The Relational Machine Calculus

Chris Barrett
chris.barrett@cs.ox.ac.uk
University of Oxford
United Kingdom

Daniel Castle
drc22@bath.ac.uk
University of Bath
United Kingdom

Willem Heijltjes
w.b.heijltjes@bath.ac.uk
University of Bath
United Kingdom

ABSTRACT

This paper presents the Relational Machine Calculus (RMC): a simple, foundational model of first-order relational programming. The RMC originates from the Functional Machine Calculus (FMC), which generalizes the lambda-calculus and its standard call-by-name stack machine in two directions. One, "locations", introduces multiple stacks, which enable effect operators to be encoded into the abstraction and application constructs. The second, "sequencing", introduces the imperative notions of "skip" and "sequence", similar to kappa-calculus and concatenative programming languages.

The key observation of the RMC is that the first-order fragment of the FMC exhibits a latent duality which, given a simple decomposition of the relevant constructors, can be concretely expressed as an involution on syntax. Semantically, this gives rise to a sound and complete calculus for string diagrams of Frobenius monoids.

We consider unification as the corresponding symmetric generalization of beta-reduction. By further including standard operators of Kleene algebra, the RMC embeds a range of computational models: the kappa-calculus, logic programming, automata, Interaction Nets, and Petri Nets, among others. These embeddings preserve operational semantics, which for the RMC is again given by a generalization of the standard stack machine for the lambda-calculus. The equational theory of the RMC (which supports reasoning about its operational semantics) is conservative over both the first-order lambda-calculus and Kleene algebra, and can be oriented to give a confluent reduction relation.

CCS CONCEPTS

• **Theory of computation → Abstract machines**; **Lambda calculus**; **Regular languages**; **Operational semantics**; **Denotational semantics**; **Categorical semantics**; **Constraint and logic programming**.

KEYWORDS

lambda-calculus, Kleene algebra, logic programming, reversible programming, non-determinism, hypergraph category, Krivine abstract machine, categorical semantics, operational semantics.

ACM Reference Format:
Chris Barrett, Daniel Castle, and Willem Heijltjes. 2024. The Relational Machine Calculus. In *39th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '24), July 8–11, 2024, Tallinn, Estonia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3661814.3662091

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LICS '24, July 8–11, 2024, Tallinn, Estonia
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0660-8/24/07
https://doi.org/10.1145/3661814.3662091

1 INTRODUCTION

The λ-calculus is widely considered the canonical model of functional programming. However, there is no similarly agreed foundation of *relational* programming. Paradigmatic examples of relational programming are logic programming, based on unification [1, 39, 43, 58, 72], and database query languages, based on Tarski's relational algebra [22, 75]. Relations also serve as the (intended) semantics for a wide range of languages. For example, non-deterministic finite automata, Kleene algebra and its variants — some of which subsume propositional Hoare logic [25, 50] — and monadic languages for non-determinism [60]. Furthermore, the category of relations is a simple folk model of (differential) linear logic and thus of the linear λ-calculus; via the Kleisli construction, it is also the basis of a simple quantitative model of the plain λ-calculus [29, 37, 53], which can even be seen as underlying game semantics and intersection types [16, 26, 35, 63].

Here we present a foundational model of (first-order, sequential) relational programming: the *Relational Machine Calculus (RMC)*. We first set out the aims and constraints that informed its design. Then, in the remainder of the introduction, we detail the origin of the RMC: from a first-order λ-calculus through several adaptations to incorporate the relational paradigm, including *duality*, *unification*, and *non-determinism*. The body of the paper is dedicated to justifying our claim of meeting the design criteria we now set out.

1.1 Design criteria

The space of all possible (first-order, sequential) relational models of computation is bewildering, and indeed there is a proliferation of languages in this space, indicating its importance and range of applications. Yet, we find none in the literature which satisfy the following — minimal, but stringent — design criteria that we would expect to be met by a "λ-calculus of relational computation".

Denotational semantics: has a relational and categorical semantics, through quotienting by a local equational theory;
Duality: exhibits a syntactic involution which switches input and output, and denotes the relational converse;
Operational semantics: preserves that of standard models of first-order, sequential (standard and relational) computation;
Confluence: orienting the equational theory gives a confluent reduction relation, sound for the operational semantics.

We proceed to motivate these criteria, which serve to tame the design space and establish standards for success in our programme.

Denotational semantics. A minimal qualifying criterion for any relational calculus is to have a denotational semantics in the category of sets and relations. We further expect a categorical semantics in an appropriate symmetric monoidal category which abstracts the structural properties of relations; we shall see momentarily our preferred candidate.

Duality. While the category of sets and relations embeds that of sets and functions, it exhibits very different structural properties; mainly, a perfect duality between input and output given by relational converse. We consider this duality a defining feature of relational computation. A paradigmatic example is the following Prolog program to concatenate two lists.

```
concat([],L,L)
concat([E | L1], L2, [E | L3]) :- concat(L1,L2,L3)
```

Fixing input lists L1 and L2 in the relation concat(L1,L2,L3) returns their concatenation as L3, but by fixing the value L3 instead, the relation may be run in reverse to (non-deterministically) return every way of splitting L3 into L1 and L2.

Duality is a common theme in programming language research [31, 36, 70], and in mathematics and physics more generally. Its aesthetic appeal barely needs justification; practically, it offers parsimony of expression by identifying a dual program (theorem) with each program (theorem) written. We require our language to feature duality in a direct and natural way: by a *syntactic* involution.

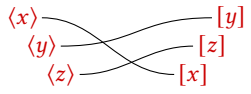
Operational semantics. We expect our language to be expressive enough to encode a range of relational models of first-order, sequential computation, including the first-order λ-calculus, automata, logic programming, and Petri nets. However, any sufficiently expressive language can encode any other: we would like the embeddings into our calculus to be, in some way, *natural*. As our criterion for this, we specify that operational semantics is preserved, for models where it is defined. This means that (at least, with respect to *operational* semantics — a relatively fine-grained notion) the encoded models may be viewed as *fragments* of our language, built from its more fundamental primitives.

Confluence. The intended relational semantics will impose an equational theory on our calculus. Following the central notion of β-reduction in the λ-calculus, we expect the equational theory (modulo certain congruences) to be orientable as a reduction relation, and that this relation is *confluent*, contributing in an essential way to an effective solution to this theory.

1.2 First-order lambda-calculus and duality

Our point of departure is the first-order λ-calculus, as first appeared as the κ-calculus [40, 67] and recently as the first-order fragment of the Functional Machine Calculus [4, 42] (FMC). These calculi feature three of our desired properties: a *denotational semantics* (albeit in Cartesian categories), an *operational semantics* in a simple stack machine, and *confluent reduction*. Crucially, they also exhibit the potential for *duality*; and in the formulation of the FMC, this is directly observable in the syntax.

Taking the operational perspective, the first-order FMC is an instruction language for a simple stack machine. Terms are sequences of *push* $[x]$ and *pop* $\langle x\rangle$ operations over variables x or constant values c, separated by sequential composition (;). The following example shows a term and its associated string diagram; the type is given by the size of its input and output stacks.



$$\langle x\rangle\,;\langle y\rangle\,;\langle z\rangle\,;[x]\,;[z]\,;[y]\;:3\Rightarrow 3$$

Operationally, a term evaluates from left to right; $\langle x\rangle$ pops the head off the stack, say a constant c, and substitutes c for x in the remaining computation; $[x]$ pushes the value substituted for x onto the stack. We write our stacks with the head to the right, to match the order of *pushes*; our term then takes a stack $e\,d\,c$ to $c\,e\,d$, where $\langle x\rangle$ pops c, $\langle y\rangle$ pops d, and $\langle z\rangle$ pops e.

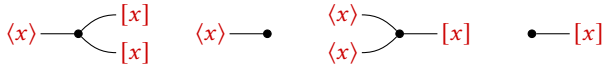
In this way, our term implements a relation between input and output stacks, as illustrated by the string diagram, where the wires represent the manipulation of the items on the stack (shown with the head at the top). The simple typing discipline of giving input and output arity tames the asymmetric nature of stacks, resulting in an internal language for symmetric monoidal categories (SMCs) in the linear case (where each *pop* $\langle x\rangle$ is matched by a unique *push* $[x]$ to its right, and vice versa).

In the FMC, a *pop* $\langle x\rangle\,;M$, with M the remaining computation, is a first-order lambda-abstraction, which we write $\kappa x.M$ following the κ-calculus. A *push* $[x]\,;M$ corresponds to an application $M\,x$, restricted to first order by forcing the argument to be a variable. The stack machine is then a simplified Krivine machine [51], replacing environments with substitution. This highlights a key aspect. A κ-abstraction is a *binder*, making the variable x *local* to $\kappa x.M$. However, *binding* and *variable scope* are the main notions that will need to be re-visited in light of duality. A pop $\langle x\rangle$ is therefore not a binder, and variables in this fragment of the RMC are *global*. This does not prevent us from encoding first-order λ-calculus: in the first-order setting, terms are never duplicated, removing any issues with variable capture or α-conversion. We may then simulate binding by stipulating that each *pop* $\langle x\rangle$ has a unique variable, which only occurs in *pushes* to its right (this is *Barendregt's convention*). Later, when duplication of terms returns, we will re-introduce local variable scope with a binding *new variable* construct $\exists x.\,M$, justifying the core fragment of the RMC as a decomposition of abstraction into its two distinct roles: *pop* and *new variable*.

Duality. Our formulation of first-order λ-calculus is tailored to reveal its latent duality, effected simply by reversing a term while switching *push* and *pop*. The dual of our previous example term:

$$\langle y\rangle\,;\langle z\rangle\,;\langle x\rangle\,;[z]\,;[y]\,;[x]\;:3\Rightarrow 3\,.$$

The linear fragment, where terms represent *permutations*, is closed under duality. By contrast, the *non-linear* first-order λ-calculus is characterised by *duplication* and *deletion*, embodied by the *diagonal* and *terminal* below left. These yield the dual terms below right, which we label *matching* and *arbitrary*, that feature multiple related *pops*, or zero.



$\langle x\rangle\,;[x]\,;[x]$ $\qquad$ $\langle x\rangle$ $\qquad$ $\langle x\rangle\,;\langle x\rangle\,;[x]$ $\qquad$ $[x]$

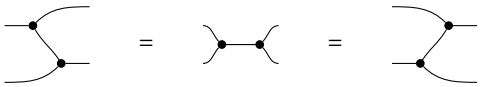
Semantics tells us that the new terms should represent the relational converse of the diagonal and terminal functions, given respectively by the partial function that sends (x,x) to x and is otherwise undefined, and the non-deterministic function sending the trivial input to all possible outputs.

Operationally, the required behaviour of *matching* $\langle x\rangle\,;\langle x\rangle\,;[x]$ for an input stack of two identical values, $c\,c$, is to return c, and for distinct values, $d\,c$, is to fail. We achieve this by generalising

pop to hold also *constants*, $\langle c \rangle$, interpreted as an *assertion* that the head of the stack is c, with failure for any other constant. Then $\langle x \rangle ; \langle x \rangle ; [x]$ evaluates by first popping c, substituting it for x in both *pop* and *push* to get $\langle c \rangle ; [c]$, which then pops and replaces the second value c (or fails for d).

The second term, *arbitrary* $[x]$, illustrates that stack values must include variables, and hence must be substituted for: the term $\langle c \rangle$ for a stack $x\, x$ incurs a substitution of c for the second x. The behaviour of *push* and *pop* thus becomes perfectly symmetric, with substitutions in both the stack and the remaining term.

Altogether these relations are abstracted categorically as *Frobenius monoids*, which we take to capture non-linear computation in the presence of duality, given in string diagrams as above and satisfied by the *Frobenius equation* below.



Intuitively, this implements the idea that *connectivity = identity*: connected wires represent a single value. In our calculus, this is modeled by *variable names*; and indeed the three diagrams in the above equation may all be represented by the term $\langle x \rangle ; \langle x \rangle ; [x] ; [x]$.

Frobenius monoids have received a huge amount of interest over the last decade or so, since they were identified as a fundamental primitive of quantum computation by the ZX-calculus [23, 24, 47]. Since then, they have been adopted as primitives in a wide range of string diagrammatic languages, including those capturing conjunctive queries, relational algebra, and aspects of logic programming [6, 10, 13, 38]. In programming language theory specifically, they have received significantly less attention, but have been used in a synthetic axiomatization of the operation of *exact conditioning* in probabilistic programming [27, 73] and to model the (partial) inverse of duplication in reversible languages [48]. The RMC offers a novel *operational* account of Frobenius monoids, thus filling a gap in the literature and bridging the study of string-diagrammatic and programming languages.

1.3 The full calculus

The calculus thus far is generated by: *push* $[x]$ and *pop* $\langle x \rangle$ operations, *sequential composition* $M ; N$, and its unit $\star$, the imperative *skip*. Associativity of composition will be implemented via the machine and the equational theory. To reach our desired expressivity, we extend this with a careful selection of features, outlined below.

Algebraic terms and unification. The β-reduction relation is the interaction of a consecutive *push* and *pop*, say $[y] ; \langle x \rangle$, to incur a (global) substitution $\{y/x\}$. With constants, both $[c] ; \langle x \rangle$ and $[x] ; \langle c \rangle$ incur $\{c/x\}$, while $[c] ; \langle c \rangle$ succeeds ($\star$) and $[d] ; \langle c \rangle$ fails.

Viewing a redex as a formal equation, these are the familiar rules of *first-order unification*, restricted to variables and constants. We generalize our calculus to allow *algebraic terms* [1] as values, and will implement unification on the machine and through a symmetric version of β-reduction. Evaluating a redex $[t] ; \langle s \rangle$ will thus produce substitutions constituting the most general unifier of t and s, or fail if none exists.

[1] We write "algebraic terms" to avoid name clashing with "first-order terms", which here refer to terms of the RMC, in contrast with "higher-order terms" of the λ-calculus.

Kleene Algebra. To internalize the partial and non-deterministic semantics of relations, we introduce non-deterministic *sum* ($+$) and *failure* (0), its unit. Indeed, the presence of non-determinism is often considered a defining feature of relational programming. With sequencing already present, we also desire a Kleene star construction ($-^*$) to model infinitary behaviour, *e.g.*, of logic programming. Our full calculus will thus be conservative over Kleene algebra (KA).

The atoms of KA may be interpreted as programs, with the axioms of KA acting as a set of (weak) fundamental laws any standard non-deterministic, sequential language should satisfy [45, 46, 50]. However, an *operational* interpretation of KA is typically only given via its correspondence with finite state automata. Our calculus can be viewed as a modification of KA to include the dual primitives of *push* and *pop*, replacing the atoms of KA and supporting its extension with a *direct* operational semantics.

Local variables. Evaluation for Kleene star, taking a term M^* non-deterministically to a sequence $M ; \ldots ; M$ of any length, reintroduces duplication of terms into the calculus, bringing with it the familiar problems of variable identity and α-conversion. We adopt a standard solution: making the scope of a variable explicit with a *new variable* construct, $\exists x.\, M$, familiar variously from nominal Kleene algebra [34], functional logic programming [39], π-calculus [59], and the ν-calculus [66] as a fragment of ML. This restores locality and gives a proper decomposition of κ-abstraction into its two roles: first, as the binder of new variables, and second, as an instruction to *pop* from the stack.

$$\kappa x.\, M \;=\; \exists x.\, (\langle x \rangle ; M)$$

Locations. We inherit a further feature from the Functional Machine Calculus: multiple stacks (or streams) on the abstract machine, indexed in a set of *locations*. This captures several computational effects: mutable store, as stacks restricted to depth one; input and output, as dedicated streams; and a probabilistic generator, as a stream of random bits — remarkably, *while retaining confluent reduction* [42]. The required generalization of the syntax is a simple parameterization of *push* and *pop* in a location a, as $[x]a$ and $a\langle x \rangle$ respectively, to operate on the indicated stack. Thereby, the RMC subsumes the operational semantics of the given effects, while $\beta\eta$-equivalence captures their algebraic theory [4, 42]. This extension allows the simple modelling of stateful models of computation such as Turing machines and Petri nets, demonstrated in Section 8.

2 THE RELATIONAL MACHINE CALCULUS

We define the *Relational Machine Calculus* (RMC) as follows. We assume a countable set of *locations* $A = \{a, b, c \ldots\}$, which each represent a stack on the abstract machine. Stacks hold algebraic terms over a signature Σ of function symbols f^n of arity n.

Definition 2.1 (Relational Machine Calculus). *Values* s, t and *(computation) terms* M, N are given by the grammars below.

s, t	::=	$x \mid f^n(t_1, \ldots, t_n) \quad (f^n \in \Sigma)$		algebraic terms
M, N	::=	$\star \mid M ; N \mid M^* \mid 0 \mid M + N$		Kleene algebra
	$\mid$	$[t]a \mid a\langle t \rangle$		stack operations
	$\mid$	$\exists x.\, M$		variable scope

From left to right, the computation terms are *skip* or *nil* $\star$, *(sequential) composition* $M\,;N$, *Kleene star* M^*, *zero* or *failure* 0, a *sum* of terms $M+N$, a *push* $[t]a$ of the value t to the location a, a *pop* $a\langle t\rangle$ from the location a to unify with t, and a *new* variable introduction $\exists x.\,M$ which binds x in M. Operator precedence: Kleene star binds tightest, then sequencing, then *new* $\exists x.\,M$, and finally sum; then $\exists x.\,M\,;N+P\,;Q^* = (\exists x.\,(M\,;N))+(P\,;(Q^*))$. We often omit the superscript on f^n.

The calculus exhibits *duality* as a syntactic involution.

Definition 2.2 (Duality). Define *duality* $(-)^\dagger$ on terms as follows.

$$\begin{array}{rclcrcl} \star^\dagger & = & \star & & (M+N)^\dagger & = & M^\dagger+N^\dagger \\ (N\,;M)^\dagger & = & M^\dagger\,;N^\dagger & & [t]a^\dagger & = & a\langle t\rangle \\ (M^*)^\dagger & = & (M^\dagger)^* & & a\langle t\rangle^\dagger & = & [t]a \\ 0^\dagger & = & 0 & & (\exists x.\,M)^\dagger & = & \exists x.\,M^\dagger \end{array}$$

We shall see throughout the paper how various operational, equational, and semantic notions either dualize or respect duality.

2.1 Operational Semantics

The small-step operational semantics is given by a stack machine, the *relational abstract machine* or *relational machine*.

Definition 2.3 (Relational Machine). A *state* is a triple (S_A, M, K) of: a *memory* S_A, a family of stacks indexed in a set of locations A; a term M; and a *continuation stack* K. These are defined as follows.

$$\begin{array}{rcll} K,L & ::= & \varepsilon \mid M\,K & \text{continuation stacks} \\ S,T & ::= & \varepsilon \mid S\,t & \text{operand stacks} \\ S_A & ::= & \{S_a \mid a\in A\} & \text{memories} \end{array}$$

The addition of a stack S_a to a memory S_A ($a\notin A$) is written $S_A\cdot S_a$. We abbreviate $f(t_1,\dots,t_n)$ to $f(T)$ where $T=t_1\dots t_n$, and $[t_1]a\,;\dots\,;[t_n]a$ to $[T]a$ and $a\langle t_n\rangle\,;\dots\,;a\langle t_1\rangle$ to $a\langle T\rangle$ (note the inversion). The *transitions* of the machine are given in Figure 1, read top–to–bottom, and are non-deterministic: a state transitions to a formal sum of states, represented by branching of the transitions. A *run* of the machine is a single rooted path in the machine tree, not necessarily to a leaf, shown with a double line as below.

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

A run is *successful* if it terminates in a state $(\,T_A\,,\,\star\,,\,\varepsilon\,)$. A state where no transition rules apply represents *failure*, and is considered to have *zero* branches.

Machine evaluation of a term M for an input memory S_A gives a (possibly infinite) number of successful runs, each with a return memory T_A. The big-step evaluation function $(S_A \Downarrow M)$ will collect these as a multiset. Since variables are global and machine steps substitute into the continuation stack, evaluation returns also a finite substitution map σ with each return memory T_A, as a pair (T_A,σ). In a composition $M\,;N$, the subsitutions from M can then be applied to N.

Notation. We denote the empty multiset by $[]$, a singleton by $[(T_A,\sigma)]$, and multiset union by $\sqcup$. Substitution maps σ and τ are applied to a term as σM and composed as $(\sigma\tau)M=\sigma(\tau M)$. The empty map is ε, and $\sigma\setminus y$ is as σ except undefined on y. The n-fold composition of a term M is M^n where $M^0=\star$ and $M^{n+1}=M\,;M^n$.

$$\frac{(\,S_A\,,\,\star\,,\,M\,K\,)}{(\,S_A\,,\,M\,,\,K\,)} \qquad \frac{(\,S_A\,,\,M^*\,,\,K\,)}{(\,S_A\,,\,\star\,,\,K\,)\quad(\,S_A\,,\,M\,;M^*\,,\,K\,)}$$

$$\frac{(\,S_A\,,\,M\,;N\,,\,K\,)}{(\,S_A\,,\,M\,,\,N\,K\,)} \qquad \frac{(\,S_A\,,\,M+N\,,\,K\,)}{(\,S_A\,,\,M\,,\,K\,)\quad(\,S_A\,,\,N\,,\,K\,)}$$

$$\frac{(\,S_A\cdot S_a\,,\,[t]a\,,\,K\,)}{(\,S_A\cdot S_a\,t\,,\,\star\,,\,K\,)} \qquad \frac{(\,S_A\,,\,\exists x.\,M\,,\,K\,)}{(\,S_A\,,\,\{y/x\}M\,,\,K\,)}\ (y\text{ fresh})$$

$$\frac{(\,S_A\cdot S_a\,x\,,\,a\langle x\rangle\,,\,K\,)}{(\,S_A\cdot S_a\,,\,\star\,,\,K\,)} \qquad \frac{(\,S_A\cdot S_a\,f(R)\,,\,a\langle f(T)\rangle\,,\,K\,)}{(\,S_A\cdot S_a\,R\,,\,a\langle T\rangle\,,\,K\,)}$$

$$\frac{(\,S_A\cdot S_a\,x\,,\,a\langle t\rangle\,,\,K\,)}{(\,\{t/x\}(S_A\cdot S_a)\,,\,\star\,,\,\{t/x\}K\,)}\ (x\notin t)$$

$$\frac{(\,S_A\cdot S_a\,t\,,\,a\langle x\rangle\,,\,K\,)}{(\,\{t/x\}(S_A\cdot S_a)\,,\,\star\,,\,\{t/x\}K\,)}\ (x\notin t)$$

Figure 1: Transitions of the Relational Machine

Definition 2.4 (Big-step operational semantics). The big-step operational semantics of the RMC is given by the evaluation function $(-\Downarrow -)$ below, where y is globally *fresh* in the $\exists x.\,M$ case.

$$\begin{array}{rcll} S_A\Downarrow\star & = & [(S_A,\varepsilon)] & \\ S_A\Downarrow M\,;N & = & [\,(U_A,\tau\sigma)\mid (T_A,\sigma)\in S_A\Downarrow M, & \\ & & \quad (U_A,\tau)\in T_A\Downarrow\sigma N\,] & \\ S_A\Downarrow M^* & = & \bigsqcup_{n\in\mathbb{N}}(S_A\Downarrow M^n) & \\ S_A\Downarrow M+N & = & (S_A\Downarrow M)\sqcup(S_A\Downarrow N) & \\ S_A\cdot S_a\Downarrow[t]a & = & [(S_A\cdot S_a\,t,\varepsilon)] & \\ S_A\cdot S_a\,x\Downarrow a\langle x\rangle & = & [(S_A\cdot S_a,\varepsilon)] & \\ S_A\cdot S_a\,x\Downarrow a\langle t\rangle & = & [(\{t/x\}(S_A\cdot S_a),\{t/x\})] & (x\notin t) \\ S_A\cdot S_a\,t\Downarrow a\langle x\rangle & = & [(\{t/x\}(S_A\cdot S_a),\{t/x\})] & (x\notin t) \\ S_A\cdot S_a\,f(R)\Downarrow a\langle f(T)\rangle & = & S_A\cdot S_a\,R\Downarrow a\langle T\rangle & \\ S_A\Downarrow\exists x.\,M & = & [\,(T_A,\sigma\setminus y)\mid (T_A,\sigma)\in S_A\Downarrow\{y/x\}M\,] & \\ S_A\Downarrow M & = & [] & \text{(otherwise)} \end{array}$$

The function $\lfloor -\rfloor$ takes a multiset to its underlying set, and the function $\mathrm{ret}(-)$ projects onto only *return memories* in $\lfloor S_A\Downarrow M\rfloor$.

We assume that fresh variables are generated globally, without clashing. In particular, in $(\exists x.\,M)\,;N$ the variable instantiating the $\exists x$ must be free for N as well as M. One may address this practically by carrying along the *formal parameters* of $(\exists x.\,M)\,;N$ into $\exists x.\,M$; we omit this for brevity.

Proposition 2.5 (Big-step semantics is well defined). *The evaluation function $(-\Downarrow -)$ is a total function.*

Proposition 2.6 (Small-step and big-step semantics agree). *For every memory S_A, term M and continuation K, there is a bijection between the elements (T_A,σ) of $(S_A\Downarrow M)$ and successful runs*

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

3 ENCODING COMPUTATIONAL MODELS, I

We embed a number of computational models (or algorithms, in the case of unification) into the RMC to illustrate the origins and purposes of its various constructs. Observe, in particular, that operational semantics is preserved, often in a strong sense. We use only one, unnamed stack, writing $[t]$ and $\langle t \rangle$. Encodings of stateful languages, making use of multiple locations, are given in Section 8.

Notation. We introduce vector notation for stacks of variables $\vec{x} = x_1 \dots x_n$, and emphasize the reversal in $\langle \overleftarrow{x} \rangle = \langle x_n \rangle ; \dots ; \langle x_1 \rangle$ by pointing the arrow left. Concatenation is given by juxtaposition. We further abbreviate $\exists x_1 \dots \exists x_n . M$ as $\exists \vec{x} . M$.

3.1 Regular expressions

Regular expressions (REs) over an alphabet Σ of constants $a, b, c, \dots$ are given by the following grammar, with their embedding $[\![-]\!]$ into RMC-terms below it, given by juxtaposition.

$$E, E' ::= \varepsilon \mid E\,E' \mid E^* \mid \varnothing \mid (E|E') \mid c \in \Sigma$$
$$M, N ::= \star \mid M ; N \mid M^* \mid 0 \mid M + N \mid [c]$$

Here, constant values c are *pushed* to the stack. The regular language $\mathcal{L}_E$ defined by E — a set of words (stacks) over Σ — is thus given directly by the evaluation of $[\![E]\!]$.

Proposition 3.1 (Regular expressions embed). *For an RE E,*

$$\mathcal{L}_E = \mathrm{ret}\lfloor \varepsilon \Downarrow [\![E]\!] \rfloor .$$

There is a *dual* embedding $[\![-]\!]^\dagger$ of REs into the grammar below, given by composition of the original embedding with duality.

$$M, N ::= \star \mid M ; N \mid M^* \mid 0 \mid M + N \mid \langle c \rangle$$

Here, constant values c are *popped* from the stack. The embedding can be considered as defining a computation which *tests* words (stacks), with a word S accepted if $\varepsilon \in \mathrm{ret}\lfloor S \Downarrow [\![E]\!]^\dagger \rfloor$. The characteristic (or indicator) function of a regular language $\mathcal{L}_E$ is thus given directly by evaluation of $[\![E]\!]^\dagger$.

3.2 Unification

A (first-order) unification algorithm [56] takes a set of formal equations $E = \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}$ and returns a most general unifier (MGU), a minimal subsitution σ such that $\sigma s_i = \sigma t_i$ for all i, if one exists. In the RMC, we may encode such equations as redexes: $[\![s \doteq t]\!] = [s] ; \langle t \rangle$. Unification (as a general concept) can then be viewed as embedding in the RMC as the following fragment.

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

Terms are sequences of pushes and pops of first-order values to and from this location. A set of equations E is encoded as follows.

$$[\![\{s_1 \doteq t_1, \dots, s_n \doteq t_n\}]\!] = [s_n] ; \dots ; [s_1] ; \langle t_1 \rangle ; \dots ; \langle t_n \rangle$$

Note that we have chosen nested redexes, since that is also what the machine reduction for constants (for $[f(s_1, \dots, s_n)] ; \langle f(t_1, \dots, t_n) \rangle$) produces; but a sequence of redexes would have been equally valid. We then observe that evaluation returns an MGU if one exists.

Proposition 3.2 (First-order unification embeds). *For a set of formal equations over algebraic terms E we have the following, where σ is an MGU for E, if one exists, and otherwise $\varepsilon \Downarrow [\![E]\!] = [\,]$.*

$$\varepsilon \Downarrow [\![E]\!] = [(\varepsilon, \sigma)]$$

For machine evaluation, which does not return a substitution explicitly, we may obtain σ by populating the initial stack with the free variables $x_1 \dots x_n$ in E. This is the domain of σ; the machine then substitutes into each variable x_i to return σx_i, as follows, so that the relation from input to output stack captures the MGU σ.

$$\frac{(\ x_1 \ \dots \ x_n \quad , [\![E]\!] , \varepsilon\ )}{(\ \sigma x_1 \ \dots \ \sigma x_n \ , \quad \star , \varepsilon\ )}$$

3.3 Kappa-calculus

Hasegawa's κ-calculus [40], which featured in the introduction, as formulated by Power and Thielecke [67], embeds as follows.

$$E, E' ::= E; E' \mid \mathsf{PUSH}\, x \mid \quad \kappa x.E$$
$$M, N ::= M ; N \mid \quad [x] \quad \mid \exists x . \langle x \rangle ; M$$

A κ-abstraction (first-order λ-abstraction) decomposes into a *new* $\exists x$ and a *pop* $\langle x \rangle$, so that κx acts as a binder as well as taking input. The asymmetry of this construct, which rules out terms such as $\langle x \rangle ; \langle x \rangle ; [x]$, gives the Cartesian semantics. The machine behaviour illustrates that the decomposition gives the correct behaviour: pop and substitute.

$$\frac{(\ S\,z\ , \exists x . (\langle x \rangle ; M) , \qquad K\ )}{(\ S\,z\ , \langle y \rangle ; \{y/x\}M , \qquad K\ )}$$
$$\frac{(\ S\,z\ , \qquad \langle y \rangle , (\{y/x\}M)\ K\ )}{(\ S\ , \qquad \star , (\{z/x\}M)\ K\ )}$$
$$(\ S\ , \quad \{z/x\}M , \qquad K\ )$$

Note that Power and Thielecke's formulation of the κ-calculus, above, unfortunately does not include the unit for sequencing; an omission which is fixed in the FMC and RMC.

3.4 Pattern-matching

In a λ-calculus with *pattern-matching* [14, 20, 49], abstractions are over *patterns*, which generally are algebraic terms, instead of over variables. Since we do not have higher-order abstraction, we will stay within κ-calculus. We will use *pairing* as our only pattern, as is standard in a Cartesian setting.

$$E, E' ::= E; E' \mid \mathsf{PUSH}\, p \mid \kappa p.E \qquad p, q ::= x \mid (p, q)$$

In $\kappa p.E$, the free variables of the pattern p bind in E. The embedding in the RMC captures this by introducing each variable as *new*:

$$M, N ::= M ; N \mid [p] \mid \exists \vec{x} . \langle p \rangle ; M \qquad (\vec{x} = \mathsf{fv}(p))$$

The behaviour of the RMC-encoding is slightly more general than that of the original pattern-matching calculus, since it can deal with situations such as $[x] ; \exists y . \exists z . \langle (y, z) \rangle$ (by substituting into x), which the latter cannot. Pattern-matching calculi address this by introducing explicit product types. With that constraint, the overall behaviour agrees, as expected.

Languages using *let-bindings* with patterns, used for string diagrams or monoidal categories [41, 67], admit an analogous encoding to that of monadic let-bindings in the FMC [42], as follows.

$$\mathsf{let}\ E = p\ \mathsf{in}\ E' \quad \mapsto \quad E ; \exists \vec{x} . \langle p \rangle ; E' \quad (\vec{x} = \mathsf{fv}(p))$$

3.5 Symmetric pattern-matching

We embed a first-order fragment of *Theseus*, a *reversible* programming language which is both forwards and backwards deterministic [17, 19, 69]. Its *values* are generated from pairing and injections. Computation is performed by application of *isomorphisms*, expressed using *symmetric pattern-matching* syntax below, where the set of values v_i (respectively, w_i) are restricted by the type system to be *exhaustive* and *non-overlapping*, guaranteeing reversibility. We omit presentation of the type system here for brevity.

$$\{\ v_1 \leftrightarrow w_1 \ \mid \ \ldots \ \mid \ v_n \leftrightarrow w_n \ \}$$

We interpret values using corresponding function symbols, and embed isomorphisms as below.

$$\exists \vec{x}_1.\, \langle v_1 \rangle \,;\, [w_1] \ + \ \ldots \ + \ \exists \vec{x}_n.\, \langle v_n \rangle \,;\, [w_n]$$

Each $\vec{x}_i$ contains exactly the variables of v_i and w_i. Isomorphism *inversion* is given by $(-)^\dagger$. The application of isomorphism M to value v is encoded as $[v]\,;M$.

3.6 Prolog

We embed pure Prolog, as defined by the following grammar.

$$\begin{array}{llll} \text{Terms:} & t \coloneqq x \mid f(t_1, \ldots, t_n) & \text{Atoms:} & A \coloneqq P(t_1, \ldots, t_n) \\ \text{Clauses:} & C \coloneqq A \coloneq A_1 \ldots A_n & \text{Programs:} & L \coloneqq C_1 \ldots C_n \end{array}$$

Prolog by defualt evaluates using a *top-down* evaluation strategy, which takes the query — that is, a chosen atom — as a *goal* to be proved and non-deterministically applies clauses, decomposing the set of goals into sub-goals, succeeding when the set is empty.

We consider atoms, as well as terms, to be modelled in the RMC by algebraic terms. The embedding of a clause C with free variables $\vec{x}$ is then given as follows.

$$[\![A \coloneq A_1 \ldots A_n]\!] \ = \ \exists \vec{x}.\, \langle A \rangle \,;\, [A_1] \,;\, \ldots \,;\, [A_n]$$

The stack will hold the set of goals, with the non-deterministic sum of clauses modelling the program itself. To select the correct solution among the non-deterministic outputs, we add a new constant END to the input stack, which can only be removed when the computation is successfully completed. The embedding of a query-program pair is thus given below.

$$[\![(Q, C_1 \ldots C_n)]\!] = [\mathsf{END}] \,;\, [Q] \,;\, ([\![C_1]\!] + \ldots + [\![C_n]\!])^* \,;\, \langle \mathsf{END} \rangle$$

As we did with unification, by populating the initial stack with the query Q we collect the substitutions generated by the machine run. The completed computation then returns the same instantiation of Q as does Prolog.

Proposition 3.3 (Prolog embeds). *Given as input a query Q and program L, the Prolog abstract interpreter [74] outputs the instance Q' if and only if*

$$\frac{(\ Q\ ,\ [\![(Q, L)]\!]\ ,\ \varepsilon\ )}{(\ Q'\ ,\ \ \star\ ,\ \varepsilon\ )}$$

In contrast to top-down evaluation, *bottom-up* evaluation starts from the set of *facts* (*i.e.*, clauses with a head A but no body A_i) and non-deterministically applies rules *in reverse*, building up the set of logical consequences of facts, succeeding when the query is reached. Remarkably, the RMC reveals a syntactic duality between the two strategies: the dual embedding $[\![(Q, L)]\!]^\dagger$ evaluates the query-program pair via the bottom-up strategy.

Further, although beyond the scope of the current work, we believe that *semi-naive evaluation* – which can asymptotically improve the performance of bottom-up queries – can be formalized in our language as a source-to-source translation.

4 EQUATIONAL THEORY AND REDUCTION

In this section, we define the equational theory of the RMC and show that it is sound with respect to a natural notion of observational equivalence induced by the operational semantics. We then recover a reduction relation through a natural orientation of (an appropriate subset of) the equations, which we prove confluent. In the star-free fragment, this relation is strongly normalising, giving rise to normal forms.

4.1 The Equational Theory

The equational theory we define is conservative over Kleene algebra, but contains the following additional axioms. For the *new* variable construct, there are axioms reminiscent of those from *nominal* Kleene algebra [34]. There are then axioms internalizing unification, with each corresponding to a step in the standard Martelli and Montanari algorithm for unification (including its "occurs check") [56]. In particular, these axioms include a symmetric version of the β-law of the κ-calculus. There is also a weaker, inequational version of the η-law; and we shall see later that, in the typed setting, we recover the stronger, standard η-law. Finally, there is an axiom allowing the permutation of locations, familiar from the Functional Machine Calculus [42]. Observe that the equational theory respects the duality of the calculus, as expected.

Definition 4.1 (Equational theory). We define the *equational theory* $(=)$ over terms of the RMC to be the least congruence generated by the following axioms.

- Those of Kleene algebra: that is, $(M,\ ;, \star, +, 0)$ forms an idempotent semiring[2] together with:

$$\begin{array}{ll} M^* \ = \ \star \ + M \,;\, M^* & M \,;\, N \leq N \ \rightarrow \ M^* \,;\, N \leq N \\ M^* \ = \ \star \ + M^* \,;\, M & M \,;\, N \leq M \ \rightarrow \ M \,;\, N^* \leq M \end{array}$$

  where $\leq$ is the natural partial order: $M \leq N$ iff $M + N = N$. In the sequel, we work modulo associativity wherever possible.
- Those axiomatizing the *new* variable constructor:

$$\begin{array}{lll} \exists x.\, M & =_\nu M & (x \notin \mathrm{fv}(M)) \\ M \,;\, (\exists x.\, N) & =_\nu \exists x.\, M \,;\, N & (x \notin \mathrm{fv}(M)) \\ (\exists x.\, M) \,;\, N & =_\nu \exists x.\, M \,;\, N & (x \notin \mathrm{fv}(N)) \\ \exists x.\, (M + N) & =_\nu \exists x.\, M \ + \ \exists x.\, N & \\ \exists x.\, \exists y.\, M & =_\nu \exists y.\, \exists x.\, M & \end{array}$$

[2] That is, $(M, ;, \star)$ is a monoid and $(M, +, 0)$ is an idempotent commutative monoid, such that $;$ distributes over $+$ and 0 is annihilative.

- Those axiomatizing β- and η-equivalence and unification:

$$\exists x.\, N\,;[t]a\,;a\langle x\rangle\,;M =_\beta \{t/x\}(N\,;M) \qquad (x \notin t)$$
$$\exists x.\, N\,;[x]a\,;a\langle t\rangle\,;M =_\beta \{t/x\}(N\,;M) \qquad (x \notin t)$$
$$\exists x.\, a\langle x\rangle\,;[x]a \leq_\eta \star$$
$$a[x]\,;\langle x\rangle a =_\upsilon \star$$
$$[f(S)]a\,;a\langle f(T)\rangle =_\upsilon [S]a\,;a\langle T\rangle$$
$$[f(S)]a\,;a\langle g(T)\rangle =_\upsilon 0 \qquad (f \neq g)$$
$$[f(S)]a\,;a\langle x\rangle =_\omega 0 \qquad (x \in S)$$
$$[x]a\,;a\langle f(S)\rangle =_\omega 0 \qquad (x \in S)$$

- Those axiomatizing permutation of locations:

$$[s]a\,;b\langle t\rangle =_\pi b\langle t\rangle\,;[s]a \qquad (a \neq b)$$

The permutation law suffices to derive the fact that "push" and "pop" on distinct locations commute with each other.

Lemma 4.2 (Permutation). *The following equations are derivable.*

$$[s]a\,;[t]b = [t]b\,;[s]a \quad a\langle s\rangle\,;b\langle t\rangle = b\langle t\rangle\,;a\langle s\rangle \quad (a \neq b)$$

Proof. Derivable from π together with η, ν and β. □

A β-redex of the κ-calculus encodes as below, left, assuming a chosen, unnamed location. As such, its β-law decomposes into the following two axioms (where x is not free in t).

$$[t]\;;\exists x.\,\langle x\rangle\,;M =_\nu \exists x.\,[t]\,;\langle x\rangle\,;M =_\beta \{t/x\}M$$

The β-axiom of the RMC generalizes and symmetrizes the second equality above: the $\exists x$ construct may additionally be separated from the sub-term $[t]\,;\langle x\rangle$ by some term N, which is also open for substitution.

We now define a notion of observational equivalence of terms, which we call *machine equivalence*, via the big-step operational semantics. In the following, we take S_A to be open in general (i.e. it may contain free variables), and possibly sharing variables with M. Open memories and substitutions are considered equivalent up to a choice of globally fresh variables. Practically, this means machine runs and memories are equipped with a context; we omit this for brevity.

Definition 4.3 (Machine equivalence). *Machine equivalence* is the relation $(\sim)$ on terms defined by

$$M \sim N \quad \text{if} \quad \forall S_A.\, \lfloor S_A \Downarrow M \rfloor = \lfloor S_A \Downarrow N \rfloor\,.$$

Proposition 4.4. *Machine equivalence is a congruence.*

We can define a similar *machine refinement* $(\lesssim)$ relation on terms as follows.

$$M \lesssim N \quad \text{if} \quad \forall S_A.\, \lfloor S_A \Downarrow M \rfloor \subseteq \lfloor S_A \Downarrow N \rfloor$$

It is easy to see that machine refinement satisfies the property that $M \lesssim N$ if and only if $M + N \sim N$, and so $(\lesssim)$ stands in the same relation to $(\sim)$ as $(\leq)$ does to $(=)$ in the algebraic theory. In particular, it is a partial order closed under all contexts.

Theorem 4.5 (Soundness of equational theory). *The equational theory is sound with respect to machine equivalence:*

$$M = N \implies M \sim N\,.$$

4.2 Reduction and Confluence

It is possible to recover a reduction relation $(\twoheadrightarrow)$ from the equational theory by orientation of the equations. The β and υ equations have an obvious orientation and the ν equations are oriented to bring $\exists x$ to the front of a term, or remove it when it does not bind any variables. We do not consider the η inequation in the untyped case. To maintain confluence in the presence of the Kleene star, we find it necessary to break the symmetry of the equational theory: we can include either the left-handed *unfolding* reduction $M^* \twoheadrightarrow \star + M\,;M^*$ or its dual $M^* \twoheadrightarrow \star + M^*\,;M$, but not both. We opt, arbitrarily, for the former. We do not find it sensible to consider the conditional equation $M\,;N \leq N \longrightarrow M^*\,;N \leq N$ or its dual as part of reduction.

We consider the main interest of confluence for reduction to be in dealing with the directed β, υ and ν equations. We thus make several further adaptations to the KA fragment of reduction which simplify the confluence proof. We choose to omit the idempotence equation, in keeping with the more general multiset operational semantics; we omit consideration of associativity and commutativity of $(+)$, although the method of proof should extend straightforwardly to these cases; and we assume right-associativity of sequencing, so terms are of the form $M\,;(N\,;\ldots\,;(P\,;\star)\ldots)$, which also has the pleasant consequence of eliminating the need to consider $(\exists x.\,M)\,;N \twoheadrightarrow \exists x.\,(M\,;N)$ where $x \notin \mathrm{fv}(N)$ and $M\,;\star \twoheadrightarrow M$. For this reason, β-reduction will take place in (also right-associative) *linear contexts*, defined as follows

$$L\{\,\} ::= \{\,\} \mid M\,;L\{\,\} \mid L\{\,\}\,;M$$

and υ-reductions will contain an arbitrary post-fixed term. We also enforce a prioritization of left- over right-distributivity, although we expect this can be lifted. Finally, the permutation equation is oriented left–to–right. The reduction relation is overall clearly *sound* (but not complete) with respect to the equational theory, and thus the operational semantics.

Definition 4.6 (Reduction). The reduction relation $(\twoheadrightarrow)$ on terms is given by the rewrite rules in Figure 2, closed under all contexts.

Our proof of confluence makes use of techniques from the field of first-order term rewriting by treating the atomic terms $[t]a$, $a\langle t\rangle$, $\star$ and 0 as constants and binders $\exists x$ as distinct function symbols for each x. Local confluence is shown by analysis of critical pairs.

Lemma 4.7. *The reduction relation $(\twoheadrightarrow)$ is locally confluent.*

We now observe the star-free fragment is strongly normalising (by the measure given in the proof of Proposition 2.5) and hence confluent by Newman's Lemma [62]. Confluence of the full reduction relation then follows from an application of the Hindley-Rosen Lemma [44].

Theorem 4.8 (Confluence). *Reduction $(\twoheadrightarrow)$ is confluent.*

In the (strongly normalising) star-free fragment, we thus recover the existence of *normal forms*. Let $(\twoheadrightarrow^*)$ be the reflexive, transitive closure of $(\twoheadrightarrow)$. We state the result for the case of one location, but in general *push* and *pop* actions on distinct locations can be arranged according to a chosen ordering.

- Kleene algebra-reduction ($\twoheadrightarrow_{\mathsf{k}}$):

$$\begin{array}{rll} M^* & \twoheadrightarrow_{\mathsf{k}} & \star \,+\, M\,;M^* \\ (M+N)\,;P & \twoheadrightarrow_{\mathsf{k}} & M\,;P \,+\, N\,;P \qquad (P \neq M+N) \\ P\,;(M+N) & \twoheadrightarrow_{\mathsf{k}} & P\,;M \,+\, P\,;N \\ M+0 & \twoheadrightarrow_{\mathsf{k}} & M \\ 0+M & \twoheadrightarrow_{\mathsf{k}} & M \\ \star\,;M & \twoheadrightarrow_{\mathsf{k}} & M \\ M\,;0 & \twoheadrightarrow_{\mathsf{k}} & 0 \\ 0\,;M & \twoheadrightarrow_{\mathsf{k}} & 0 \end{array}$$

- New variable-reduction ($\twoheadrightarrow_{\nu}$):

$$\begin{array}{rll} \exists x.\, M & \twoheadrightarrow_{\nu} & M \qquad\qquad (x \notin M) \\ M\,;\exists x.\, N & \twoheadrightarrow_{\nu} & \exists x.\, M\,;N \qquad (x \notin M) \\ \exists x.\,(M+N) & \twoheadrightarrow_{\nu} & \exists x.\, M \,+\, \exists x.\, N \end{array}$$

- β- and unification-reduction ($\twoheadrightarrow_{\beta}, \twoheadrightarrow_{\upsilon}$):

$$\begin{array}{rll} \exists x.\, L\{[t]a\,;a\langle x\rangle\,;M\} & \twoheadrightarrow_{\beta} & \exists x.\, \{t/x\}L\{M\} \quad (x \notin t) \\ \exists x.\, L\{[x]a\,;a\langle t\rangle\,;M\} & \twoheadrightarrow_{\beta} & \exists x.\, \{t/x\}L\{M\} \quad (x \notin t) \\ [x]a\,;a\langle x\rangle\,;M & \twoheadrightarrow_{\upsilon} & M \\ [f(T)]a\,;a\langle f(S)\rangle\,;M & \twoheadrightarrow_{\upsilon} & [T]a\,;a\langle S\rangle\,;M \\ [f(T)]a\,;a\langle g(S)\rangle\,;M & \twoheadrightarrow_{\upsilon} & 0 \qquad\qquad (f \neq g) \\ [f(S)]a\,;a\langle x\rangle\,;M & \twoheadrightarrow_{\upsilon} & 0 \qquad\qquad (x \in S) \\ [x]a\,;a\langle f(S)\rangle\,;M & \twoheadrightarrow_{\upsilon} & 0 \qquad\qquad (x \in S) \end{array}$$

- Permutation-reduction ($\twoheadrightarrow_{\pi}$):

$$[t]a\,;b\langle s\rangle\,;M \twoheadrightarrow_{\pi} b\langle s\rangle\,;[t]a\,;M \qquad (a \neq b)$$

Figure 2: Reduction rules of the RMC

Corollary 4.9 (Normal forms). *For any closed, star-free RMC-term M, we have that $M \twoheadrightarrow^* N$, where*

$$N \equiv N_1 + \ldots + N_n\,,$$

where each N_i is sum-free and of the form

$$N_i \equiv \exists\vec{x}.\, \langle s_n\rangle\,;\ldots;\langle s_1\rangle\,;[t_1]\,;\ldots;[t_m]\,,$$

where $\vec{x}$ are the variables of s_j and t_k.

Note, normal forms are taken modulo associativity and commutativity of $(+)$ and permutation of $\exists x$. We close this section with a theorem about the equational theory (and thus about operational semantics) proved easily using the existence of normal forms.

Theorem 4.10 (Equations for duality). *We have the following:*

- *for any closed, star-free term M, we have $M \leq M\,;M^\dagger\,;M$;*
- *if M is also sum-free then $M = M\,;M^\dagger\,;M$.*

The first statement is familiar as the unique law added to KA in Kleene algebra *with converse* [15]; the second statement is familiar as the weakening of invertibility in the definition of *inverse monoids* and *inverse categories* [21].

5 THE SIMPLY-TYPED RMC

In this section, we introduce a simple typing discipline giving the input and output arity of terms, making their stack use explicit. This results in a natural language for string diagrams where types represent the input and output wires, which we explicate in the subsequent section on categorical semantics.

$$\frac{}{\star\colon n_A \Rightarrow n_A}\ \text{skip} \qquad \frac{M\colon k_A \Rightarrow m_A \qquad N\colon m_A \Rightarrow n_A}{M\,;N\colon k_A \Rightarrow n_A}\ \text{seq}$$

$$\frac{}{0\colon m_A \Rightarrow n_A}\ \text{zero} \qquad \frac{M\colon m_A \Rightarrow n_A \qquad N\colon m_A \Rightarrow n_A}{M+N\colon m_A \Rightarrow n_A}\ \text{sum}$$

$$\frac{M\colon n_A \Rightarrow n_A}{M^*\colon n_A \Rightarrow n_A}\ \text{star} \qquad \frac{M\colon m_A \Rightarrow n_A}{\exists x.\, M\colon m_A \Rightarrow n_A}\ \text{new}$$

$$\frac{}{[t]a\colon n_A \Rightarrow n_A + 1_a}\ \text{push} \qquad \frac{}{a\langle t\rangle\colon 1_a + n_A \Rightarrow n_A}\ \text{pop}$$

Figure 3: The simply-typed RMC

In a first-order setting, the power of types is naturally limited. Types do not confer termination: the fragment without Kleene star is inherently terminating, while the full RMC is non-terminating also when typed. Nor do types prevent failure, as in the term 0 or failure of unification, and this is semantically correct: failure represents the empty relation, and so should not be excluded from the typed calculus.

What types do enforce is a notion of *progress*: they guarantee that there are sufficient elements on the stack for the machine to continue. In particular, they prevent terms with Kleene star from consuming (or producing) an arbitrary number of stack elements. These constraints are essential for correct relational composition, and hence in giving a relational semantics to terms, where types represent the source and target of a relation. Specifically, types allow the sound strengthening the η-axiom to an equivalence, which is necessary for a monoidal categorical semantics.

Formally, the type of a term M will give the input and output arity *for each location*. With a single, unnamed location, we will have $M\colon m \Rightarrow n$ for natural numbers m and n; for a set of locations A, we will have an input arity m_a and output arity n_a for each $a \in A$.

Definition 5.1 (Simple types). A *memory type* $n_A = \{n_a \mid a \in A\}$ is a family of natural numbers in a (finite) set of locations A. A *type* $m_A \Rightarrow n_A$ consists of an *input* and an *output* memory type.

We may consider a memory type n_A as a function from locations to natural numbers, where $n_A(a) = n_a$ for $a \in A$ and $n_A(b) = 0$ for any $b \notin A$, and silently expand n_A to n_B for any $B \supset A$. For any natural number n we may write n_a for the singleton family in $\{a\}$. The *empty* memory type 0_A (or simply 0) is everywhere zero, and the *sum* $n_A + m_A$ of two memory types is given location-wise by $(n_A + m_A)(a) = n_A(a) + m_A(a)$.

Definition 5.2 (The simply-typed RMC). The typing rules for simply-typed RMC-terms $M\colon m_A \Rightarrow n_A$ are given in Figure 3.

The present type system, which records only the *number* of elements consumed and produced on the stack, is much simplified with respect to that of the FMC [42]. Being higher-order, the latter records also the *types* of stack elements, as a vector of types $\vec{\tau}$. Here, we only need the length of $\vec{\tau}$ as a natural number n; because the RMC is first-order, and also because we have a single-sorted

signature Σ for algebraic terms, by which we assume only a single base type for values. A many-sorted signature, where each sort is represented by a different base type, may be accommodated with stack types $\vec{\tau}$ as on the FMC.

We have the following three basic properties. First, *subject reduction* (reduction preserves types). Second, *expansion*: if the machine runs with a given input stack, it will also run with a larger stack, and the output will be equally larger. Third, *duality* exchanges input and output types.

Lemma 5.3. *The following properties of the type system hold:*

- Subject reduction: *if* $M : k_A \Rightarrow n_A$ *and* $M \to N$ *then also* $N : k_A \Rightarrow n_A$.
- Expansion: *if* $M : k_A \Rightarrow n_A$ *then* $M : k_A + m_A \Rightarrow m_A + n_A$.
- Duality: *if* $M : k_A \Rightarrow n_A$ *then* $M^\dagger : n_A \Rightarrow k_A$.

To connect types with machine behaviour, we extend types to stacks and memories in the expected way: a stack is typed $S : n$ where $n = |S|$, the length of S, and a memory is typed $S_A : n_A$ where $S_a : n_a$ for all $a \in A$. Continuation stacks and states are then typed by the following rules.

$$\frac{}{\varepsilon : n_A \Rightarrow n_A} \qquad \frac{M : k_A \Rightarrow m_A \quad K : m_A \Rightarrow n_A}{M\,K : k_A \Rightarrow n_A}$$

$$\frac{S_A : k_A \quad M : k_A \Rightarrow m_A \quad K : m_A \Rightarrow n_A}{(S_A, M, K) : 0 \Rightarrow n_A}$$

Theorem 5.4 (The machine respects types). *For a run*

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

if $(\, S_A \,,\, M \,,\, K \,) : 0 \Rightarrow n_A$ *then* $(\, T_A \,,\, N \,,\, L \,) : 0 \Rightarrow n_A$.

A machine state $(\, S_A \,,\, M \,,\, K \,)$ makes *progress* unless M is a pop $a\langle t \rangle$ and S_a is empty. Note that types rule out the latter configuration, and thus ensure progress. Since types are preserved by the above theorem, we have the following corollary.

Corollary 5.5 (Machine progress). *Any state on a run from a typed state makes progress.*

We define a typed notion of machine equivalence, where we expect input stacks to respect typing.

Definition 5.6 (Typed machine equivalence). *Typed machine equivalence* is defined as the relation $M \sim N : m_A \Rightarrow n_A$ on similarly typed terms, which holds if:

$$\forall S_A : m_A.\ \lfloor S_A \Downarrow M \rfloor = \lfloor S_A \Downarrow N \rfloor \,.$$

We can strengthen the η-axiom for the *typed* equational theory, as it ensures that the stack involved is non-empty.

Definition 5.7 (Typed equational theory). The *typed equational theory* $M = N : m_A \Rightarrow n_A$ relates similarly typed terms of the RMC by the least congruence generated by the axioms of the untyped equational theory, together with the strengthened η-axiom

$$\exists x.\, a\langle x \rangle \,;\, [x]a =_\eta \star : 1_a + m_A \Rightarrow m_A + 1_a \,.$$

The η-law of the κ-calculus is decomposed as follows, where $x \notin \mathrm{fv}(M)$, and indeed our symmetric η-axiom is equivalent.

$$\exists x.\, \langle x \rangle \,;\, [x] \,;\, M =_\nu (\exists x.\, \langle x \rangle \,;\, [x]) \,;\, M =_\eta \star \,;\, M = M$$

Theorem 5.8 (Soundness of typed equational theory). *The typed equational theory is sound for typed machine equivalence:*

$$M = N : m_A \Rightarrow n_A \implies M \sim N : m_A \Rightarrow n_A \,.$$

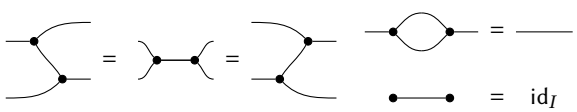
6 CATEGORICAL SEMANTICS

We give a sound and complete categorical semantics for the sum-free, star-free and function-symbol-free fragment of the RMC, illustrating how the core calculus provides a term language and operational interpretation of Frobenius monoids, and thus of a natural class of string diagrams. Following this, we show that the corresponding fragments with linear and "Cartesian" variable policies provide a term language for symmetric monoidal and Cartesian categories, respectively. The extension of the categorical semantics to include non-deterministic sum follows easily, but we leave to future work the categorical interpretation of iteration and of algebraic terms.

Frobenius monoids are used as a categorical abstraction for modelling a range of computational phenomena, many of interest to programming language theorists. While the literature contains computational interpretations of compact closed categories [18] and certain accounts of operational semantics of string diagrams [11, 12], to the knowledge of the authors, the RMC gives the first syntactic account of Frobenius monoids via a notion of β-reduction, and the first operational account of Frobenius monoids via a stack machine. To highlight the correspondence with string diagrams, we work with a single location, but the results extend easily to multiple locations [3].

Notation. Given a category C and objects $X, Y \in C$ we denote by $C(X, Y)$ the corresponding homset. We write the identity morphism on X as id_X, or simply X. Composition is written in diagrammatic order, using infix (;). We denote the tensor product of a *symmetric monoidal category (SMC)* C by $\otimes$, its unit by I and its symmetry natural transformation as sym [55]. Components of natural transformations are indexed by subscripts, which we sometimes omit. We elide all associativity and unit isomorphisms associated with monoidal categories and sometimes. Recall that a commutative monoid in a given SMC is an object X together with a multiplication $\mu : X \otimes X \to X$ and a unit $\eta : I \to X$ satisfying the expected associativity, commutativity and unit equations, and a cocommutative comonoid is defined dually. We denote by CMon the category of commutative monoids. We will use string diagrams throughout this section to describe (co-)monoids in SMCs.

Definition 6.1 (Frobenius monoid). In an SMC $(C, \otimes, 1)$, an *extra-special commutative Frobenius monoid* $(X, \delta, \epsilon, \mu, \eta)$ consists of a commutative monoid (X, μ, η) and a cocommutative comonoid (X, δ, ϵ) that satisfy the *Frobenius*, *special*, and *extra* equations, given below in the string-diagrammatic language of Figure 4.



The categorical abstraction of relations which is modelled by the RMC is that of *extra-hypergraph categories*, as defined below. This is a slight variant of the usual definition of *hypergraph category* [32], which additionally requires that the relevant Frobenius structures

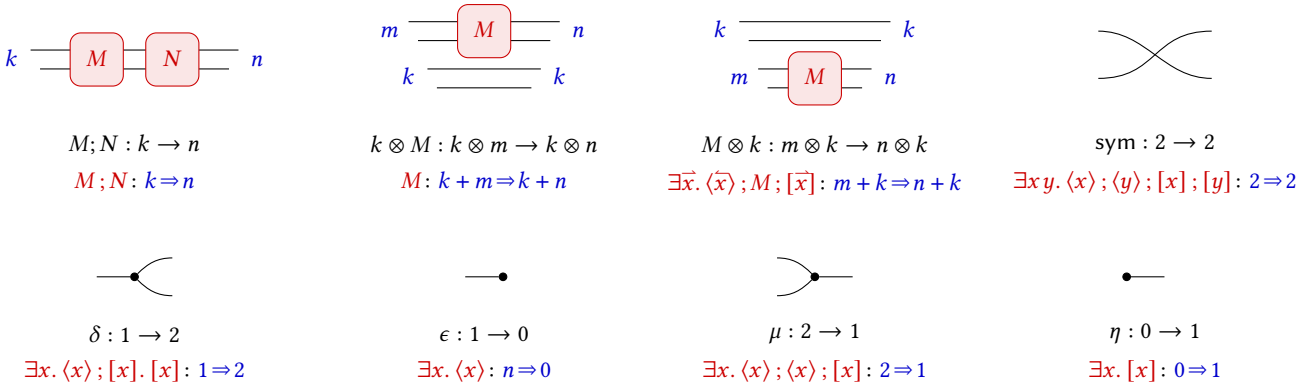


Figure 4: Equipment of the Relational Machine Category: string diagrams, categorical combinators, and RMC-terms.

satisfy the *extra* equation. Indeed, a paradigmatic example of such a category is that of sets and relations.

Definition 6.2. [Extra-hypergraph category] An *extra-hypergraph category* is a SMC C in which each object $X \in C$ is equipped with an extra-special commutative Frobenius structure $(\delta_X, \epsilon_X, \mu_X, \eta_X)$, which together satisfy the following coherences:

$$\delta_{X\otimes Y} = (\delta_X \otimes \delta_Y);(X \otimes \mathsf{sym}_{X,Y} \otimes Y) \qquad \epsilon_{X\otimes Y} = \epsilon_X \otimes \epsilon_Y$$
$$\mu_{X\otimes Y} = (X \otimes \mathsf{sym}_{X,Y} \otimes Y);(\mu_A \otimes \mu_Y) \qquad \eta_{X\otimes Y} = \eta_X \otimes \eta_Y$$

and the unit coherence that $\eta_I = \mathsf{id}_I = \epsilon_I$. A functor of extra-hypergraph categories is a strong symmetric monoidal functor that additionally preserves the Frobenius structures.

We can define the hypergraph categorical structure of the RMC by assigning terms to string diagrams as in Figure 4. We take objects to be the natural numbers, and wires to represent the input and output stacks, with the head of the stack at the top. In contrast to the λ-calculus, categorical composition of terms is given by sequential compostion $M\,;N$ rather than by substitution. Right-action $k \otimes M$ of the tensor product is given by stack expansion (as described in Lemma 5.3). Left-action $M \otimes k$ lifts the k arguments from the stack as variables $\vec{x}$, to restore them after evaluating M on the remaining stack. The remaining equipment is familiar from the discussion in the introduction of this paper.

Definition 6.3. The *Relational Machine Category*, $\mathsf{RMC}[\Sigma]$, of RMC-terms, is defined by the following data.

- Objects: natural numbers
- Morphisms: $\mathsf{RMC}[\Sigma](m, n)$ is the set of closed simply-typed terms $M : m \Rightarrow n$ over Σ, modulo (=).
- Identity: at type n, the term $\star : n \Rightarrow n$
- Composition: for terms $M : k \Rightarrow m$ and $N : m \Rightarrow n$, their composition is $M\,;N : k \Rightarrow n$
- Tensor product: on objects $m \otimes n = m + n$, with unit $I = 0$.
- The right- and left-action of the tensor product on morphisms, the symmetry, and the Frobenius monoids at type 1 are given by Figure 4. The associators and unitors are given by the identity.

Note that it suffices to define a Frobenius monoid on the generating object 1, with Frobenius monoids at n defined using the compatibility conditions of Definition 6.2.

Theorem 6.4 (Soundness). *The Relational Machine Category $\mathsf{RMC}[\Sigma]$ is an extra-hypergraph category.*

It is immediate from the axioms of Kleene algebra that $\mathsf{RMC}[\Sigma]$ is also *CMon-enriched*, *i.e.*, every homset $\mathsf{RMC}[\Sigma](n, m)$ is a commutative monoid with addition $(+)$ and unit 0 and composition of morphisms distributing over addition. This is an easy and appropriate extension of the model to incorporate non-determinism.

Completeness of the categorical semantics for the core fragment follows an easy normal form argument. Let $\emptyset$ denote the empty signature.

Theorem 6.5 (Hypergraph completeness). *The subcategory of sum-free, star-free terms of $\mathsf{RMC}[\emptyset]$ is equivalent to the free extra-hypergraph category.*

Here, and below, the free category in question is generated over a single object (and the empty set of morphisms).

By restricting the variable policy of this fragment appropriately, we similarly recover term languages for symmetric monoidal and Cartesian categories. Let *linear* RMC-terms to be those for which each occurring variable appears exactly once in a *push* and once in a *pop*. Then, since linear RMC-terms containing no function symbols are exactly permutations, we have the following.

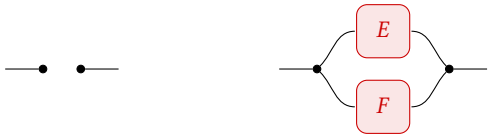
Theorem 6.6 (Linear completeness). *The subcategory of linear, sum-free, star-free terms of $\mathsf{RMC}[\emptyset]$ is equivalent to the free symetric monoidal category.*

We similarly take the *Cartesian* RMC-terms to be those for which each bound variable occurs exactly once in a *pop* (but possibly in many *pushes*), *and* in which no *pop* contains anything other than a variable. Observe further that an RMC-signature Σ is just an algebraic signature with no equations, which can be used to generate a free Cartesian category. Then a simple normal form argument proves the following.

Theorem 6.7 (Cartesian completeness). *The subcategory of sum-free, star-free terms of $\mathsf{RMC}[\Sigma]$ is equivalent to the free Cartesian category generated over Σ.*

Consideration of the case where the signature includes an equational theory is left for future work: although it is unproblematic in the Cartesian case, the general case would require working with unification *modulo a theory*.

We conclude this section by giving some simple examples of RMC-terms and their corresponding string diagrams. Following existing work on string diagrams for (the positive fragment of) Tarski's calculus of relations [6, 10, 13, 38, 75], we give terms and diagrams whose anticipated relational denotation is given by *top* (*i.e.*, everything is related), *intersection*, and *converse*. The remaining operations of the calculus of relations are already included in the grammar of regular expressions (see subsection 3.1). Top ($\top$) and intersection ($\cap$) are encoded as below.



$\top$ $\quad E \cap F$

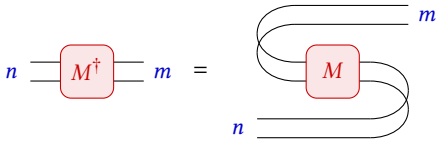
$\exists x.\, \exists y.\, \langle x\rangle\,;[y] \qquad \exists x.\, \exists y.\, \langle x\rangle\,;[x]\,;E\,;\langle y\rangle\,;[x]\,;F\,;\langle y\rangle\,;[y]$

The compositions $\eta;\delta$ and $\mu;\epsilon$ form the cups and caps of a (self-dual) compact closed structure. By β-reduction we have that $\eta;\delta = \exists \vec{x}.\, [\vec{x}]\,;[\vec{x}]$, and dually, as shown below.

$\exists x.\, [x]\,;[x] \qquad \exists x.\, \langle x\rangle\,;\langle x\rangle$

We then have the following proposition characterizing the converse of a typed RMC-term.

Proposition 6.8 (Typed Duality). *For closed* $M\colon m \Rightarrow n$ *we have* $M^\dagger = \exists \vec{x}\vec{y}.\, \langle\vec{x}\rangle\,;[\vec{y}]\,;M\,;\langle\vec{x}\rangle\,;[\vec{y}]\colon n \Rightarrow m$, *where* $|\vec{x}| = n$, $|\vec{y}| = m$.



7 RELATIONAL SEMANTICS

In this section, we investigate in detail the relational semantics of the typed RMC. Although the denotational semantics is very close to the input/output behaviour of the big-step semantics, there are some differences: forgetting the output substitution, and the interpretation of $\exists x.\, M$ and $\langle t\rangle$. The former is interpreted semantically as the union of all possible interpretations of x as some closed term t, leaving the latter to become a simple equality check predicate; there is no need for unification since free variables are dealt with by (possibly infinite) non-determinism of the interpretation of $\exists x$.

For simplicity, we again work with a single location, but the results generalize easily. Now, let us recall the extra-hypergraph structure of the category of sets and relations.

Definition 7.1. The category Rel of sets and relations forms an extra-hypergraph category where the tensor product is given by the Cartesian product, the unit given by the singleton set $\{\varepsilon\}$, and a Frobenius monoid on each object A given by $\delta_A = \{(x,(x,x)) \mid x \in A\}$, $\epsilon_A = \{(x,\varepsilon) \mid x \in A\}$ and with μ_A and η_A by their respective converses.

The relational semantics of typed RMC-terms can easily be read off their normal forms: for example, compare the relational semantics of δ, ϵ, μ and η with the RMC-terms interpreting the same morphisms in Figure 4. We define it formally as an extra-hypergraph functor $[\![-]\!] : \mathrm{RMC}[\Sigma] \to \mathrm{Rel}$. Let $T0$ be the set of *closed* algebraic terms built over Σ, *i.e.* containing no variables. To interpret open terms, we define a valuation $v : \mathrm{Var} \to T0$ on the set of variables Var; the valuation $v\{x \leftarrow t\}$ assigns t to x and otherwise behaves as v. We elide the isomorphism between closed algebraic terms t in the syntax of RMC-terms and $t \in T0$. We extend the action of v to algebraic terms with variables Γ so that $v(f(t_1,\ldots,t_n)) = f(v(t_1),\ \ldots, v(t_n))$. In the following, we consider relations $A \to B$ as functions $A \to \mathcal{P}(B)$, where $\mathcal{P}$ is the powerset functor, and we will write $f^n : X \to X$ for the n-fold composition of a function $f : X \to X$ and X^n for the n-fold product of a set X.

Definition 7.2 (Relational semantics). The *relational semantics* of the typed RMC is defined inductively on types by $[\![n]\!] = T0^n$. We define the relation $[\![M\colon m \Rightarrow n]\!]_v$ inductively on the type derivation of terms as follows, and may abbreviate this as $[\![M]\!]_v$.

$$
\begin{aligned}
[\![\star\colon n \Rightarrow n]\!]_v(S) &= [S] \\
[\![M\,;N\colon k \Rightarrow n]\!]_v(S) &= [U \mid T \in [\![M]\!]_v(S),\ U \in [\![N]\!]_v(T)] \\
[\![0\colon m \Rightarrow n]\!]_v(S) &= [\,] \\
[\![M + N\colon m \Rightarrow n]\!]_v(S) &= [\![M]\!]_v(S) \sqcup [\![N]\!]_v(S) \\
[\![[t]\colon n \Rightarrow n+1]\!]_v(S) &= [S\, v(t)] \\
[\![\langle t\rangle\colon n+1 \Rightarrow n]\!]_v(S\, u) &= \begin{cases} [\vec{s}] & \text{if } t = v(u) \\ [\,] & \text{otherwise} \end{cases} \\
[\![\exists x.\, M\colon m \Rightarrow n]\!]_v(S) &= \bigsqcup_{t \in T0} [\![M]\!]_{v\{x \leftarrow t\}}(S) \\
[\![M^*\colon n \Rightarrow n]\!]_v(S) &= \bigsqcup_{n \in \mathbb{N}} [\![M]\!]_v^n(S)
\end{aligned}
$$

Given a closed term M, its relational semantics $[\![M]\!]$ (written without a subscript) is given by $\lfloor [\![M]\!]_\varepsilon \rfloor$, where ε is the empty valuation.

From the following lemma, soundness of the semantics follows.

Lemma 7.3 (Substitution). *For any* $M\colon m \Rightarrow n$, *we have*

$$[\![M\colon m \Rightarrow n]\!]_{v\{x \leftarrow t\}} = [\![\{t/x\}M\colon m \Rightarrow n]\!]_v\ .$$

Lemma 7.4 (Relational semantics is hypergraph functor). *The relational semantics* $[\![-]\!] : \mathit{RMC}[\Sigma] \to \mathit{Rel}$ *is a well-defined extra-hypergraph functor.*

In fact, if we additionally consider $\mathrm{RMC}[\Sigma]$ and Rel as CMon-enriched categories, with the monoid on homsets in Rel given by the union of relations, it is easy to see the relational semantics functor is also CMon-enriched functor.

8 ENCODING COMPUTATIONAL MODELS, II

We conclude with further encodings of computational models, demonstrating the use of locations, an innovation of the Functional Machine Calculus [4, 42] that may capture stateful behaviour.

8.1 Guarded Command Language

In 1975 Dijkstra introduced the *Guarded Command Language (GCL)*: a simple non-deterministic imperative programming language [28].

The eponymous *guarded command* $B \to S$ is given by a sequence of statements S to be evaluated when the *guard* Boolean expression B is true. Statements are atomic actions — here, we consider writing to a memory cell ($a := N$) — or are sets of guarded commands wrapped in *conditional* (if) or *iteration* (do) keywords. The former construct executes (non-deterministically) one of the statements whose guard is true, and aborts execution if none are. The iteration construct successively executes (again, non-deterministically) one of the statements whose guard is true, and terminates successfully when none are. Reading from a memory cell a is denoted $!a$. The full grammar of the GCL is given below:

Boolean Expressions:	$B ::= \mathsf{tt} \mid \mathsf{ff} \mid \dots$
Numeric Expressions:	$N ::= n \mid !a \mid \dots$
Guarded Commands:	$C ::= \mathsf{abort} \mid C_1 \square C_2 \mid B \to S$
Statements:	$S ::= \mathsf{skip} \mid S_1; S_2 \mid \mathsf{if}\ C \mid \mathsf{do}\ C \mid a := N$

where $n \in \mathbb{Z}$ and a ranges over a set of variable names. [3] Input and output is performed by manipulation of memory cells, which we take to hold integers: so, in standard imperative style, programs are called for their side-effects.

The embedding of the GCL in the RMC is given below.

$$\begin{aligned} [\![\mathsf{tt}]\!] &= [\top] & [\![!a]\!] &= \exists x.\, a\langle x\rangle ; [x]a ; [x] \\ [\![\mathsf{ff}]\!] &= [\bot] & [\![a := V]\!] &= [\![V]\!] ; \exists xy.\, a\langle x\rangle ; \langle y\rangle ; [y]a \\ [\![n]\!] &= [n] & [\![B \to S]\!] &= [\![B]\!] ; \langle\top\rangle ; [\![S]\!] \\ [\![\mathsf{abort}]\!] &= 0 & [\![C_1 \square C_2]\!] &= [\![C_1]\!] + [\![C_2]\!] \\ [\![\mathsf{skip}]\!] &= \star & [\![S_1; S_2]\!] &= [\![S_1]\!] ; [\![S_2]\!] \\ [\![\mathsf{if}\ C]\!] &= [\![C]\!] & [\![\mathsf{do}\ C]\!] &= [\![C]\!]^* \end{aligned}$$

Encodings of global memory cells are familiar from the FMC [42]: locations model mutable variables, whose value is held as the single item on the corresponding stack. The result of an expression, including reading from store, is pushed to the main (unnamed) stack; a store update replaces the stored value with the top value on the main stack.

Proposition 8.1 (The RMC implements GCL). *If the GCL successfully executes a guarded command C on input memory S_A, with output memory T_A, then there exists an RMC run*

$$\frac{(\ S_A\ ,\ [\![C]\!]\ ,\ \varepsilon\ )}{(\ T_A\ ,\quad \star\ ,\ \varepsilon\ )}\ .$$

Note that our encoding in fact outputs (non-deterministically) the memory at every stage of the computation, and not only the memories at the end of a successful GCL execution.

8.2 Turing Machines

We demonstrate an encoding of a Turing machine $\mathcal{M}$ with a set of *states* S, initial state $i \in S$, halting state $h \in S$, alphabet Σ, blank symbol $0 \in \Sigma$, and transition function $\delta : (S \setminus \{h\}) \times \Sigma \to S \times \Sigma \times \{\boldsymbol{L}, \boldsymbol{R}\}$. To do so, we work with an RMC signature consisting of $S \cup \Sigma$ and three locations l, r, and q, used to record the tape to the left of the head, the tape under and to the right of the head (in reverse), and the current state, respectively.

A Turing machine $\mathcal{M}$ then encodes as follows. Below left are the encodings of tape moves $m \in \{\boldsymbol{L}, \boldsymbol{R}\}$ and of a transition in δ as a five-tuple $d = (s, a, s', a', m)$. Then $\mathcal{M}$ encodes as an iterated non-deterministic sum of transitions, below right.

$$\begin{aligned} [\![\boldsymbol{L}]\!] &= \exists x.\, l\langle x\rangle ; [x]r \\ [\![\boldsymbol{R}]\!] &= \exists x.\, r\langle x\rangle ; [x]l \\ [\![d]\!] &= q\langle s\rangle ; r\langle a\rangle ; [s']q ; [a']r ; [\![m]\!] \end{aligned} \qquad [\![\mathcal{M}]\!] = \left(\sum_{d\in\delta} [\![d]\!]\right)^*$$

Instantiating the tape in both directions S_l and S_r with streams of blank symbols[4], we have the following result.

Proposition 8.2 (The RMC implements Turing machines). *A Turing machine $\mathcal{M}$ halts with tape T_l and T_r to the left and right of the head, respectively, if and only if there exists a machine run*

$$\frac{(\ S_l \cdot S_r \cdot s_q\ ,\ [\![\mathcal{M}]\!]\ ,\ \varepsilon\ )}{(\ T_l \cdot T_r \cdot h_q\ ,\quad \star\ ,\ \varepsilon\ )}\ .$$

8.3 Interaction Nets

Interaction nets are a graphical model of computation first introduced as a generalisation of multiplicative proof nets [52]. A term calculus for interaction nets [30] is given as follows. A *net* is a pair $\langle T \mid \Delta\rangle$ where T is a stack of algebraic terms over a signature Σ, and $\Delta = \{t_1 \doteq u_1, \dots, t_n \doteq u_n\}$ is a set of formal equations. Each variable occurs at most twice in a net. Rewriting of nets is performed according to a set of rules $\mathcal{R}$, with each rule a pair of terms $f(S) \bowtie g(U)$ such that $f \neq g$ and with each variable occurring exactly twice. The set $\mathcal{R}$ is further required to be *deterministic*: there is at most one rule for each pair of f and g; and *symmetric*: if $f(S) \bowtie g(U)$ then $g(U) \bowtie f(S)$. Variables are *local* to each rule, and are instantiated with fresh variables during computation.

The evaluation relation ($\to_{\mathcal{R}}$) is generated by the following rules, adapted from [30]. We write $T \doteq U$ for the pairwise equations of T and U of equal length, and ε the empty set of equations.

$$\begin{aligned} \langle T \mid \Delta, x \doteq u\rangle &\to_{\mathcal{R}} \langle \{u/x\}T \mid \{u/x\}\Delta\rangle \\ \langle T \mid \Delta, u \doteq x\rangle &\to_{\mathcal{R}} \langle \{u/x\}T \mid \{u/x\}\Delta\rangle \\ \langle T \mid \Delta, f(R) \doteq g(V)\rangle &\to_{\mathcal{R}} \langle T \mid \Delta, R \doteq S, U \doteq V\rangle \quad (f(S) \bowtie g(U)) \end{aligned}$$

To encode interaction nets we use two locations l, r to hold the left and right sides of each equation in $\Delta = \{t_1 \doteq u_1, \dots, t_n \doteq u_n\}$, which is then interpreted as the memory below left. Below it is the interpretation of a rule in $\mathcal{R}$, where $\vec{x}$ are the variables occurring in S and U. A set of rules $\mathcal{R}$ is encoded as below right. The terms T in a configuration $\langle T \mid \Delta\rangle$ are held on the unnamed main location.

$$\begin{aligned} [\![\{t_i \doteq u_1\}_{i\le n}]\!] &= (t_1 \dots t_n)_l \cdot (u_1 \dots u_n)_r \\ [\![f(S) \bowtie g(U)]\!] &= \exists \vec{x}.\, l\langle f(S)\rangle ; r\langle g(U)\rangle \end{aligned} \qquad [\![\mathcal{R}]\!] = \left(\sum_{r\in\mathcal{R}} [\![r]\!]\right)^*$$

[3] Note, we make several simplifications with regards to the original grammar. In particular, guarded commands and lists of statements are allowed to be empty. We consider only a meagre set of expressions, but, with a little more work, *e.g.* operators such as addition or comparison may be accounted for.

[4] It also easy enough to simulate an infinite tape with stacks, using special end-of-stack markers $\bot$ and $\top$. The encoding of a read of $a \in \Sigma$ then additionally checks for the end of either stack, pushing markers along as needed: that is, in the case of $a = 0$, using $r\langle a\rangle + (r\langle\bot\rangle ; [\bot]l) + (r\langle\top\rangle ; [\top]r)$ instead of $r\langle a\rangle$.

Proposition 8.3 (Interaction nets embed). *For a set of rules $\mathcal{R}$ and a configuration $\langle T \mid \Delta \rangle$ we have that $\langle T \mid \Delta \rangle \twoheadrightarrow^*_{\mathcal{R}} \langle U \mid \varepsilon \rangle$ if and only if*

$$\frac{(\ T \cdot [\![\Delta]\!] \quad , [\![\mathcal{R}]\!] \ , \varepsilon\ )}{(\ U \cdot \varepsilon_l \cdot \varepsilon_r \ , \qquad \star \ , \varepsilon\ )}.$$

8.4 Petri Nets

Petri nets are a type of discrete event dynamic system, first introduced in [65] as a graphical model of concurrency. They have since found real-world applications in a range of areas, including transport, manufacturing, fault diagnosis, and power systems [64].

We define a Petri net as a pair (P, T) of finite sets of *places* P and *transitions* $T \subset \mathcal{M}_f(P) \times \mathcal{M}_f(P)$, where $\mathcal{M}_f$ denotes finite multisets. A *state* $s \in \mathcal{M}_f(P)$ is a distribution of *tokens* over the places P. A transition $t = (t^-, t^+)$ may *fire* as follows, where $\subseteq$, $(\setminus)$ and $\uplus$ are multiset inclusion, difference, and union respectively.

$$s \ \mapsto \ (s \setminus t^-) \uplus t^+ \qquad (t^- \subseteq s)$$

Petri nets encode directly in the RMC with a signature $\Sigma = \{\circ\}$ consisting of only a constant $\circ$ for tokens, and the set of places P as the locations. A state s embeds as the memory $[\![s]\!] = S_P$ consisting of stacks of tokens, where each stack S_p holds the number of tokens at p in s. Writing $[p_1, \ldots, p_n]$ for a multiset of places, transitions embed as below, and a Petri net as the term $[\![(P, T)]\!] = (\sum_{t \in T} [\![t]\!])^*$.

$$[\![\ ([p_1, \ldots, p_n], [q_1, \ldots, q_m])\ ]\!] \ = \\ p_1\langle\circ\rangle\,;\ \ldots\ ;\, p_n\langle\circ\rangle\,;\,[\circ]q_1\,;\ \ldots\ ;\,[\circ]q_n$$

Proposition 8.4. *For a Petri net (P, T) and states $s, s' \in \mathcal{M}_f(P)$, we have $s \mapsto^* s'$ if and only if*

$$\frac{(\ [\![s]\!] \quad , [\![(P,T)]\!] \ , \varepsilon\ )}{(\ [\![s']\!] \ , \qquad \star \ , \varepsilon\ )}.$$

9 CONCLUSION AND FURTHER RESEARCH

We believe that by exposing the duality of relational programming in syntax we have arrived at a natural and convincing foundational model of the paradigm. We hope that the RMC will allow the development of tools and reasoning techniques for relational programming, similar to those founded in the λ-calculus which have proved so important for functional programming: type systems and denotational semantics, operational semantics, equational reasoning and confluent reduction. In this paper, we proved fundamental results supporting this new calculus, focussing on showing it meets the design criteria set out and justified in the introduction. We now outline several areas of future work.

9.1 Further Research

Foundations for functional logic programming. It appears straightforward to conceive of a higher-order RMC, based on the higher-order κ-calculus and Functional Machine Calculus (FMC) [4, 42, 67], which would have potential as a foundational model of *functional* logic programming. Already, the FMC, which seeded this work, has shown how to seamlessly integrate the higher-order κ-calculus with global state, by the use of locations — which are also present in the RMC. Thus, there is potential, even, for a unification of logic, functional *and imperative* programming.

Bridging programming languages and string diagrams. String diagrams have found practical applications in an increasing range of domains [23, 47, 57, 61]. This often involves their implementation at scale: for example, the ZX-diagrams used for quantum circuit optimization may have on the order of 10^4–10^7 nodes. As string diagrams grow, tools to represent and reason about them become more important [7, 8]. This can be seen by the introduction of circuit description languages which allow the high-level specification of algorithms which can yet be compiled down to circuit (or diagram) level [33, 54], as well as by the interest in techniques for string diagram rewriting using hypergraphs and DPO-rewriting [7, 8]. However, term rewriting is well established in comparison to its graphical counterpart, and could serve to complement hypergraph rewriting techniques. The RMC gives a principled foundation to programming languages for string diagrams — at least for the class given by Frobenius monoids. A higher-order RMC would allow the factorization of large diagrams and their compact representation.

Although the class of string diagrams this paper deals with is limited, we believe it is possible to extend the RMC in a way which gives a unifying account of — and uniform syntax and operational semantics to — a wide class of string diagrams via *unification modulo theory*. For example, by considering values from the theory of commutative monoids, and performing unification modulo this theory, we can represent as computations string diagrams of *graphical resource algebra* [9]; by considering values from the theory of Abelian groups, we can represent as computations string diagrams of *graphical linear algebra* and the *phase-free ZX calculus*. In each of these cases we have a constant 0, a binary function symbol $+$, and terms

$$[0] \qquad \exists xy.\ \langle x\rangle\,;\,\langle y\rangle\,;\,[x+y]$$
$$\langle 0\rangle \qquad \exists xy.\ \langle x+y\rangle\,;\,[y]\,;\,[x]$$

providing a second commutative monoid structure and its dual, with the expected equational theory induced by the operational semantics of unification modulo. We hope further for links to (differential) linear logic and monoidal differential categories via the observation that the free commutative (co-)monoid (co-)monad models the exponential modality [5].

Weighted relations. Although there are a wide range of languages with a relational semantics, the range of languages taking a *weighted* relational semantics [53, 76] of some form is vast. Broadly conceived, these include linear-algebraic [2, 77], probabilistic and quantum λ-calculi [71], and classes of string diagrams modelling similar domains [6, 10, 13, 23, 24, 38, 47]. In fact, the RMC generalizes easily to include *weighted* terms and thus weighted non-determinism. Restricting this general non-determinism in order to give accounts of stochastic or unitary processes in probabilistic or quantum domains, respectively, is less immediate and will require more sophisticated type systems, as in [17, 69]. However, Frobenius monoids have already been used to axiomatize the exact conditioning operation of probabilistic programming languages (albeit in the finite dimensional case) [27], and thus investigation into a *probabilistic machine calculus* appears warranted [68]. Quantum processes also admit Frobenius monoids as a fundamental primitive[23, 24, 47]; furthermore, they include as essential aspects both non-determinism and reversiblity, and thus investigation into a *quantum machine calculus* appears warranted, too.

REFERENCES

[1] Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: a functional datalog. In *Proc. 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 214–227. ACM, 2016.

[2] Ali Assaf, Alejandro Díaz-Caro, Simon Perdrix, Christine Tasson, and Benoît Valiron. Call-by-value, call-by-name and the vectorial behaviour of the algebraic lambda-calculus. *Logical Methods in Computer Science*, 10(4), 2014.

[3] Chris Barrett. *On the Simply-Typed Functional Machine Calculus: Categorical Semantics and Strong Normalisation.* PhD thesis, University of Bath, 2023.

[4] Chris Barrett, Willem Heijltjes, and Guy McCusker. The Functional Machine Calculus II: Semantics. In *31st EACSL Annual Conference on Computer Science Logic (CSL)*, volume 252 of *LIPIcs*, pages 10:1–10:18, 2023.

[5] Richard Blute, J. Robin B. Cockett, and Robert A. G. Seely. Differential categories. *Mathematical Structures in Computer Science*, 16(6):1049–1083, 2006.

[6] Filippo Bonchi, Alessandro Di Giorgio, and Alessio Santamaria. Deconstructing the calculus of relations with tape diagrams. *Proceedings of the ACM on Programming Languages*, 7(POPL):1864–1894, 2023.

[7] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. String diagram rewrite theory I: Rewriting with Frobenius structure. *Journal of the ACM*, 69(2), 2022.

[8] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. String diagram rewrite theory II: Rewriting with symmetric monoidal structure. *Mathematical Structures in Computer Science*, 32(4):511–541, 2022.

[9] Filippo Bonchi, Joshua Holland, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Diagrammatic algebra: From linear to concurrent systems. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019.

[10] Filippo Bonchi, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Graphical affine algebra. In *Proc. 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2019.

[11] Filippo Bonchi, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Contextual equivalence for signal flow graphs. In Jean Goubault-Larrecq and Barbara König, editors, *Proc. Foundations of Software Science and Computation Structures (FOSSACS) - 23rd International Conference*, volume 12077 of *Lecture Notes in Computer Science*, pages 77–96. Springer, 2020.

[12] Filippo Bonchi, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Bialgebraic foundations for the operational semantics of string diagrams. *Information and Computation*, 281:104767, 2021.

[13] Filippo Bonchi, Jens Seeber, and Paweł Sobociński. Graphical conjunctive queries. In *Proc. 27th EACSL Annual Conference on Computer Science Logic (CSL)*, volume 119 of *LIPIcs*, pages 13:1–13:23, 2018.

[14] V. Breazu-Tannen, D. Kesner, and L. Puel. A typed pattern calculus. In *Proc. Eighth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 262–274, 1993.

[15] Paul Brunet and Damien Pous. Algorithms for Kleene algebra with converse. *Journal of Logical and Algebraic Methods in Programming*, 85(4):574–594, 2016.

[16] Ana C. Calderon and Guy McCusker. Understanding game semantics through coherence spaces. *Electronic Notes in Theoretical Computer Science*, 265:231–244, 2010.

[17] Kostia Chardonnet, Alexis Saurin, and Benoît Valiron. A Curry-Howard correspondence for linear, reversible computation. In Bartek Klin and Elaine Pimentel, editors, *Proc. 31st EACSL Annual Conference on Computer Science Logic (CSL)*, volume 252 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[18] Chao-Hong Chen and Amr Sabry. A computational interpretation of compact closed categories: reversible programming with negative and fractional types. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[19] Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. Symmetries in reversible programming: from symmetric rig groupoids to reversible programming languages. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–32, 2022.

[20] Horatiu Cirstea and Claude Kirchner. Combining first and higher-order computations using rho-calculus: Towards a semantics of ELAN. In Dov M. Gabbay and Maarten de Rijke, editors, *Proc. Frontiers of Combining Systems, Second International Workshop (FroCoS)*, pages 95–120. Research Studies Press/Wiley, 1998.

[21] J. Robin B. Cockett and Stephen Lack. Restriction categories I: categories of partial maps. *Theoretical Computer Science*, 270(1-2):223–259, 2002.

[22] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[23] Bob Coecke and Ross Duncan. Interacting quantum observables: Categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, 2011.

[24] Bob Coecke, Dusko Pavlovic, and Jamie Vicary. A new description of orthogonal bases. *Mathematical Structures in Computer Science*, 23(3):555–567, 2013.

[25] John Horton Conway. *Regular algebra and finite machines.* Chapman and Hall Mathematics Series. Chapman and Hall, 1971.

[26] Daniel de Carvalho. Execution time of λ-terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018.

[27] Elena Di Lavore and Mario Román. Evidential decision theory via partial markov categories. In *Proc. 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE, 2023.

[28] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[29] Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.

[30] Maribel Fernández and Ian Mackie. A calculus for interaction nets. In Gopalan Nadathur, editor, *Proc. Principles and Practice of Declarative Programming, International Conference (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 170–187. Springer, 1999.

[31] Andrzej Filinski. Declarative continuations: an investigation of duality in programming language semantics. In David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné, editors, *Proc. Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 224–249. Springer, 1989.

[32] Brendan Fong and David I. Spivak. Hypergraph Categories. *CoRR*, abs/1806.08304, 2018.

[33] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. Proto-quipper with dynamic lifting. *Proceedings of the ACM on Programming Languages*, 7(POPL):309–334, 2023.

[34] Murdoch James Gabbay and Vincenzo Ciancia. Freshness and name-restriction in sets of traces with names. In Martin Hofmann, editor, *Proc. Foundations of Software Science and Computational Structures (FOSSACS)- 14th International Conference*, volume 6604 of *Lecture Notes in Computer Science*, pages 365–380, 2011.

[35] Dan R. Ghica and Guy McCusker. The regular-language semantics of second-order idealized ALGOL. *Theoretical Computer Science*, 309(1):469–502, 2003.

[36] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[37] Jean-Yves Girard. Normal functors, power series and λ-calculus. *Annals of Pure and Applied Logic*, 37(2):129–177, 1988.

[38] Tao Gu and Fabio Zanasi. Functorial semantics as a unifying perspective on logic programming. In *Proc. 9th Conference on Algebra and Coalgebra in Computer Science (CALCO)*, volume 211 of *LIPIcs*, pages 17:1–17:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[39] Michael Hanus. Functional Logic Programming: From Theory to Curry. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics: Essays in Memory of Harald Ganzinger*, Lecture Notes in Computer Science, pages 123–168. Springer, Berlin, Heidelberg, 2013.

[40] Masahito Hasegawa. Decomposing typed lambda calculus into a couple of categorical programming languages. In *Proc. Category Theory and Computer Science (CTCS), 6th International Conference*, volume 953 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 1995.

[41] Masahito Hasegawa. *Models of Sharing Graphs: A categorical semantics of let and letrec.* PhD thesis, University of Edinburgh, United Kingdom, 1997.

[42] Willem Heijltjes. The Functional Machine Calculus. In *Proc. 38th Conference on the Mathematical Foundations of Programming Semantics (MFPS)*, volume 1 of *ENTICS*, 2022.

[43] Jason Hemann and Daniel P. Friedman. microkanren : A minimal functional core for relational programming. In *Workshop on Scheme and Functional Programming (Scheme '13)*, 2013.

[44] J. R. Hindley. *The Church Rosser Property and a Result in Combinatory Logic.* PhD thesis, University of Newcastle-upon-Tyne, 1964.

[45] C. A. R. Hoare, Ian J. Hayes, Jifeng He, Carroll Morgan, A. W. Roscoe, Jeff W. Sanders, Ib Holm Sørensen, J. Michael Spivey, and Bernard Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[46] C. A. R. Hoare and Stephan van Staden. The laws of programming unify process calculi. *Science of Computer Programming*, 85:102–114, 2014.

[47] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Completeness of the ZX-calculus. *Logical Methods in Computer Science*, 16(2), 2020.

[48] Robin Kaarsgaard and Mathys Rennela. Join inverse rig categories for reversible functional programming, and beyond. *Electronic Proceedings in Theoretical Computer Science*, 351:152–167, 2021.

[49] Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, 398(1):16–31, May 2008.

[50] Dexter Kozen. On Hoare logic and Kleene algebra with tests. *ACM Transactions on Computational Logic*, 1(1):60–76, 2000.

[51] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, September 2007.

[52] Yves Lafont. Interaction Nets. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 95–108. ACM Press, 1990.

[53] Jim Laird, Giulio Manzonetto, Guy McCusker, and Michele Pagani. Weighted relational models of typed lambda-calculi. In *Proc. 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 301–310. IEEE, 2013.

[54] Bert Lindenhovius, Michael W. Mislove, and Vladimir Zamdzhiev. Enriching a linear/non-linear lambda calculus: A programming language for string diagrams. In Anuj Dawar and Erich Grädel, editors, *Proc. 33rd Annual ACM/IEEE Symposium*

on Logic in Computer Science (LICS), pages 659–668. ACM, 2018.
[55] Saunders Mac Lane. *Categories for the working mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1998.
[56] Alberto Martelli and Ugo Montanari. An efficien unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
[57] Paul-André Melliès. Local states in string diagrams. In *Rewriting and Typed Lambda Calculi - Joint International Conference (RTA-TLCA)*, volume 8560 of *Lecture Notes in Computer Science*, pages 334–348, 2014.
[58] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proc. Third International Conference on Logic Programming*, volume 225, pages 448–462, 1986.
[59] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.
[60] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
[61] Keiko Nakata and Masahito Hasegawa. Small-step and big-step semantics for call-by-need. *CoRR*, abs/0907.4640, 2009.
[62] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.
[63] C.-H. Luke Ong. Quantitative semantics of the lambda calculus: Some generalisations of the relational model. In *Proc. 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017.
[64] Pawel Pawlewski. *Petri Nets*. IntechOpen, Rijeka, Feb 2010.
[65] Carl Petri. *Kommunikation mit Automaten*. PhD thesis, TU Darmstadt, 1962.
[66] Andrew M. Pitts and Ian David Bede Stark. Observable properties of higher order functions that dynamically create local names, or what's new? In *Proc. Mathematical Foundations of Computer Science, 18th International Symposium, MFCS'93*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer, 1993.
[67] John Power and Hayo Thielecke. Closed Freyd- and kappa-categories. In *Proc. Automata, Languages and Programming, 26th International Colloquium, ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 625–634. Springer, 1999.
[68] Wojciech Rozowski, Tobias Kappé, Dexter Kozen, Todd Schmid, and Alexandra Silva. Probabilistic guarded KAT modulo bisimilarity: Completeness and complexity. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 261 of *LIPIcs*, pages 136:1–136:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
[69] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. From symmetric pattern-matching to quantum control. In *Foundations of Software Science and Computation Structures FOSSACS*, pages 348–364. Springer, 2018.
[70] Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.
[71] Peter Selinger and Benoît Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University Press, 1 edition, 2009.
[72] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1):17–64, 1996.
[73] Dario Stein and Sam Staton. Compositional semantics for probabilistic programs with exact conditioning. In *Proc. 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021.
[74] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.
[75] Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.
[76] Takeshi Tsukada and Kazuyuki Asada. Linear-algebraic models of linear logic as categories of modules over Σ-semirings. In Christel Baier and Dana Fisman, editors, *Proc. 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 60:1–60:13. ACM, 2022.
[77] Lionel Vaux. The algebraic lambda calculus. *Mathematical Structures in Computer Science*, 19(5):1029–1059, 2009.