A,\, j \qquad T_A,\, N \Downarrow_n U_A,\, k}{R_A,\, j\,;i{\to}N \Downarrow_n R_A,\, j}$$

Cases 3) through 5) follow similarly to the third *Push* case above, for passage reduction, by reconfiguring the evaluation derivation and observing that its size is preserved.

□

The previous two lemmata then give typed strong normalization: the first shows that for a typed term a measured evaluation exists; the second that this measure bounds the number of beta-steps in the reduction of the term.

Theorem 9.6 (Typed strong normalization) *If $\Gamma \vdash M : \tau$ then M is strongly normalizing.*

Proof. Let s be a substitution map taking every $x : \sigma$ in Γ to a zero-term 0_σ. Then $s \in \textsc{eval}(\Gamma)$ by Lemma 9.3 and $sM \in \textsc{eval}(\tau)$ by Lemma 9.4. Let $\tau = \overline{\overline{\rho}} \Rightarrow \overline{\overline{\sigma}}_I$. Lemma 9.3 gives $0_{\overline{\overline{\rho}}} \in \textsc{eval}(\overline{\overline{\rho}})$, and by the definition of $\textsc{eval}(-)$ there is an evaluation $0_{\overline{\overline{\rho}}}, sM \Downarrow_m S_A, i$. By Lemma 9.5 reduction on M preserves the measure m and reduces it in case of beta-steps. Since by Lemma 6.1 affine reduction is strongly normalizing, an infinite reduction from M would have infinitely many beta-steps, a contradiction. □

References

[1] Appel, A., D. MacQueen, R. Milner and M. Tofte, *Unifying exceptions with constructors in standard ML*, Technical Report ECS-LFCS-88-55, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University (1988).
Available online

[2] Barrett, C., D. Castle and W. Heijltjes, *The Relational Machine Calculus*, in: P. Sobocinski, U. D. Lago and J. Esparza, editors, *Proc. 39th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 9:1–9:15, ACM (2024).
https://doi.org/10.1145/3661814.3662091

[3] Barrett, C., W. Heijltjes and G. McCusker, *The Functional Machine Calculus II: Semantics*, in: B. Klin and E. Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023).
https://doi.org/10.4230/LIPIcs.CSL.2023.10

[4] Benton, N. and A. Kennedy, *Exceptional syntax*, Journal of Functional Programming **11**, pages 395–410 (2001).
https://doi.org/10.1017/S0956796801004099

[5] Bergstra, J. A. and J. W. Klop, *Strong normalization and perpetual reductions in the lambda calculus*, J. Inf. Process. Cybern. **18**, pages 403–417 (1982).
Available online

[6] Bierman, G. M., *A computational interpretation of the lambda-mu-calculus*, in: L. Brim, J. Gruska and J. Zlatuska, editors, *Proc. 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 336–345, Springer (1998).
https://doi.org/10.1007/BFB0055783

[7] Bloom, S. L. and Z. Ésik, *Iteration Theories - The Equational Logic of Iterative Processes*, EATCS Monographs on Theoretical Computer Science, Springer (1993), ISBN 978-3-642-78036-3.
https://doi.org/10.1007/978-3-642-78034-9

[8] Carraro, A., T. Ehrhard and A. Salibra, *The stack calculus*, in: D. Kesner and P. Viana, editors, *Proc. Seventh Workshop on Logical and Semantic Frameworks, with Applications (LSFA)*, volume 113 of *EPTCS*, pages 93–108 (2012).
https://doi.org/10.4204/EPTCS.113.10

[9] Crolard, T., *A confluent lambda-calculus with a catch/throw mechanism*, Journal of Functional Programming **9**, pages 625–647 (1999).
https://doi.org/10.1017/S0956796899003512

[10] Davies, R. and F. Pfenning, *Intersection types and computational effects*, in: M. Odersky and P. Wadler, editors, *Proc. Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 198–208, ACM (2000).
https://doi.org/10.1145/351240.351259

[11] de Groote, P., *A simple calculus of exception handling*, in: M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Proc. Second International Conference on Typed Lambda Calculi and Applications (TLCA '95)*, volume 902 of *Lecture Notes in Computer Science*, pages 201–215, Springer (1995).
https://doi.org/10.1007/BFB0014054

[12] de'Liguoro, U. and R. Treglia, *Intersection types for a λ-calculus with global store*, in: N. Veltri, N. Benton and S. Ghilezan, editors, *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 5:1–5:11, ACM (2021).
https://doi.org/10.1145/3479394.3479400

[13] Douence, R. and P. Fradet, *A systematic study of functional language implementations*, ACM Transactions on Programming Languages and Systems **20**, pages 344–387 (1998).
https://doi.org/10.1145/276393.276397

[14] Fiore, M. and S. Staton, *Substitution, jumps, and algebraic effects*, in: *Proc. Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS)*, pages 41:1–41:10, ACM (2014).
https://doi.org/10.1145/2603088.2603163

[15] Fuhs, C. and C. Kop, *Polynomial interpretations for higher-order rewriting*, in: A. Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 176–192, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012).
https://doi.org/10.4230/LIPICS.RTA.2012.176

[16] Gandy, R., *Proofs of strong normalization*, in: J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477, Academic Press (1980).

[17] Gentzen, G., *Untersuchungen über das logische Schließen I, II*, Mathematische Zeitschrift **39**, pages 176–210, 405–431 (1934–1935). English translation in: The Collected Papers of Gerhard Gentzen, M.E. Szabo (ed.), North-Holland 1969.

[18] Griffin, T., *A formulae-as-types notion of control*, in: F. E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco (POPL)*, pages 47–58, ACM Press (1990).
https://doi.org/10.1145/96709.96714

[19] Hasegawa, M., *Decomposing typed lambda-calculus into a couple of categorical programming languages*, in: *International Conference on Category Theory and Computer Science* (1995).
https://doi.org/10.1007/3-540-60164-3_28

[20] Heijltjes, W., *The Functional Machine Calculus*, in: *Proceedings of the 38th Conference on the Mathematical Foundations of Programming Semantics, MFPS XXXVIII*, volume 1 of *ENTICS* (2022).
https://doi.org/10.46298/ENTICS.10513

[21] Heijltjes, W., *The Functional Machine Calculus III: Choice (early announcement)*, CoRR **abs/2411.04615** (2024), 2411.04615.
https://doi.org/10.48550/ARXIV.2411.04615

[22] Heijltjes, W., *Quantitative types for the Functional Machine Calculus*, in: M. Fernández, editor, *10th International Conference on Formal Structures for Computation and Deduction (FSCD)*, volume 337 of *LIPIcs*, pages 24:1–24:20, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025).
https://doi.org/10.4230/LIPICS.FSCD.2025.24

[23] Heijltjes, W. and G. Majury, *Simple types for probabilistic termination*, in: J. Endrullis and S. Schmitz, editors, *33rd EACSL Annual Conference on Computer Science Logic (CSL)*, volume 326 of *LIPIcs*, pages 31:1–31:22, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2025).
https://doi.org/10.4230/LIPIcs.CSL.2025.31

[24] Herzberg, D. and T. Reichert, *Concatenative programming: An overlooked paradigm in functional programming*, in: *Proc. 4th International Conference on Software and Data Technologies (ICSOFT)*, pages 257–263 (2009).
Available online

[25] Hillerström, D. and S. Lindley, *Shallow effect handlers*, in: S. Ryu, editor, *Proc. Programming Languages and Systems - 16th Asian Symposium (APLAS)*, volume 11275 of *Lecture Notes in Computer Science*, pages 415–435, Springer (2018).
https://doi.org/10.1007/978-3-030-02768-1_22

[26] Hirschkoff, D., E. Prebet and D. Sangiorgi, *On the representation of references in the pi-calculus*, in: *31st International Conference on Concurrency Theory (CONCUR)*, LIPIcs, pages 34:1–34:20, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020).
https://doi.org/10.4230/LIPIcs.CONCUR.2020.34

[27] Jones, S. L. P., A. Reid, F. Henderson, C. A. R. Hoare and S. Marlow, *A semantics for imprecise exceptions*, in: B. G. Ryder and B. G. Zorn, editors, *Proc. 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 25–36, ACM (1999).
https://doi.org/10.1145/301618.301637

[28] Kameyama, Y. and M. Sato, *Strong normalizability of the non-deterministic catch/throw calculi*, Theoretical Computer Science **272**, pages 223–245 (2002).
https://doi.org/10.1016/S0304-3975(00)00352-2

[29] Kammar, O., S. Lindley and N. Oury, *Handlers in action*, in: G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 145–158, ACM (2013).
https://doi.org/10.1145/2500365.2500590

[30] Kiselyov, O., A. Sabry and C. Swords, *Extensible effects: an alternative to monad transformers*, in: *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 59–70 (2013).
https://doi.org/10.1145/2503778.2503791

[31] Klop, J. W., *Combinatory Reduction Systems*, Ph.D. thesis, Rijksuniversiteit Utrecht (1980).
Available online

[32] Krivine, J.-L., *A call-by-name lambda-calculus machine*, Higher-Order and Symbolic Computation **20**, pages 199–207 (2007).
https://doi.org/10.1007/s10990-007-9018-9

[33] Laird, J., *Exceptions, continuations and macro-expressiveness*, in: D. L. Métayer, editor, *Proc. 11th European Symposium on Programming ESOP*, volume 2305 of *Lecture Notes in Computer Science*, pages 133–146, Springer (2002).
https://doi.org/10.1007/3-540-45927-8_10

[34] Landin, P. J., *The mechanical evaluation of expressions*, The Computer Journal **6**, pages 308–320 (1964).
https://doi.org/10.1093/comjnl/6.4.308

[35] Landin, P. J., *The next 700 programming languages*, Communications of the ACM **9**, pages 157–166 (1966).
https://doi.org/10.1145/365230.365257

[36] Lebresne, S., *A type system for call-by-name exceptions*, Log. Methods Comput. Sci. **5** (2009).
https://doi.org/10.2168/LMCS-5(4:1)2009

[37] Levy, P. B., *Call-by-push-value: A subsuming paradigm*, in: *International Conference on Typed Lambda Calculi and Applications (TLCA)*, pages 228–243, Springer, Berlin, Heidelberg (1999).
https://doi.org/10.1007/3-540-48959-2_17

[38] Levy, P. B., *Call-by-push-value: A functional/imperative synthesis*, volume 2 of *Semantic Structures in Computation*, Springer Netherlands (2003).
https://doi.org/10.1007/978-94-007-0954-6

[39] Levy, P. B., *Monads and adjunctions for global exceptions*, in: S. D. Brookes and M. W. Mislove, editors, *Proc. 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS)*, volume 158 of *Electronic Notes in Theoretical Computer Science*, pages 261–287, Elsevier (2006).
https://doi.org/10.1016/J.ENTCS.2006.04.014

[40] Liang, S., P. Hudak and M. Jones, *Monad transformers and modular interpreters*, in: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 333–343 (1995).
https://doi.org/10.1145/199448.199528

[41] Maršik, J., M. Amblard and P. de Groote, *Introducing* $(\!|\lambda|\!)$*, a* λ*-calculus for effectful computation*, Theoretical Computer Science **869**, pages 108–155 (2021).
https://doi.org/10.1016/j.tcs.2021.02.038

[42] Maurer, L., P. Downen, Z. M. Ariola and S. L. P. Jones, *Compiling without continuations*, in: A. Cohen and M. T. Vechev, editors, *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 482–494, ACM (2017).
https://doi.org/10.1145/3062341.3062380

[43] Milner, R., J. Parrow and D. Walker, *A calculus of mobile processes, I*, Information and Computation **100**, pages 1–40 (1992).
https://doi.org/10.1016/0890-5401(92)90008-4

[44] Moggi, E., *Notions of computation and monads*, Information and Computation **93**, pages 55–92 (1991).
https://doi.org/10.1016/0890-5401(91)90052-4

[45] Nederpelt, R., *Strong normalization in a typed lambda calculus with lambda structured types*, Ph.D. thesis, Technische hogeschool Eindhoven (1973).
Available online

[46] Ong, C. L. and C. A. Stewart, *A Curry-Howard foundation for functional computation with control*, in: P. Lee, F. Henglein and N. D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 215–227, ACM Press (1997).
https://doi.org/10.1145/263699.263722

[47] Parigot, M., $\lambda\mu$*-Calculus: an algorithmic interpretation of classical natural deduction*, in: *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 624 of *Lecture Notes in Computer Science (LNCS)*, pages 190–201 (1992).
https://doi.org/10.1007/BFb0013061

[48] Pestov, S., D. Ehrenberg and J. Groff, *Factor: A dynamic stack-based programming language*, ACM SIGPLAN Notices **45**, pages 43–58 (2010).
https://doi.org/10.1145/1899661.1869637

[49] Peyton Jones, S. L., A. D. Gordon and S. Finne, *Concurrent Haskell*, in: H. Boehm and G. L. S. Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–308, ACM Press (1996).
https://doi.org/10.1145/237721.237794

[50] Plotkin, G. and J. 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).
https://doi.org/10.1007/3-540-45931-6_24

[51] Plotkin, G. and M. Pretnar, *Handling algebraic effects*, Logical Methods in Computer Science **9** (2013).
https://doi.org/10.2168/LMCS-9(4:23)2013

[52] Plotkin, G. D., *Call-by-name, call-by-value and the* λ*-calculus*, Theoretical Computer Science **1**, pages 125–159 (1975).
https://doi.org/10.1016/0304-3975(75)90017-1

[53] Power, A. and H. Thielecke, *Closed Freyd- and κ-categories*, in: *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1644 of *LNCS*, pages 625–634, Springer (1999).
https://doi.org/10.1007/3-540-48523-6_59

[54] Riecke, J. G. and H. Thielecke, *Typed exeptions and continuations cannot macro-express each other*, in: J. Wiedermann, P. van Emde Boas and M. Nielsen, editors, *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, pages 635–644, Springer (1999).
https://doi.org/10.1007/3-540-48523-6_60

[55] Rocha, P. and L. Caires, *Propositions-as-types and shared state*, Proc. ACM Program. Lang. **5**, pages 1–30 (2021).
https://doi.org/10.1145/3473584

[56] Rossberg, A., B. L. Titzer, A. Haas, D. L. Schuff, D. Gohman, L. Wagner, A. Zakai, J. F. Bastien and M. Holman, *Bringing the web up to speed with webassembly*, Communications of the ACM **61**, pages 107–115 (2018).
https://doi.org/10.1145/3282510

[57] Saabas, A. and T. Uustalu, *A compositional natural semantics and Hoare logic for low-level languages*, Theor. Comput. Sci. **373**, pages 273–302 (2007).
https://doi.org/10.1016/J.TCS.2006.12.020

[58] Simpson, A. K. and G. D. Plotkin, *Complete axioms for categorical fixed-point operators*, in: *Proc. 15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 30–41, IEEE Computer Society (2000).
https://doi.org/10.1109/LICS.2000.855753

[59] Spivey, J. M., *A functional theory of exceptions*, Science of Computer Programming **14**, pages 25–42 (1990).
https://doi.org/10.1016/0167-6423(90)90056-J

[60] Streicher, T. and B. Reus, *Classical logic, continuation semantics and abstract machines*, Journal of Functional Programming **8**, pages 543–572 (1998).
https://doi.org/10.1017/S0956796898003141

[61] Swierstra, W., *Data types à la carte*, Journal of Functional Programming **18**, pages 423–436 (2008).
https://doi.org/10.1017/S0956796808006758

[62] Tait, W. W., *Intensional interpretations of functionals of finite type I*, The Journal of Symbolic Logic **32**, pages 198–212 (1967).
https://doi.org/10.2307/2271658

[63] Takahashi, M., *Parallel reductions in lambda-calculus*, Information and Computation **118**, pages 120–127 (1995).
https://doi.org/10.1006/inco.1995.1057

[64] van Bakel, S., *Exception handling and classical logic*, in: E. Komendantskaya, editor, *Proc. 21st International Symposium on Principles and Practice of Programming Languages (PPDP 2019)*, pages 21:1–21:14, ACM (2019).
https://doi.org/10.1145/3354166.3354186

[65] van Raamsdonk, F., P. Severi, M. H. Sørensen and H. Xi, *Perpetual reductions in lambda-calculus*, Inf. Comput. **149**, pages 173–225 (1999).
https://doi.org/10.1006/inco.1998.2750

[66] Wadler, P., *How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages*, in: J. Jouannaud, editor, *Proc. Functional Programming Languages and Computer Architecture (FPCA 1985)*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128, Springer (1985).
https://doi.org/10.1007/3-540-15975-4_33

[67] Wu, N. and T. Schrijvers, *Fusion for free: Efficient algebraic effect handlers*, in: *International Conference on Mathematics of Program Construction* (2015).
https://doi.org/10.1007/978-3-319-19797-5_15