




Quantitative Types for the Functional Machine Calculus

Willem Heijltjes   

Department of Computer Science, University of Bath, UK

Abstract

The Functional Machine Calculus (FMC, Heijltjes 2022) extends the lambda-calculus with the computational effects of global mutable store, input/output, and probabilistic choice while maintaining confluent reduction and simply-typed strong normalization. Based in a simple call-by-name stack machine in the style of Krivine, the FMC models effects through additional argument stacks, and introduces sequential composition through a continuation stack to encode call-by-value behaviour, where simple types guarantee termination of the machine.

The present paper provides a discipline of quantitative types, also known as non-idempotent intersection types, for the FMC, in two variants. In the weak variant, typeability coincides with termination of the stack machine and with spine normalization, while exactly measuring the transitions in machine evaluation. The strong variant characterizes strong normalization through a notion of perpetual evaluation, while giving an upper bound to the length of reductions. Through the encoding of effects, quantitative typeability coincides with termination for higher-order mutable store, input/output, and probabilistic choice.

2012 ACM Subject Classification Theory of computation → Type structures; Theory of computation → Lambda calculus

Keywords and phrases lambda-calculus, computational effects, intersection types

Digital Object Identifier 10.4230/LIPIcs.FSCD.2025.24

Related Version *Extended version with all proofs:* <https://arxiv.org/abs/2505.09960>

Acknowledgements I am grateful to Vincent van Oostrom for many interesting discussions. Thanks also to the anonymous referees for suggestions and pointers to the literature.

1 Introduction

The Functional Machine Calculus (FMC) [5, 24] is a new approach to extending the λ -calculus with computational effects, preserving confluent reduction and simply-typed strong normalization for the effects of global mutable store, input/output, and probabilistic choice. This paper presents a generalization of the type system to *non-idempotent intersection types*, a notion of *quantitative* types that not only characterizes weak or strong normalisation, but in addition provides quantitative information about the length of reductions.

It is well known that introducing effects into the λ -calculus renders it non-confluent. The FMC gets around this by starting not from β -reduction, which indeed is incompatible with global effect operations, but from the (simplified) Krivine machine [30]. This gives a call-by-name operational semantics to the λ -calculus in the form of an abstract machine with a single stack, where application pushes its argument and abstraction pops. The FMC then introduces additional stacks to model effects, for example mutable variables as stacks of depth one and a random generator as a probabilistically generated stream (an infinite stack). It further adds sequential composition, naturally interpreted on the stack machine, which allows the encoding of call-by-value evaluation as well as Moggi’s computational metalanguage [34] and Levy’s call-by-push-value [32]. Confluent reduction and simple types are then derived from the behaviour of the machine, where types have a natural interpretation as a guarantee of successful termination of the machine [24].



© Willem Heijltjes;

licensed under Creative Commons License CC-BY 4.0

10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025).

Editor: Maribel Fernández; Article No. 24; pp. 24:1–24:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The FMC continues to be developed, for example with an extension to *exception handling* [25] and as a symmetric variant for relational programming [4], and there are many remaining questions regarding its type system and possible extensions to it. The simply-typed FMC is constrained in its expressiveness, for instance requiring the consumption of a fixed number of probabilistic values from the stream representing a random generator. Like monadic encodings of effects such as store, semantically it remains a Cartesian closed category [5]. Richer type systems are required (and expected) to more accurately capture the nature of effects, as demonstrated by a recent type system for probabilistic termination [26]. The aim of the present paper is to take a next step in this direction, moving from simple types to intersection types.

Intersection type systems [12, 41], which characterize exactly the normalizing terms of a calculus, have become an important semantic tool (see [8] for a recent overview). They are a form of finite polymorphism, where a term may be assigned multiple types. Conversely, a collection of multiple types is inhabited by the intersection of the inhabitants of its constituents – hence the name *intersection types*. The original *idempotent* intersection types, where type collections are sets, were later joined by *non-idempotent* variants [11, 27, 36], where collections are multisets. In addition to characterizing normalization, these resource-sensitive versions provide *quantitative* information about terms such as the maximal length of reductions. They are closely related to a range of other models and techniques, including resource calculi and differential λ -calculus [9, 21, 28], relational models [31, 38], and game semantics [37]. The relation between both variants manifests at the semantic level as a connection between Scott models and relational models [18, 19]. Intersection types have been studied from various other angles including category theory [33] and logic and proof theory [20, 23, 39].

Importantly for the present purpose, intersection types have been used to study computational effects [7, 13, 16]. In particular, intersection type systems exist for probabilistic choice [2, 10], mutable store [1, 14, 15], and monadic effects in general [22].

Where these efforts start from untyped effects and introduce tailored intersection types to capture their semantics, the development in this paper is in the opposite direction: it starts from a simply-typed calculus for effects, the FMC, and derives intersection types essentially by the standard generalization, which is to replace the input types of a function with a collection type. The litmus test is whether both approaches arrive at the same solution. Validating both views, this is confirmed in Section 7, where it is demonstrated that modulo the distinction between idempotent and non-idempotent types, the type system presented here agrees with that of De'Liguoro and Treglia for higher-order store [14]. Appendix A extends the comparison to the non-idempotent type system for store of Alves, Kesner, and Ramos [1].

The contributions of the paper are as follows. The overall result is to give a system of quantitative types for the FMC that is implicitly reflected on encoded effects and strategies: higher-order store, input/output, and probabilistic choice, the call-by-name and call-by-value λ -calculus [40], the computational metalanguage [34], and call-by-push-value [32] (see Section 7 for some of their encodings, and [24] for the remainder). The type system has two variants, *weak* and *strong*. The main theorem for the weak system, Theorem 20, is that weak quantitative types characterize both termination of the machine, where types give the exact number of steps in machine evaluation, and spine normalization. That machine termination and spine normalization coincide then holds untyped, which is a new result for the FMC relating the machine to reduction. The main result for strong quantitative types, Theorem 29, is that typing coincides with strong normalization and gives an upper bound

to reduction length. The development is via a new notion of *perpetual evaluation* for the FMC, an inductive definition of iterated weak head reduction to normal form analogous to *perpetual strategies* for the λ -calculus [3, 6, 43].

An interesting observation is that weak typing corresponds to machine termination, an important and natural notion for the FMC, but not to weak normalization. That is because, unlike for the call-by-name λ -calculus, the machine may terminate with a non-empty stack of return values, about which the type system gives no guarantees (i.e. they may be non-terminating).

The paper uses a variant definition of the FMC that differs superficially from the original [24]. The changes and their benefits are explained after the calculus is introduced in the next section. Confluence holds [25], but since it is unpublished at time of writing, the present exposition is formulated so as not to rely on it.

The development aims to put the emphasis on careful definitions, so that proofs become straightforward case analyses and structural inductions. This manifests particularly in the perpetual evaluation relation used to relate strong types and strong normalization. Proofs that are indeed of this kind are omitted (an extended version with all proofs in appendices is available from the author's website: <http://willem.heijltj.es/pdf/2025-QFMC.pdf>).

2 The Functional Machine Calculus

This section will define the FMC: its syntax, the abstract machine, a quantitative big-step semantics, its reduction relations, and the type system.

The primary intuition for the FMC is as an instruction language for a machine with multiple stacks, indexed by a set of *locations* $\{a, b, c, \dots\}$, and together called a *memory*. The calculus interacts with individual stacks through *push* and *pop* constructs: $[N]a.M$ pushes the term N to the stack a and continues as M , and $a\langle x \rangle.M$ pops from the stack a , using the popped term as x in M . This generalizes the evaluation of λ -terms on the (simplified) Krivine machine [30], where application pushes its argument, and abstraction pops. The standard, call-by-name λ -calculus [40] then embeds by choosing a *default* location λ , by letting $M N = [N]\lambda.M$ and $\lambda x.M = \lambda\langle x \rangle.M$. For brevity, the location λ may be omitted, writing instead $[N].M$ respectively $\langle x \rangle.M$. The locations of the FMC may encode *reader/writer* effects: higher order store as stacks of depth one, I/O as separate input and output streams, and non-determinism and probabilistic choice as non-deterministically and probabilistically generated input streams (see [24] for details).

The FMC further includes *sequencing* $M; N$ and its unit *skip* \star . These are composition and identity of FMC terms viewed as *memory transformers*: computations with an input memory and output memory. The big-step semantics, below, uses this interpretation directly, while it is implemented on the machine via a continuation stack, in standard fashion.

► **Definition 1** (Syntax). *The terms of the Functional Machine Calculus are given by the following grammar.*

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

From left to right these are: a variable x , an abstraction or pop $a\langle x \rangle.M$ on location a which binds x in M , an application or push $[N]a.M$ of term N on location a , the unit or skip \star , and sequencing $M; N$.

The operational meaning of the five constructs will be expressed through a small-step semantics, in the form of a simple stack machine, and a big-step semantics, in the form of an inductive evaluation relation. The abstract machine will use the following components.

24:4 Quantitative Types for the FMC

An *operand stack* or *argument stack* S is a stack of terms, represented with the head on the right and using ε for the empty stack. A *memory* is a family of operand stacks indexed in a set of locations A . A *continuation stack* is a stack of terms with the head to the left.

$$S, T ::= \varepsilon \mid S \mathbf{M} \quad S_A ::= \{S_a \mid a \in A\} \quad K, L ::= \mathbf{M} K \mid \varepsilon$$

For a memory S_A , the stack a being empty, ε_a , or outside the family, $a \notin A$, are equivalent. Thus we may fix a set of locations A going forward, noting that any given term uses only a fixed, finite set of locations, determined syntactically. Let $S_A \cdot \mathbf{M}_a$ denote the memory S_A with \mathbf{M} pushed to the stack indexed by a .

► **Definition 2** (Abstract machine). A machine state (S_A, \mathbf{M}, K) is a triple of a memory S_A , a term \mathbf{M} , and a continuation stack K . The transitions or steps of the machine given by the following rules.

$$\frac{(\ S_A \ , \ [N]a. \mathbf{M} \ , \ K \)}{(\ S_A \cdot N_a \ , \ \mathbf{M} \ , \ K \)} \quad \frac{(\ S_A \ , \ \mathbf{M} ; N \ , \ K \)}{(\ S_A \ , \ \mathbf{M} \ , \ N K \)}$$

$$\frac{(\ S_A \cdot N_a \ , \ a\langle x \rangle. \mathbf{M} \ , \ K \)}{(\ S_A \ , \ \{N/x\} \mathbf{M} \ , \ K \)} \quad \frac{(\ S_A \ , \ \star \ , \ \mathbf{M} K \)}{(\ S_A \ , \ \mathbf{M} \ , \ K \)}$$

A final state is of the form $(S_A, \star, \varepsilon)$ and a failure state of either of the forms (S_A, \mathbf{x}, K) or $(S_A, a\langle x \rangle. \mathbf{M}, K)$ where $S_a = \varepsilon$. A run is a sequence of steps, written as below, where the length of the run $n \geq 1$ is the number of states it traverses (which may be omitted from the notation). A run is successful if it terminates in a final state.

$$\frac{(\ S_A \ , \ \mathbf{M} \ , \ K \)}{(\ T_A \ , \ N \ , \ L \)}^n$$

The following big-step semantics describes the successful runs on the machine, for a given term, as an inductive *evaluation* relation \Downarrow_n between memories. The length of a run will correspond exactly to the number of rule instances in a derivation for \Downarrow_n , recorded as the number n . This correspondence, and a similar one with the type system defined later in this section, is the reason to measure the length of a run by its *states* and not its *steps*: a unit term \star is introduced by a rule, so correspondingly the unit run will have length one, not zero.

► **Definition 3** (Quantified big-step semantics). The quantified evaluation relation \Downarrow_n between an input memory, a term, and an output memory is defined inductively as follows.

$$\frac{}{S_A, \star \Downarrow_1 S_A} \quad \frac{R_A, \mathbf{M} \Downarrow_m S_A \quad S_A, N \Downarrow_n T_A}{R_A, \mathbf{M} ; N \Downarrow_{m+n+1} T_A}$$

$$\frac{S_A \cdot N_a, \mathbf{M} \Downarrow_n T_A}{S_A, [N]a. \mathbf{M} \Downarrow_{n+1} T_A} \quad \frac{S_A, \{N/x\} \mathbf{M} \Downarrow_n T_A}{S_A \cdot N_a, a\langle x \rangle. \mathbf{M} \Downarrow_{n+1} T_A}$$

Both the machine and the evaluation relation are *deterministic*: there is exactly one rule for each constructor (except none for the variable). Evaluation is thus a partial function, undefined where the machine is non-terminating or ends in a failure state. Note that it is not inductive on terms due to the substitution in the rule for abstraction, which handles the *variable* construct.

► **Proposition 4** (Small-step and big-step semantics agree). *Successful machine runs of length n correspond to quantified evaluation \Downarrow_n :*

$$\frac{(S_A, M, \varepsilon)}{(T_A, \star, \varepsilon)}^n \iff S_A, M \Downarrow_n T_A$$

We will see two reduction relations. Regular reduction \rightarrow is a rewrite relation, closed under all contexts, and spine reduction \rightarrow_w is restricted to spine contexts, which means everywhere except within the argument N of an application $[N]a.M$.

► **Definition 5** (Reduction). *The reduction rules are as follows.*

$$\begin{array}{ll} \text{Beta:} & [N]a.a\langle x \rangle.M \rightarrow \{N/x\}M \\ \text{Passage:} & [N]b.a\langle x \rangle.M \rightarrow a\langle x \rangle.[N]b.M \quad (a \neq b, x \notin \text{fv}(N)) \\ \text{Next:} & \star; M \rightarrow M \\ \text{Prefix (pop):} & a\langle x \rangle.N; M \rightarrow a\langle x \rangle.(N; M) \quad (x \notin \text{fv}(M)) \\ \text{Prefix (push):} & [P]a.N; M \rightarrow [P]a.(N; M) \\ \text{Associate:} & (P; N); M \rightarrow P; (N; M) \end{array}$$

Reduction contexts C are given by the following grammar.

$$C ::= \{ \} \mid a\langle x \rangle.C \mid [M]a.C \mid [C]a.M \mid C; M \mid M; C$$

Spine contexts W are given by the following grammar.

$$W ::= \{ \} \mid a\langle x \rangle.W \mid [M]a.W \mid W; M \mid M; W$$

The (capturing) substitution of the hole $\{ \}$ in a context C by a term M is written $C\{M\}$. The reduction relation \rightarrow is given by closing the reduction rules $M \rightarrow N$ under any context, $C\{M\} \rightarrow C\{N\}$. Spine reduction \rightarrow_w is given by closing under spine contexts, $W\{M\} \rightarrow_w W\{N\}$. The reflexive-transitive closure of a relation \rightarrow is written \rightarrow^* , and reduction to normal form as \rightarrow^\dagger .

Without the beta-rule, reduction is strongly normalizing.

► **Proposition 6.** *Non-beta reduction is SN.*

Proof. The following pair of measures strictly decreases. Writing $|M|$ for the size of a term M , first the sum over $|M|$ for every subterm $M; N$ (equal for *passage*, decreasing for other rules); second the sum over $|M|$ for every subterm $[N]a.M$ (decreasing for *passage*). ◀

The above definitions differ from the original presentation [24] in the following ways, and for the following reasons. First, instead of introducing sequencing $M; N$ as a primitive, the original syntax used a sequential variable $x.M$ and defined sequencing as an operation. The two formulations are equivalent modulo the reductions for sequencing (the Next, Prefix, and Associate reductions), whose normal forms use sequencing exclusively as $x; M$. The original formulation reduced the number of reduction rules and avoided the need for a continuation stack on the machine, emphasizing the view of terms as sequences of machine instructions. Present thinking however prefers simplicity not through fewer but through more elementary components, viewing the more refined reduction relation and the continuation stack as benefits rather than obstacles. The current syntax emphasizes the clean combination of λ -calculus and sequencing and more readily accommodates current and future extensions such as exceptions [25] and concurrency.

Second, the original formulation used a single yet complex reduction rule, below, where the ellipsis represents any number of abstractions and applications on locations other than a .

$$[N]a \dots a \langle x \rangle. M \rightarrow \dots \{N/x\}M$$

Again the emphasis has shifted to multiple simpler rules, Beta and Passage. These follow the natural case analysis for an application meeting an abstraction, whether they have the same or different locations, and give more structured normal forms, where all abstractions precede all applications, as we will see in Sections 4 and 6.

The main technical results of the paper, in the following sections, will establish the relation between the operational semantics, reduction, and quantitative types. The latter are defined next.

► **Definition 7** (Quantitative types). *Quantitative types for the FMC are built up in four mutually recursive layers : computation types, collection types, vector types, and memory types, given by the following grammars.*

$$\begin{array}{ll} \text{Computation types:} & \tau ::= \bar{\tau} \Rightarrow \bar{\kappa} \\ \text{Collection types:} & \iota, \kappa, \mu, \nu, \pi ::= [\tau_1, \dots, \tau_n] \\ \text{Vector types:} & \bar{\iota} ::= \iota_1 \dots \iota_n \\ \text{Memory types:} & \bar{\iota} ::= \{\bar{\iota}_a \mid a \in A\} \end{array}$$

Collection types are multisets, i.e. considered modulo associativity, symmetry, and unitality, where multiset union is denoted as a sum, $\iota + \kappa$.

These four categories reflect the components of the operational semantics. Terms will be typed with computation types (or simply *types*) $M : \tau$, as will continuation stacks $K : \tau$. Collection types may be viewed as *value types*, along the computation/value distinction of call-by-push-value [32], and apply to the argument term in an application and to the terms on an argument stack (note the continuation stack holds terms as *computations*). Argument stacks themselves are typed by vector types, $S : \bar{\iota}$, and memories by memory types, $S_A : \bar{\iota}$.

Note that the base cases for the definition are given by empty collection types $[]$ and empty vector and memory types, both denoted ε . Composition of type vectors is by juxtaposition, $\bar{\iota} \bar{\kappa}$, extended to memories point-wise, $\bar{\iota} \bar{\kappa} = \{\bar{\iota}_a \bar{\kappa}_a \mid a \in A\}$. The singleton family $a(\bar{\iota})$ has $\bar{\iota}$ at location a and ε at any other location. The default location λ may again be omitted, writing $\bar{\iota}$ for $\lambda(\bar{\iota})$.

Since stacks are first-in-first-out, identity terms are of the form below, with pops and pushes in inverse order to each other. To match this in the type system, type vectors on the left of an implication are implicitly reversed, so that identity types may be written $\bar{\iota} \Rightarrow \bar{\iota}$.

$$a_n \langle x_n \rangle \dots a_1 \langle x_1 \rangle. [x_1] a_1 \dots [x_n] a_n. \star : a_n(\iota_n) \dots a_1(\iota_1) \Rightarrow a_1(\iota_1) \dots a_n(\iota_n)$$

A *typing context* $\Gamma, \Delta, \Lambda, \Sigma$ is a function from variables to collection types, written as a sequence $\Gamma = x_1 : \iota_1, \dots, x_n : \iota_n$ where any other variable y is implicitly assigned the empty type, $\Gamma(y) = []$. The *union* or *sum* of two contexts $\Gamma + \Delta$ is given pointwise: $(\Gamma + \Delta)(x) = \Gamma(x) + \Delta(x)$.

► **Definition 8** (The quantitatively typed FMC). *The typing rules in Figure 1 define the quantitatively-typed Functional Machine Calculus, in two variants: weak and strong, with typing judgements $\Gamma \vdash_w M : \tau$ respectively $\Gamma \vdash_s M : \tau$ that assign the term M the type τ in the context Γ . The weak type system is given by the rules of Figure 1 with unlabelled (\vdash)*

and weak judgements (\vdash_w), and the strong system by those with unlabelled (\vdash) and strong judgements (\vdash_s). The weight n of a derivation is the number of rule instances for abstraction, application, sequencing, and unit, and may be indicated in a judgement as \vdash^n .

The first section of Figure 1 gives the weak system for *terms*, the next section extends the weak system to memories, continuation stacks, and machine states, and the final section gives the specific rules for the strong type system (which is defined only for terms, not states). When it is clear from the context whether a judgement concerns the weak or the strong type system, the subscripts on \vdash_w and \vdash_s may be dropped.

The key difference between both systems is in the third premise to the strong application rule. In both application rules, the first premise assigns a type to the argument N which may be empty in the case where it will not be used by M . This corresponds to machine evaluation, where N may be discarded without evaluation, but not to strong normalization, where N must itself be normalizing. The strong application rule therefore requires N to be typed separately, a standard approach [8, 11], here via the third premise.

Typing judgements for machine states are of the form $\vdash (S_A, M, K): \varepsilon \Rightarrow \bar{\ell}$, with empty context and input type. This prevents failure states from being typed: the empty context means there are no computation types for free variables, and an empty input type means there must always be sufficient items in memory for any abstraction to pop. *Progress*, the idea that a machine state is either final or has a transition, is thus immediate from types, once it is proved that machine transitions preserve typing in Lemma 10 below.

Non-idempotent intersection types for the λ -calculus [11] embed into quantitative FMC types in the manner of simple types [24], by the following interpretation, using the default location λ (omitted from the notation). The weak type system then embeds the system \mathcal{W} for λ -terms of [11], while the strong system differs from system \mathcal{S} in two minor ways: in the presence of the weakening rule, and for the strong application rule, in requiring a witness also when the argument has a non-empty collection type.

Base type: $o = \varepsilon \Rightarrow \varepsilon$
 Arrow types: $[\sigma_1, \dots, \sigma_n] \rightarrow \tau = [\sigma_1, \dots, \sigma_n] \bar{\kappa} \Rightarrow \bar{\mu}$ where $\tau = \bar{\kappa} \Rightarrow \bar{\mu}$

3 Machine Termination

This section will show that the successful evaluation and weak typeability coincide, where the weights agree. First, in the weak type system, the following *substitution* typing rule is admissible: it can be introduced without altering the expressiveness of the type system.

$$\frac{\Gamma \vdash N: \iota \quad \Delta, x: \iota \vdash M: \tau}{\Gamma + \Delta \vdash \{N/x\}M: \tau} \text{sub}$$

► **Lemma 9** (Substitution, weak). *There are derivations $\Gamma \vdash_w^n N: \nu$ and $\Delta, x: \nu \vdash_w^m M: \tau$ if and only if there is a derivation $\Gamma + \Delta \vdash_w^k \{N/x\}M: \tau$ where $k = n + m$.*

For the weakly typed calculus, a machine step from a typed state leads to a typed state of weight exactly one less.

► **Lemma 10** (Evaluation reduces typing weights). *If*

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

then $\vdash_w^n (S_A, M, K): \varepsilon \Rightarrow \bar{\ell}$ if and only if $\vdash_w^{n-1} (T_A, N, L): \varepsilon \Rightarrow \bar{\ell}$.

Variable:	$\frac{}{\mathbf{x} : [\tau] \vdash \mathbf{x} : \tau} \text{var}$
Abstraction:	$\frac{\Gamma, \mathbf{x} : \iota \vdash M : \bar{\kappa} \Rightarrow \bar{\mu}}{\Gamma \vdash a\langle \mathbf{x} \rangle . M : a(\iota) \bar{\kappa} \Rightarrow \bar{\mu}} \text{abs}$
Application (weak):	$\frac{\Gamma \vdash_w N : \iota \quad \Delta \vdash_w M : a(\iota) \bar{\kappa} \Rightarrow \bar{\mu}}{\Gamma + \Delta \vdash_w [N]a . M : \bar{\iota} \Rightarrow \bar{\mu}} \text{app}$
Unit:	$\frac{}{\vdash \star : \bar{\iota} \Rightarrow \bar{\iota}} \text{unit}$
Sequencing:	$\frac{\Gamma \vdash N : \bar{\iota} \Rightarrow \bar{\kappa} \quad \Delta \vdash M : \bar{\kappa} \Rightarrow \bar{\mu}}{\Gamma + \Delta \vdash N ; M : \bar{\iota} \Rightarrow \bar{\mu}} \text{seq}$
Collection:	$\frac{(\Gamma_i \vdash M : \tau_i)_{1 \leq i \leq n}}{\Gamma_1 + \dots + \Gamma_n \vdash M : [\tau_1, \dots, \tau_n]} \text{col}$
<hr/>	
Memory:	$\frac{}{\vdash \varepsilon_A : \varepsilon} \text{m0} \quad \frac{\vdash S_A : \bar{\iota} \quad \vdash M : \kappa}{\vdash S_A \cdot M_a : \bar{\iota} a(\kappa)} \text{m1}$
Continuation stack:	$\frac{}{\vdash \varepsilon : \bar{\iota} \Rightarrow \bar{\iota}} \text{k0} \quad \frac{\Gamma \vdash M : \bar{\iota} \Rightarrow \bar{\kappa} \quad \vdash K : \bar{\kappa} \Rightarrow \bar{\mu}}{\vdash M K : \bar{\iota} \Rightarrow \bar{\mu}} \text{k1}$
State:	$\frac{\vdash S_A : \bar{\iota} \quad \vdash M : \bar{\iota} \Rightarrow \bar{\kappa} \quad \vdash K : \bar{\kappa} \Rightarrow \bar{\mu}}{\vdash (S_A, M, K) : \varepsilon \Rightarrow \bar{\mu}} \text{st}$
<hr/>	
Application (strong):	$\frac{\Gamma \vdash_s N : \iota \quad \Delta \vdash_s M : a(\iota) \bar{\kappa} \Rightarrow \bar{\mu} \quad \Lambda \vdash_s N : \tau}{\Gamma + \Delta + \Lambda \vdash_s [N]a . M : \bar{\kappa} \Rightarrow \bar{\mu}} \text{APP}$
Weakening:	$\frac{\Gamma \vdash_s M : \tau}{\Gamma + \Delta \vdash_s M : \tau} \text{WEAK}$

■ **Figure 1** The quantitatively typed Functional Machine Calculus.

It follows that weak types guarantee termination of the machine while measuring the length of a run.

► **Theorem 11** (Weak types quantify evaluation). *For a typed state $\vdash_w^n (S_A, M, \varepsilon) : \varepsilon \Rightarrow \bar{l}$ there exists a typed memory $\vdash_w^m T_A : \bar{l}$ such that $S_A, M \Downarrow_k T_A$ where $n = m + k$.*

Proof. Recall that failure states cannot be typed. A typed state $\vdash_w^n (S_A, M, K) : \varepsilon \Rightarrow \bar{l}$ is then either final or has a machine transition. In the case of a transition, by Lemma 10 the new state is typed with weight one less. Thus, the machine continues taking steps, preserving types but reducing weights, until it reaches a final state, giving a run of length k as below left, so that $S_A, M \Downarrow_k T_A$ by Proposition 4. The final state $(T_A, \star, \varepsilon)$ is typed as below right, with weight $m + 1$ where $\vdash_w^m T_A : \bar{l}$. This represents a run of length 1, so that a run of length k starts from a weight $m + k = n$.

$$\frac{(\overline{S_A, M, \varepsilon})_k}{(\overline{T_A, \star, \varepsilon})} \quad \frac{\vdash_w^m T_A : \bar{l} \quad \frac{\vdash_w^1 \star : \bar{l} \Rightarrow \bar{l}}{\vdash_w^0 \varepsilon : \bar{l} \Rightarrow \bar{l}}}{\vdash_w^{m+1} (T_A, \star, \varepsilon) : \varepsilon \Rightarrow \bar{l}} \text{st} \quad \blacktriangleleft$$

For the reverse, that all terminating states can be typed, it is now sufficient to give a type to final states $(T_A, \star, \varepsilon)$. We may do so simply by typing each term in the memory T_A with the empty collection type \square . Denote by \square^n the type vector of $\square \dots \square$ of length n , and by \square^f for a function $f : A \rightarrow \mathbb{N}$ the memory type $\{\square^{f(a)} \mid a \in A\}$. Let $|T|$ denote the length of a stack T and let $|T_A| = f$ where $f(a) = |T_a|$ denote the dimensions of a memory T_A . For types, likewise let $|\bar{l}|$ denote the length of a type vector and $|\bar{l}| = f$ where $f(a) = |\bar{l}_a|$ the dimensions of a memory type.

► **Theorem 12** (Evaluation implies weak typing). *If $S_A, M \Downarrow_n T_A$ then $\vdash_w^n (S_A, M, \varepsilon) : \varepsilon \Rightarrow \square^f$ where $f = |T_A|$.*

Proof. The memory T_A may be typed $\vdash_w^0 T_A : \square^f$, and its final state $\vdash_w^0 (T_A, \star, \varepsilon) : \varepsilon \Rightarrow \square^f$. By Proposition 4 there is a run of length n to this state from (S_A, M, ε) . By Lemma 10 each step in this run, backwards from the final state, gives a previous state with the same type but weight one greater, so that by induction $\vdash_w^n (S_A, M, \varepsilon) : \varepsilon \Rightarrow \square^f$. ◀

Combining both theorems, since a typeable state $\vdash_w (S_A, M, \varepsilon) : \varepsilon \Rightarrow \bar{l}$ terminates with a memory $T_A : \bar{l}$, which is also typed by \square^f where $f = |\bar{l}|$, we have $\vdash_w^n (S_A, M, \varepsilon) : \varepsilon \Rightarrow \square^f$ where the weight n of the latter is exactly the number of machine steps to successful termination.

4 Spine Normalization

The present section will establish that weak typing and successful evaluation further coincide with spine normalization. First we observe the shape of spine normal forms and demonstrate that they may be typed.

► **Proposition 13** (Spine normal forms). *The normal forms of spine reduction \rightarrow_w are the terms \mathcal{W} given by the following mutually recursive grammars.*

$$\begin{aligned} \mathcal{W} &::= a\langle x \rangle. \mathcal{W} \mid \mathcal{V} \\ \mathcal{V} &::= [M]a. \mathcal{V} \mid x ; \mathcal{W} \mid x \mid \star \end{aligned}$$

► **Lemma 14** (Spine normal forms are weakly typeable). *For any \mathcal{W} there is a derivation $\Gamma \vdash_w \mathcal{W} : \bar{l} \Rightarrow \square^f$.*

24:10 Quantitative Types for the FMC

Spine reduction preserves and reflects typing, where weights are strictly reduced in the reduction rules *beta* ($[N]a.a\langle x \rangle.M \rightarrow \{N/x\}M$) and *next* ($\star ; M \rightarrow M$), and preserved otherwise.

► **Lemma 15** (Weighted subject reduction and expansion). *If $M \rightarrow_w N$ then $\Gamma \vdash_w^m M : \tau$ if and only if $\Gamma \vdash_w^n N : \tau$, where $m = n + 2$ in the case of a beta-step or next-step and $m = n$ otherwise.*

It now follows that weak typeability and spine normalization coincide.

► **Theorem 16** (Weak typing characterizes spine normalization). *A term is weakly typeable if and only if it is spine-normalizing.*

Proof. From left to right, since non-beta steps are strongly normalizing by Proposition 6, an infinite reduction must contain infinitely many beta-steps. By Lemma 15 spine beta-reduction decreases the weight of the typing derivation, while other steps do not increase it. Then spine reduction is terminating. From right to left, proceed by induction on the sum length of all spine reduction paths. Spine-normal terms are typeable by Lemma 14, and for a step $M \rightarrow_w N$ if N is typeable then so is M by Lemma 15. ◀

Since weak typing characterizes both spine normalization and machine termination (by Theorems 11 and 12), the natural corollary would be that machine termination and spine normalization coincide. In one direction, this follows directly: by Theorem 12 termination implies typeability, and by Theorem 16 typeability implies spine normalization.

In the other direction, however, there is still a small gap to be closed: a spine-normalizing term M may be typeable as $\Gamma \vdash_w M : \bar{\ell} \Rightarrow \bar{\kappa}$, but the machine is only guaranteed to run for states of type $\varepsilon \Rightarrow \bar{\kappa}$. In other words, for M to run successfully, a memory S_A of type $\vdash_w S_A : \bar{\ell}$ must be provided, and each variable $x : \iota$ in Γ must be substituted by a term N typed $\vdash_w N : \iota$. However, these are not guaranteed to exist, since not all quantitative types are inhabited. This is in contrast with simple types for the FMC, which are inhabited [24, Remark 3.7].

► **Proposition 17** (No type inhabitation). *There exist types τ such that $\nexists M : \tau$ for any M .*

Proof. Since types are preserved under spine reduction, if a type is inhabited by a term, it is inhabited by its spine normal form. The only normal form inhabiting the type $\varepsilon \Rightarrow \varepsilon$ is $\vdash \star : \varepsilon \Rightarrow \varepsilon$ (it cannot be an abstraction, since the type has no inputs; it cannot contain free variables, since the context is empty; and it cannot be an application, since the type has no outputs). Similarly, the only spine normal forms inhabiting $\varepsilon \Rightarrow [\varepsilon \Rightarrow \varepsilon]$ are of the form $\vdash [M].\star : \varepsilon \Rightarrow [\varepsilon \Rightarrow \varepsilon]$. It follows that the collection type $[\varepsilon \Rightarrow \varepsilon, \varepsilon \Rightarrow [\varepsilon \Rightarrow \varepsilon]]$ is not inhabited, and so neither is the type $\varepsilon \Rightarrow [\varepsilon \Rightarrow \varepsilon, \varepsilon \Rightarrow [\varepsilon \Rightarrow \varepsilon]]$. ◀

To connect spine normalization and termination, for every spine normal term \mathcal{W} a memory S_A and substitution map σ will be given such that for S_A and $\sigma\mathcal{W}$ the machine terminates. A *substitution map* $\sigma = \{M_1/x_1, \dots, M_n/x_n\}$ is a finite function from variables to terms, applied as σM to a term M as a simultaneous substitution. To allow a successful machine run for $\sigma\mathcal{W}$, the memory S_A and substitution map σ must be such that, at any point in the computation, there are sufficient terms on all stacks to pop from. To this end the following notions are defined: for a natural number d , a *memory of dimension d* will have at least d terms of dimension $d - 1$ on each stack, and a *term of dimension d* consists of sufficient applications to generate a memory of dimension d . These are formalized as follows, again noting that the locations A for a given term are finite and may be determined syntactically.

For a stack $S = \varepsilon N_1 \dots N_n$ let $[S]a.M$ denote the term $[N_1]a \dots [N_n]a.M$, and for a memory S_A let $[S_A].M$ denote a term $[S_{a_1}]a_1 \dots [S_{a_n}]a_n.M$ where $A = \{a_1 \dots a_n\}$. For a natural number d , a *memory of dimension d* is one S_A where $|S_A| = f$ with $f(a) \geq d$ for all $a \in A$, and each term in S_A is a term of dimension $d - 1$, where a *term of dimension d* is of the form $[S_A].\star$ with S_A of dimension d . A *substitution map of dimension d* is one that sends every variable to a term of dimension d . Observe that an element of dimension d is also one of any $d' < d$.

► **Lemma 18** (Spine normal forms terminate). *For a spine-normal term \mathcal{W} there is a dimension d such that for any memory S_A and substitution map σ of dimension d , there is a memory T_A such that $S_A, \sigma\mathcal{W} \Downarrow T_A$.*

Since machine termination guarantees typing (Theorem 12), the above would be sufficient to connect spine normalization to weak typeability. The following lemma is however instructive in connecting spine reduction and machine termination explicitly.

► **Lemma 19** (Evaluation is invariant under spine reduction). *If $M \rightarrow_w N$ then $S_A, \sigma M \Downarrow T_A$ if and only if $S_A, \sigma N \Downarrow T_A$ for any memories S_A and T_A and substitution map σ .*

The main theorem connecting weak typing, spine normalization, and machine termination is then the following.

► **Theorem 20** (Weak types characterize termination and spine normalization). *For a term M the following are equivalent:*

1. M is weakly typeable $\Gamma \vdash_w M : \tau$,
2. M is spine-normalizing,
3. M is machine-terminating: there are memories S_A, T_A and a substitution map σ such that $S_A, \sigma M \Downarrow T_A$.

Proof. By Theorem 16 statements 1 and 2 are equivalent. From statement 2 to 3, if M spine normalizes to \mathcal{W} then Lemma 18 gives memories S_A and T_A and a substitution map σ such that $S_A, \sigma\mathcal{W} \Downarrow T_A$, and since machine evaluation is reflected by spine reduction (Lemma 19), also $S_A, \sigma M \Downarrow T_A$. From 3 to 1, by Theorem 12 for $S_A, \sigma M \Downarrow T_A$ we get $\vdash_w (S_A, \sigma M, \varepsilon) : \varepsilon \Rightarrow \boxed{f}$ where $f = |T_A|$. The typing rule for states gives $\vdash_w S_A : \bar{\tau}$ and $\vdash_w \sigma M : \bar{\tau} \Rightarrow \boxed{f}$ for some $\bar{\tau}$. By the weak substitution lemma (Lemma 9), since σM is typeable so is M , and we have $\Gamma \vdash_w M : \bar{\tau} \Rightarrow \boxed{f}$ for some Γ . ◀

Note that the above theorem resolves the problem of uninhabited types as follows: if M is typed $\Gamma \vdash_w M : \tau$ where either Γ or τ contains an uninhabited (input) type, then (by 1 \Rightarrow 2) it has a spine normal form, which in turn (by 2 \Rightarrow 3) evaluates on the machine, which then (by 3 \Rightarrow 1) guarantees that M may also be typed with only inhabited types.

5 Strong Normalization

Where the weak type system characterizes machine evaluation and spine normalization, the strong type system characterizes strong normalization (SN). This section will prove one direction, that normalization reduces the weight of strong type derivations and, hence, that strong quantitative types guarantee SN and give upper bounds to the length of reductions.

We start with the substitution lemma for strong types. The substitution typing rule can be eliminated, maintaining or reducing weight, and introduced, maintaining weight. The inequality is due to the weakening rule WEAK: substitution into a weakened part of the context erases the typing information of a term, maintaining only the types for its context.

► **Lemma 21** (Substitution, strong). *If there are derivations $\Gamma \vdash_s^n N : \nu$ and $x : \nu, \Delta \vdash_s^m M : \tau$ then there is a derivation $\Gamma + \Delta \vdash_s^k \{N/x\}M : \tau$ where $k \leq n + m$, and vice versa where $k = n + m$.*

The following lemma gives preservation of typing for beta-reduction, reducing weights.

► **Lemma 22** (Subject reduction, beta). *If $M \rightarrow N$ by a beta-step and $\Gamma \vdash_s^m M : \tau$ then $\Gamma \vdash_s^n N : \tau$ where $m > n$.*

While we have subject reduction, note that we do not have subject *expansion*. In the reduction $[N]a.a(x).M \rightarrow \{N/x\}M$, if x is not free in M , there is no typing information for N in the reduct, but this is required for the strong application rule in the redex. This familiar situation presents an obstacle in demonstrating that all strongly normalising terms can be typed, covered in the next section.

For the remaining reduction rules, we do have subject expansion as well as reduction. Recall that the *next* reduction, $\star ; M \rightarrow M$, reduces weights, while the other non-beta reduction rules maintain weights. It would be possible to use an adjusted weight that counts *only* application and abstraction typing rules, to give exact weights for non-beta subject reduction, since non-beta rules are inherently terminating by Proposition 6.

► **Lemma 23** (Subject reduction and expansion, non-beta). *If $M \rightarrow N$ by a non-beta-step then $\Gamma \vdash_s^m M : \tau$ if and only if $\Gamma \vdash_s^n N : \tau$ where $m \geq n$.*

We conclude this section with the theorem that typed terms are strongly normalizing.

► **Theorem 24** (Strong normalization). *If $\Gamma \vdash_s M : \tau$ then M is strongly normalizing.*

Proof. By *subject reduction*, Lemmata 22 and 23, reduction preserves types while maintaining or reducing weights. Since beta-steps strictly reduce the weight of a derivation and non-beta-steps are strongly normalizing by Proposition 6, all reduction paths must be finite. ◀

6 Perpetual Evaluation

For the reverse direction, that all strongly normalizing terms are typeable, the challenge is that of *weakened* terms, the argument terms N in a reduction $[N]a.a(x).M \rightarrow \{N/x\}M$ where x is not free in M . Reasoning by beta-expansion, to derive SN for $[N]a.a(x).M$ we need termination information for N , which the reduct $\{N/x\}M = M$ does not immediately provide. This problem constitutes the essential difference between the λI - and λK -calculus (see [3]) and is familiar from many strong normalization arguments, in particular those deriving SN from weak normalization by maintaining the weakened term N in the term structure [29, 35] and those using perpetual reduction strategies [6, 43].

The present approach will be a variant of perpetual reduction. We will see a *perpetual evaluation* relation $M \blacktriangleright \mathcal{M}$ which relates a term M to its normal form \mathcal{M} , that in the case of a redex $[N]a.a(x).M$ not only evaluates $\{N/x\}M$ but in addition requires an evaluation $N \blacktriangleright \mathcal{M}'$. This definition, as a big-step evaluation relation instead of a strategy, avoids maintaining weakened terms in the syntax, either as redexes or explicit substitutions.

The proof then becomes straightforward: strong normalizability of M implies perpetual evaluation $M \blacktriangleright \mathcal{M}$, implies strong typeability $\Gamma \vdash_s M : \tau$. As before, we begin by observing the shape of normal forms.

► **Proposition 25** (Normal forms). *The normal forms of \rightarrow are the terms \mathcal{M} given by the following grammars.*

$$\begin{aligned} \mathcal{M} &::= a(x).\mathcal{M} \mid \mathcal{N} \\ \mathcal{N} &::= [\mathcal{M}]a.\mathcal{N} \mid x;\mathcal{M} \mid x \mid \star \end{aligned}$$

Beta	$\frac{[S_A].\{N/x\}M \triangleright \mathcal{M} \quad N \triangleright \mathcal{M}'}{[S_A].[N]a.\langle x \rangle.M \triangleright \mathcal{M}} \quad \frac{[S_A].a.\langle x \rangle.[N]b.M \triangleright \mathcal{M}}{[S_A].[N]b.a.\langle x \rangle.M \triangleright \mathcal{M}}$	Passage
Next	$\frac{[S_A].M \triangleright \mathcal{M}}{[S_A].(\star; M) \triangleright \mathcal{M}} \quad \frac{[S_A].a.\langle x \rangle.(N; M) \triangleright \mathcal{M}}{[S_A].(a.\langle x \rangle.N; M) \triangleright \mathcal{M}}$	Prefix (pop)
Associate	$\frac{[S_A].(P; (N; M)) \triangleright \mathcal{M}}{[S_A].((P; N); M) \triangleright \mathcal{M}} \quad \frac{[S_A].[P]a.(N; M) \triangleright \mathcal{M}}{[S_A].([P]a.N; M) \triangleright \mathcal{M}}$	Prefix (push)
Normal (abstraction)	$\frac{M \triangleright \mathcal{M}}{a.\langle x \rangle.M \triangleright a.\langle x \rangle.\mathcal{M}}$	
Normal (unit)	$\frac{(N_i \triangleright \mathcal{M}_i)_{i \leq n}}{[N_1]a_1 \dots [N_n]a_n.\star \triangleright [\mathcal{M}_1]a_1 \dots [\mathcal{M}_n]a_n.\star}$	
Normal (variable)	$\frac{(N_i \triangleright \mathcal{M}_i)_{i \leq n}}{[N_1]a_1 \dots [N_n]a_n.x \triangleright [\mathcal{M}_1]a_1 \dots [\mathcal{M}_n]a_n.x}$	
Normal (sequence)	$\frac{(N_i \triangleright \mathcal{M}_i)_{i \leq n} \quad M \triangleright \mathcal{M}}{[N_1]a_1 \dots [N_n]a_n.(x; M) \triangleright [\mathcal{M}_1]a_1 \dots [\mathcal{M}_n]a_n.(x; \mathcal{M})}$	

■ **Figure 2** Perpetual evaluation.

Perpetual evaluation will reduce terms to normal form by *iterated weak head reduction*. That is, terms are reduced in a context of applications, a *weak head context*, which functions similarly to the memory of the abstract machine. Where the machine would reach a *final* or *failure* state, reduction continues (i.e. is *iterated*) for the remaining subterms, which on the machine would constitute the memory and continuation stack.

We will denote a term M in weak head context $[N_1]a_1 \dots [N_n]a_n.M$ as $[S_A].M$, where $S_A = \varepsilon \cdot (N_1)_{a_1} \dots (N_n)_{a_n}$ is the memory corresponding to the sequence of applications $[N_1]a_1$ through $[N_n]a_n$. Note that the notation $[S_A]$ is not a function on S_A , since multiple application sequences may correspond to the same memory (to be exact: those modulo $[P]b.[N]a.M \sim [N]a.[P]b.M$ where $a \neq b$). Typing, however, collapses these distinctions again: a term $[S_A].\star$ will have the type $\varepsilon \Rightarrow \bar{t}$ where $S_A : \bar{t}$.

► **Definition 26** (Perpetual evaluation). *The perpetual evaluation relation $M \triangleright \mathcal{M}$ between terms and normal forms is given inductively by the rules in Figure 2.*

The salient features of perpetual evaluation are the following. The first six rules each correspond to a reduction rule in weak head context. Simultaneously, these can be seen to implement machine evaluation, where the context represents the memory. The beta-rule requires the additional normalization of the argument N , as discussed above. The remaining four rules cover four of the five cases of normal forms, with the fifth, for application, covered by the presence of weak head contexts. As discussed above, these cases represent final or failure states on the machine, where reduction needs to iterate on the subterms representing the remaining memory and continuation stack.

The following lemma then shows that a strongly normalizing term also has a finite perpetual evaluation. To avoid reference to confluence, it is phrased as a combination of two statements: one, that SN implies perpetual evaluation, and two, that the normal form given by perpetual evaluation is indeed a normal form of the original term.

► **Lemma 27** (Strong normalization implies perpetual evaluation). *If M is strongly normalizing then $M \blacktriangleright \mathcal{M}$ and $M \rightarrow_{\text{p}} \mathcal{M}$ for some \mathcal{M} .*

The definition of perpetual evaluation as an inductive relation means we may assign the same type to a term and its normal form simultaneously.

► **Lemma 28** (Perpetual evaluation implies strong typeability). *If $M \blacktriangleright \mathcal{M}$ then $\Gamma \vdash_{\text{s}} M : \tau$ and $\Gamma \vdash_{\text{s}} \mathcal{M} : \tau$ for some Γ and τ .*

This gives the required equivalence between strong quantitative typing, strong normalization, and perpetual evaluation.

► **Theorem 29** (Strong typing characterizes strong normalization). *For a term M , the following are equivalent:*

1. M is strongly typeable, $\Gamma \vdash_{\text{s}} M : \tau$,
2. M is strongly normalizing,
3. there is a perpetual evaluation $M \blacktriangleright \mathcal{M}$.

Proof. From 1 to 2 is Theorem 24, from 2 to 3 is Lemma 27, and from 3 to 1 is Lemma 28. ◀

7 Quantitative Types for Higher-Order Store

The quantitative type system extends to computational effects and evaluation strategies via their encoding in the FMC. We will explore this by demonstrating that the present type system agrees with the intersection types of De'Liguoro and Treglia for a monadic λ -calculus with higher-order store [14]. The comparison will be exact, but informal, for the reason that their type system is *idempotent*, i.e. their type collections are sets, not multisets. Modulo this distinction, their typed calculus will embed directly into the FMC. Appendix A in addition gives the correspondence with the related quantitative types of Alves, Kesner, and Ramos [1], a correspondence that is likewise informal but exact, as it is restricted to a fragment of their type system.

The untyped calculus consists of six syntactic constructs, which embed as follows, where the *values* V are the first column and the *computations* M the second and third. The embedding into the FMC uses the default location λ for regular application, abstraction, and monadic constructs, omitted from the notation both in terms and types, i.e. $[N].M$ and $\langle x \rangle.M$ abbreviate $[N]\lambda.M$ and $\lambda\langle x \rangle.M$, and ι may abbreviate the singleton memory $\lambda(\iota)$.

Values V :	Computations M :	
$x = x$	$[V] = [V].\star$	$\text{get}_a(\lambda x.M) = a\langle x \rangle.[x]a.M$
$\lambda x.M = \langle x \rangle.M$	$M \gg V = M; V$	$\text{set}_a(V, M) = a\langle _ \rangle.[V]a.M$

The current presentation uses $M \gg V$ instead of De'Liguoro and Treglia's $M \star V$ to avoid conflicting with *skip*, and $a, b, c \dots$ instead of ℓ for memory locations. The embeddings are standard [17, 24, 42]. They further agree with the encodings of application, let-syntax, and *update* $a := M$ and lookup $!a$, as follows (the first column gives the encodings by De'Liguoro and Treglia [14], the second their standard FMC interpretation [24]).

$VW = [W] \gg V$	$[W].V \leftarrow ([W].\star); V$
$\text{let } x = !a \text{ in } M = \text{get}_a(\lambda x.M)$	$(a\langle x \rangle.[x]a.[x].\star); \langle x \rangle.M \rightarrow_{\text{p}} a\langle x \rangle.[x]a.M$
$a := V; M = \text{set}_a V, M$	$(a\langle _ \rangle.[V]a.\star); M \rightarrow_{\text{p}} a\langle _ \rangle.[V]a.M$

Types for the monadic calculus, given below, are stratified like those of the FMC, but unlike the latter they include an intersection operator \wedge and unit ω at each level.

$$\begin{aligned} \text{Value types:} \quad & \delta ::= \alpha \mid \delta \rightarrow \tau \mid \delta \wedge \delta' \mid \omega \\ \text{Store types:} \quad & \sigma ::= \langle a : \delta \rangle \mid \sigma \wedge \sigma' \mid \omega \\ \text{Configuration types:} \quad & \kappa ::= \delta \times \sigma \mid \kappa \wedge \kappa' \mid \omega \\ \text{Result types:} \quad & \tau ::= \sigma \rightarrow \kappa \mid \tau \wedge \tau' \mid \omega \end{aligned}$$

A subtyping relation \leq and equivalence $=$ for these types helps embed them into FMC types. First, implication distributes over intersection, $\delta \rightarrow (\tau \wedge \tau') = (\delta \rightarrow \tau) \wedge (\delta \rightarrow \tau')$, so that value types are given by a large intersection over arrow types $\delta \rightarrow \tau$ and variables α . Value types then encode as collection types, where $\delta \wedge \delta' = \delta + \delta'$ and $\omega = []$, and arrow types by their standard encoding [24], where $\delta \rightarrow \tau$ adds the additional input δ to the FMC implication type given by τ . Since value types are collections, $\delta \rightarrow \tau$ embeds as a singleton collection, as follows.

$$\delta \rightarrow \tau = [\delta \overline{\mu} \Rightarrow \overline{\nu}] \quad \text{where } \tau = \overline{\mu} \Rightarrow \overline{\nu}$$

For store types, collections distribute over location indexing, $\langle a, \delta \rangle \wedge \langle a, \delta' \rangle = \langle a, \delta \wedge \delta' \rangle$. A store type is thus of the following form, and embeds as a memory type over a set of locations A representing (non-empty) memory cells.

$$\bigwedge_{a \in A} \langle a : \delta_a \rangle = \{\delta_a \mid a \in A\}$$

Note that store types use the intersection operator to combine the types at different locations. Configuration types κ are then memory types over a set of locations A that include the default location λ , in the first line below, and computation types τ are implications from store types to memory types, second line below.

$$\begin{aligned} \delta \times \sigma &= \overline{\iota} \quad \text{where } \overline{\iota}_\lambda = \delta \text{ and } \overline{\iota}_a = \sigma_a \text{ for } a \neq \lambda \\ \sigma \rightarrow \kappa &= \sigma \Rightarrow \kappa \end{aligned}$$

Figure 3 gives the comparison between intersection typing for store and quantitative FMC typing. With the difference that the former admits weakening and contraction, whereas for the latter contexts are linear, the relation is an embedding that exactly follows those of terms and types.

8 Conclusion

Quantitative types for the FMC arise from simple types by the standard generalization, of replacing input types (or *value* types) with multisets. Nevertheless, through the encodings of effects and strategies in the FMC, this gives a quantitative characterization of effectful higher-order programs, in agreement with known results.

Of the technical contributions of the paper, two deserve closer attention. One is the exact quantification of machine evaluation by the weak type system, simply by counting the typing rules corresponding to machine steps, without the need for dedicated technical adaptations. Fundamentally this relies on the extension of the Krivine machine with sequencing and, in particular, *skip*, as signifying successful termination. The direct correspondence between weak quantitative types and machine evaluation underlines how this is an interesting and natural extension of the λ -calculus that broadens our perspective on higher-order computation.

The second main technical contribution is the perpetual evaluation relation. Its equivalence with strong normalization holds independently of the strong type system, and it is expected to prove useful in strong normalization proofs for potential future type systems.

$$\begin{array}{c}
\frac{}{\Gamma, x : \delta \vdash x : \delta} \\
\frac{\Gamma, x : \delta \vdash M : \tau}{\Gamma \vdash \lambda x. M : \delta \rightarrow \tau} \\
\frac{\Gamma \vdash V : \delta}{\Gamma \vdash [V] : \sigma \rightarrow \delta \times \sigma} \\
\frac{\Gamma \vdash M : \sigma \rightarrow \delta' \times \sigma' \quad \Gamma \vdash V : \delta' \rightarrow \sigma' \rightarrow \delta'' \times \sigma''}{\Gamma \vdash M \star V : \sigma \rightarrow \delta'' \times \sigma''} \\
\frac{\Gamma, x : \delta \vdash M : \sigma \rightarrow \kappa}{\Gamma \vdash \text{get}_a(\lambda x. M) : (\langle a : \delta \rangle \wedge \sigma) \rightarrow \kappa} \\
\frac{\Gamma \vdash V : \delta \quad \Gamma \vdash M : (\langle a : \delta \rangle \wedge \sigma) \rightarrow \kappa}{\Gamma \vdash \text{set}_a(V, M) : \sigma \rightarrow \kappa}
\end{array}
\qquad
\begin{array}{c}
\frac{\overline{(x : [\tau] \vdash x : \tau)}_{\tau \in \iota}}{x : \iota \vdash x : \iota} \\
\frac{\Gamma, x : \iota \vdash M : \bar{\mu} \Rightarrow \bar{\nu}}{\Gamma \vdash \langle x \rangle. M : \iota \bar{\mu} \Rightarrow \bar{\nu}} \\
\frac{\Gamma \vdash V : \iota \quad \vdash \star : \iota \bar{\mu} \Rightarrow \bar{\mu} \iota}{\Gamma \vdash [V]. \star : \bar{\mu} \Rightarrow \bar{\mu} \iota} \\
\frac{\Gamma \vdash M : \bar{\mu} \Rightarrow \bar{\mu}' \iota' \quad \Delta \vdash V : \iota' \bar{\mu}' \Rightarrow \bar{\mu}'' \iota''}{\Gamma + \Delta \vdash M ; V : \bar{\mu} \Rightarrow \bar{\mu}'' \iota''} \\
\frac{\overline{(x : [\tau] \vdash x : \tau)}_{\tau \in \iota} \quad \Gamma, x : \iota' \vdash M : a(\iota) \bar{\mu} \Rightarrow \bar{\nu}}{x : \iota \vdash x : \iota \quad \Gamma, x : \iota + \iota' \vdash [x]a. M : \bar{\mu} \Rightarrow \bar{\nu}} \\
\frac{\Gamma \vdash a \langle x \rangle. [x]a. M : a(\iota + \iota') \bar{\mu} \Rightarrow \bar{\nu}}{\Gamma \vdash a \langle x \rangle. [x]a. M : a(\iota) \bar{\mu} \Rightarrow \bar{\nu}} \\
\frac{\Gamma \vdash V : \iota \quad \Delta \vdash M : a(\iota) \bar{\mu} \Rightarrow \bar{\nu}}{\Gamma + \Delta \vdash [V]a. M : \bar{\mu} \Rightarrow \bar{\nu}} \\
\frac{\Gamma + \Delta \vdash [V]a. M : \bar{\mu} \Rightarrow \bar{\nu}}{\Gamma + \Delta \vdash a \langle _ \rangle. [V]a. M : a(\square) \bar{\mu} \Rightarrow \bar{\nu}}
\end{array}$$

■ **Figure 3** A comparison of intersection typing for store to quantitative FMC typing. Let $\delta = \iota$, $\sigma = \bar{\mu}$, $\kappa = \bar{\nu}$, and $\tau = \bar{\mu} \Rightarrow \bar{\nu}$, and similarly for δ' , δ'' , etc. The last rule on the left has the side condition that a is not assigned in σ , which gives the correspondence between σ and $a(\square) \bar{\mu}$ on the right via the subtyping inequality $\omega \leq \langle a : \omega \rangle$, by $\sigma = \omega \wedge \sigma \leq \langle a : \omega \rangle \wedge \sigma = a(\square) \bar{\mu}$.

References

- 1 Sandra Alves, Delia Kesner, and Miguel Ramos. Quantitative global memory. In Helle Hvid Hansen, Andre Scedrov, and Ruy J. G. B. de Queiroz, editors, *29th International Workshop on Logic, Language, Information, and Computation (WoLLIC)*, volume 13923 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2023. doi:10.1007/978-3-031-39784-4_4.
- 2 Melissa Antonelli, Ugo Dal Lago, and Paolo Pistone. Curry and Howard meet Borel. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'22)*, pages 45:1–45:13. ACM, 2022. doi:10.1145/3531130.3533361.
- 3 Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984. doi:10.1016/c2009-0-14341-6.
- 4 Chris Barrett, Daniel Castle, and Willem Heijltjes. The Relational Machine Calculus. In Pawel Sobocinski, Ugo Dal Lago, and Javier Esparza, editors, *Proc. 39th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 9:1–9:15. ACM, 2024. doi:10.1145/3661814.3662091.
- 5 Chris Barrett, Willem Heijltjes, and Guy McCusker. The Functional Machine Calculus II: Semantics. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2023.10.
- 6 Jan A. Bergstra and Jan Willem Klop. Strong normalization and perpetual reductions in the lambda calculus. *J. Inf. Process. Cybern.*, 18(7/8):403–417, 1982.
- 7 Lasse Blaauwbroek. On the interaction between unrestricted union and intersection types and computational effects. Master’s thesis, Eindhoven University of Technology, 2017. URL: <https://research.tue.nl/en/studentTheses/d2c9d434-4f17-402e-a757-97df04ac1646>.
- 8 Viviana Bono and Mariangiola Dezani-Ciancaglini. A tale of intersection types. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 7–20. ACM, 2020. doi:10.1145/3373718.3394733.
- 9 Gérard Boudol. The lambda-calculus with multiplicities. In *International Conference on Concurrency Theory (CONCUR)*, 1993.
- 10 Flavien Breuvar and Ugo Dal Lago. On intersection types and probabilistic lambda calculi. In David Sabel and Peter Thiemann, editors, *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, pages 8:1–8:13. ACM, 2018. doi:10.1145/3236950.3236968.
- 11 Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017. doi:10.1093/JIGPAL/JZX018.
- 12 M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ -calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980. doi:10.1305/NDJFL/1093883253.
- 13 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 198–208. ACM, 2000. doi:10.1145/351240.351259.
- 14 Ugo de'Liguoro and Riccardo Treglia. Intersection types for a λ -calculus with global store. In Niccolò Veltri, Nick Benton, and Silvia Ghilezan, editors, *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, pages 5:1–5:11. ACM, 2021. doi:10.1145/3479394.3479400.
- 15 Ugo de'Liguoro and Riccardo Treglia. From semantics to types: The case of the imperative λ -calculus. *Theor. Comput. Sci.*, 973:114082, 2023. doi:10.1016/J.TCS.2023.114082.

- 16 Mariangiola Dezani-Ciancaglini, Paola Giannini, and Simona Ronchi Della Rocca. Intersection, universally quantified, and reference types. In Erich Grädel and Reinhard Kahle, editors, *18th Annual Conference of the EACSL on Computer Science Logic (CSL)*, volume 5771 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2009. doi:10.1007/978-3-642-04027-6_17.
- 17 Rémi Douence and Pascal Fradet. A systematic study of functional language implementations. *ACM Transactions on Programming Languages and Systems*, 20(2):344–387, 1998. doi:10.1145/276393.276397.
- 18 Thomas Ehrhard. Collapsing non-idempotent intersection types. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPICs*, pages 259–273. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICs.CSL.2012.259.
- 19 Thomas Ehrhard. The Scott model of linear logic is the extensional collapse of its relational model. *Theor. Comput. Sci.*, 424:20–45, 2012. doi:10.1016/J.TCS.2011.11.027.
- 20 Thomas Ehrhard. Non-idempotent intersection types in logical form. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 12077 of *LNCS*, pages 198–216, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-45231-5_11.
- 21 Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003. doi:10.1016/S0304-3975(03)00392-X.
- 22 Francesco Gavazzo, Riccardo Treglia, and Gabriele Vanoni. Monadic intersection types, relationally. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 22–51. Springer, 2024. doi:10.1007/978-3-031-57262-3_2.
- 23 Giulio Guerrieri, Willem B. Heijltjes, and Joseph W. N. Paulus. A deep quantitative type system. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic, CSL*, volume 183 of *LIPICs*, pages 24:1–24:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CSL.2021.24.
- 24 Willem Heijltjes. The Functional Machine Calculus. In *Proceedings of Mathematical Foundations of Programming Semantics (MFPS XXXVIII)*, volume 1 of *Electronic Notes in Theoretical Informatics and Computer Science*, 2022. doi:10.46298/entics.10513.
- 25 Willem Heijltjes. The Functional Machine Calculus III: Control. Submitted, 2025.
- 26 Willem Heijltjes and Georgina Majury. Simple types for probabilistic termination. In Jörg Endrullis and Sylvain Schmitz, editors, *33rd EACSL Annual Conference on Computer Science Logic (CSL)*, Leibniz International Proceedings in Informatics, pages 31:1–31:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 2025. doi:10.4230/LIPICs.CSL.2025.31.
- 27 A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 161–174. ACM, 1999. doi:10.1145/292540.292556.
- 28 Assaf J. Kfoury. A linearization of the lambda-calculus and consequences. *Journal of Logic and Computation*, 10(3):411–436, 2000. doi:10.1093/LOGCOM/10.3.411.
- 29 Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, Rijksuniversiteit Utrecht, 1980.
- 30 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20:199–207, 2007. doi:10.1007/s10990-007-9018-9.

- 31 Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. Weighted relational models of typed lambda-calculi. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 301–310, 2013. doi:10.1109/LICS.2013.36.
- 32 Paul Blain Levy. *Call-by-push-value: A functional/imperative synthesis*, volume 2 of *Semantic Structures in Computation*. Springer Netherlands, 2003. doi:10.1007/978-94-007-0954-6.
- 33 Damiano Mazza, Luc Pellissier, and Pierre Vial. Polyadic approximations, fibrations and intersection types. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158094.
- 34 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.
- 35 Rob Nederpelt. *Strong normalization in a typed lambda calculus with lambda structured types*. PhD thesis, Technische hogeschool Eindhoven, 1973.
- 36 Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: Why nonlinearity and amnesia make type systems work. In *Proc. 9th International Conference on Functional Programming (ICFP)*, pages 138–149. ACM, 2004. doi:10.1145/1016850.1016871.
- 37 C.-H. Luke Ong. Quantitative semantics of the lambda calculus: Some generalisations of the relational model. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2017. doi:10.1109/LICS.2017.8005064.
- 38 Luca Paolini, Mauro Piccolo, and Simona Ronchi Della Rocca. Essential and relational models. *Math. Struct. Comput. Sci.*, 27(5):626–650, 2017. doi:10.1017/S0960129515000316.
- 39 Elaine Pimentel, Simona Ronchi Della Rocca, and Luca Roversi. Intersection types from a proof-theoretic perspective. *Fundamenta Informaticae*, 121(1-4):253–274, 2012. doi:10.3233/FI-2012-778.
- 40 Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.
- 41 G. Pottinger. A type assignment for the strongly normalizable λ -terms. In *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577. Academic Press, London, 1980.
- 42 A.J. Power and Hayo Thielecke. Closed Freyd- and κ -categories. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1644 of *Lecture Notes in Computer Science*, pages 625–634. Springer, 1999. doi:10.1007/3-540-48523-6_59.
- 43 Femke van Raamsdonk, Paula Severi, Morten Heine Sørensen, and Hongwei Xi. Perpetual reductions in lambda-calculus. *Inf. Comput.*, 149(2):173–225, 1999. doi:10.1006/inco.1998.2750.

A Addendum to Section 7: encoding *Quantitative Global Memory*

This Appendix will compare the present quantitative types against the multi-types of Alves, Kesner, and Ramos in *Quantitative Global Memory* [1]. The paper uses the same calculus as De’Liguoro and Treglia [14], characterising it as *weak open call-by-value*, i.e. not reducing under abstractions while allowing open terms. The calculus and its FMC encoding are as follows, using the standard interpretation of call-by-value λ -calculus [17, 24, 42], where the standard encoding of call-by-value application, $v\ t = t; [v]; \langle x \rangle. x$, is reduced to $t; v$.

Values v :	Terms t :	
$x = x$	$v = [v].\star$	$\text{get}_a(\lambda x.t) = a\langle x \rangle.[x]a.t$
$\lambda x.t = \langle x \rangle.t$	$v\ t = t; v$	$\text{set}_a(v, t) = a\langle _ \rangle.[v]a.t$

The paper gives a system of quantitative types for this calculus. Here, we will consider it in simplified form, omitting base types and restricting our focus to four (out of eight) categories. The type system, and its encoding in quantitative FMC types, is then as follows.

$$\begin{array}{c}
\frac{}{x : [\sigma] \vdash x : \sigma} \qquad \frac{}{x : [\sigma] \vdash x : \sigma} \\
\\
\frac{\Gamma, x : \mathcal{M} \vdash t : \mathcal{S} \gg \kappa}{\Gamma \vdash \lambda x. t : \mathcal{M} \Rightarrow (\mathcal{S} \gg \kappa)} \qquad \frac{\Gamma, x : \mathcal{M} \vdash t : \mathcal{S} \Rightarrow \kappa}{\Gamma \vdash \langle x \rangle. t : \mathcal{M} \mathcal{S} \Rightarrow \kappa} \\
\\
\frac{(\Gamma_i \vdash v : \sigma_i)_{i \leq n}}{\Gamma_1 + \dots + \Gamma_n \vdash v : [\sigma_1, \dots, \sigma_n]} \qquad \frac{(\Gamma_i \vdash v : \sigma_i)_{i \leq n}}{\Gamma_1 + \dots + \Gamma_n \vdash v : [\sigma_1, \dots, \sigma_n]} \\
\\
\frac{\Gamma \vdash v : \mathcal{M}}{\Gamma \vdash v : \mathcal{S} \gg (\mathcal{M} \times \mathcal{S})} \qquad \frac{\Gamma \vdash v : \mathcal{M} \quad \vdash \star : \mathcal{M} \mathcal{S} \Rightarrow \mathcal{M} \mathcal{S}}{\Gamma \vdash [v]. \star : \mathcal{S} \Rightarrow \mathcal{M} \mathcal{S}} \\
\\
\frac{\Gamma \vdash v : \mathcal{M} \Rightarrow (\mathcal{S}' \gg \kappa) \quad \Delta \vdash t : \mathcal{S} \gg (\mathcal{M} \times \mathcal{S}')}{\Gamma + \Delta \vdash v t : \mathcal{S} \gg \kappa} \qquad \frac{\Delta \vdash t : \mathcal{S} \Rightarrow \mathcal{M} \mathcal{S}' \quad \Gamma \vdash v : \mathcal{M} \mathcal{S}' \Rightarrow \kappa}{\Gamma + \Delta \vdash t; v : \mathcal{S} \Rightarrow \kappa} \\
\\
\frac{\Gamma, x : \mathcal{M} \vdash t : \{a : \mathcal{N}\}; \mathcal{S} \gg \kappa}{\Gamma \vdash \text{get}_a(\lambda x. t) : \{a : \mathcal{M} + \mathcal{N}\}; \mathcal{S} \gg \kappa} \qquad \frac{\frac{(x : [\sigma] \vdash x : \sigma)_{\sigma \in \mathcal{N}}}{x : \mathcal{N} \vdash x : \mathcal{N}} \quad \Gamma, x : \mathcal{M} \vdash t : a(\mathcal{N}) \mathcal{S} \Rightarrow \kappa}{\Gamma, x : \mathcal{M} + \mathcal{N} \vdash [x] a. t : \mathcal{S} \Rightarrow \kappa}}{\Gamma \vdash a \langle x \rangle. [x] a. t : a(\mathcal{M} + \mathcal{N}) \mathcal{S} \Rightarrow \kappa} \\
\\
\frac{\Gamma \vdash v : \mathcal{M} \quad \Delta \vdash t : \{a : \mathcal{M}\}; \mathcal{S} \gg \kappa}{\Gamma + \Delta \vdash \text{set}_a(v, t) : \{a : []\}; \mathcal{S} \gg \kappa} \qquad \frac{\Gamma \vdash v : \mathcal{M} \quad \Delta \vdash t : a(\mathcal{M}) \mathcal{S} \Rightarrow \kappa}{\Gamma + \Delta \vdash [v] a. t : \mathcal{S} \Rightarrow \kappa}}{\Gamma + \Delta \vdash a \langle _ \rangle. [v] a. t : a([]) \mathcal{S} \Rightarrow \kappa}
\end{array}$$

■ **Figure 4** A comparison of multi-types for store to quantitative FMC typing. The FMC types omit the default location λ , using $\mathcal{M}\mathcal{S}$ for $\lambda(\mathcal{M})SSS$. In the last rule on the left, the addition of $\{a : []\}$ (in green) to the conclusion type corrects a minor mistake in [1] (confirmed in private communication with the authors).

Value types:	$\sigma ::= \mathcal{M} \Rightarrow (\mathcal{S} \gg \kappa)$	$= \lambda(\mathcal{M}) \mathcal{S} \Rightarrow \kappa$
Configuration types:	$\kappa ::= \mathcal{M} \times \mathcal{S}$	$= \lambda(\mathcal{M}) \mathcal{S}$
Multi-types:	$\mathcal{M} ::= [\sigma_1, \dots, \sigma_n]$	$= [\sigma_1, \dots, \sigma_n]$
State types:	$\mathcal{S} ::= \{a_1 : \mathcal{M}_1, \dots, a_n : \mathcal{M}_n\}$	$= a_1(\mathcal{M}_1) \dots a_n(\mathcal{M}_n)$

That is: *value types* encode as computation types τ , *state types* as memory types \bar{l} excluding the main location λ , *multi-types* as collection types ι , and *configuration types* as memory types \bar{l} .

The type system features the properties that it is *tight*, it counts exactly the reduction steps to normal form, and it is *split*, it separately counts beta-steps, store operations, and the size of the normal form. These quantities are however not meaningful in the FMC encoding, which measures machine evaluation rather than reduction, identifies beta-reduction and store operations (both are stack interactions), and considers spine reduction, while the encoded call-by-value reduction corresponds to weak head reduction in the FMC. Accordingly, we will here consider only the structural correspondence between both type systems, omitting the triple counters and the base types a , v , and n which are used to identify normal forms. Thus simplified, the type system and its FMC encoding are given in Figure 4.