Resource Transition Systems and Full Abstraction for Linear Higher-Order Effectful Programs

Ugo Dal Lago
University of Bologna, Italy
INRIA Sophia Antipolis, France

Francesco Gavazzo
University of Bologna, Italy
INRIA Sophia Antipolis, France

Abstract

We investigate program equivalence for linear higher-order (sequential) languages endowed with primitives for computational effects. More specifically, we study operationally-based notions of program equivalence for a *linear* λ-calculus with *explicit copying* and *algebraic effects à la* Plotkin and Power. Such a calculus makes explicit the interaction between copying and linearity, which are *intensional* aspects of computation, with effects, which are, instead, *extensional*. We review some of the notions of equivalences for linear calculi proposed in the literature and show their limitations when applied to effectful calculi where copying is a first-class citizen. We then introduce *resource transition systems*, namely transition systems whose states are built over tuples of programs representing the available resources, as an operational semantics accounting for both intensional and extensional interactive behaviours of programs. Our main result is a *sound and complete* characterization of contextual equivalence as *trace equivalence* defined on top of resource transition systems.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases algebraic effects, linearity, program equivalence, full abstraction

Related Version http://www.cs.unibo.it/ dallago/resbranch.pdf

Acknowledgements The authors are supported by the ERC Consolidator Grant DLV-818616 DIAPASoN.

1 Introduction

This work aims to study operationally-based equivalences for higher-order sequential programming languages enjoying three main features, which we are going to explain: *algebraic effects*, *linearity*, and *explicit copying*.

Algebraic Effects Since the early days of programming language semantics, the study of computational effects, i.e. those aspects of computations that go beyond the pure process of computing, has been of paramount importance. Starting with the seminal work by Moggi [49, 50], modelling and understanding computational effects in terms of monads [43] has been a standard practice in the denotational semantics of higher-order sequential languages. More recently, Plotkin and Power [60, 57, 58] have extended the analysis of computational effects in terms of monads to *operational semantics*, introducing the theory of *algebraic effects*. Accordingly, computational effects are produced by effect-triggering operations whose behaviour is, in essence, algebraic. Examples of such operations are nondeterministic and probabilistic choices, primitives for I/O, primitives for reading and writing from a global store, and many others. The operational analysis of computational effects in terms of algebraic operations also gave new insights not only on the operational semantics of effectful programming languages but also on their theories of equality, this way leading to the development of, e.g., effectful logical relations [36, 12], effectful applicative and normal form/open bisimulation [21, 19], and logic-based equivalences [67, 46].

Linearity and Copying The analysis of effectful computations in terms of monads and algebraic effects is, in its very essence, *extensional*: ultimately, a program represents a function from inputs to monadic outputs. However, when reasoning about computational effects, also *intensional* aspects of programs may be relevant. In particular, *linearity* [34, 69, 8] (and its quantitative refinements [33, 32, 14, 4, 23]) has been recognised as a fundamental tool to reason about computational effects [28, 48], as witnessed by a number of programming languages, such as Clean [55], Rust [47], Granule [52], and Linear Haskell [9], which explicitly rely on linearity to structure and manage effects. Indeed, the interaction between linearity, copying, and computational effects deeply influences program equivalence: there are effectful programs that cannot be discriminated without allowing the environment to copy them, and thus program transformations which are *sound* if linearity is guaranteed, but *unsound* in presence of copying.

A simple, yet instructive example of such a transformation, which we will carefully examine in the next section, is given by distributivity of λ-abstraction over probabilistic choice operators: $\lambda x.(e \oplus f) \simeq (\lambda x.e) \oplus (\lambda x.f)$. This transformation is well-known to be unsound for 'classical' call-by-value probabilistic languages [16]. However, it is sound if the programs involved cannot be copied [27, 26]. What, instead, we expect to be unsound is the transformation $!(e \oplus f) \simeq !e \oplus !f$, where the operator ! (bang) is the usual linear logic exponential modality making terms under its scope copyable and erasable. It is thus natural to ask if, and to what extent, the aforementioned notions of effectful program equivalence can be extended to *linear* languages with *explicit copying*.

Our Contribution In this paper we introduce *resource transition systems* as an intensional, resource-sensitive operational semantics for linear languages with algebraic operations and explicit copying. Resource transition systems combine standard *extensional* properties of effectful computations with linearity and copying, whose nature is, instead, *intensional*. We model the former using monads—as one does for ordinary effectful semantics—and the latter by shifting from program-based transition systems to *tuple-based* transition systems, as one does in environmental bisimulation [62, 44]. Indeed, a resource transition system can be thought of as an ordinary transition system whose states are built over tuples of copyable programs and linear values representing the available resources produced by a program while interacting with the external environment. Another possible way to look at resource transition systems is as an interactive semantics defined on top of the so-called storage model [68]. We then define and study trace equivalence on resource transition systems. Our main result states that trace equivalence is *sound* and *complete* for contextual equivalence. To the best of the authors' knowledge, this is the first full abstraction result for a linear λ-calculus with arbitrary algebraic effects and explicit copying.

Outline This paper is structured as follows. After an informal introduction to program equivalence for effectful linear languages (Section 2), Section 3 recalls some background notions on monads and algebraic operations. Section 4 introduces our vehicle calculus and its operational semantics. Resource-sensitive resource transition systems and their associated notions of equivalence are given in Section 5. Due to space constraints, several details have been omitted. The interested reader can find them in the extended version of the present paper [20].

2 Effects, Linearity, and Program Equivalence

In this section, we give a gentle introduction to program equivalence in presence of linearity, explicit copying, and effects. In this work, we are concerned with *operationally-based* equivalences, example of those being contextual and CIU equivalences [51, 45], logical relations [61, 56, 66] and, bisimulation-based equivalences [1, 40, 41, 62]. Moreover, among operationally-based equivalences, we seek for lightweight ones, by which we mean equivalences which are as easy to use as possible (otherwise, contextual equivalence would be enough). Accordingly, we do not consider equivalences in the spirit of logical relations—which usually require heavy techniques such as biorthogonality [54] and step-indexing [3] when applied to calculi in which recursion is present, either at the level of types or at the level of terms. Instead, we focus on *first-order* equivalences [44], viz. notions of trace equivalence and bisimilarity.

Our running examples in this paper are the already mentioned distributivity of (lambda) abstraction and bang over (fair) probabilistic choice in probabilistic call-by-value λ-calculi [24, 18, 27]:

$$\lambda x.(e \oplus f) \simeq (\lambda x.e) \oplus (\lambda x.f) \qquad (\lambda\text{-dist})$$

$$!(e \oplus f) \simeq\, !e \oplus !f \qquad (!\text{-dist})$$

It is well-known [16] that in call-by-value probabilistic languages, lambda abstraction does not distribute over probabilistic choice. In a linear setting, however, we see that any resource-sensitive notion of program equivalence $\simeq$ should actually validate the equivalence (λ-dist) but not (!-dist). Why? Let us look at the transition systems describing the (interactive) behaviour (Figure 1) of the programs involved in (λ-dist), where we write $[\![e]\!]$ for the result of the evaluation of an expression e. One way to understand the failure of the equivalence (λ-dist)

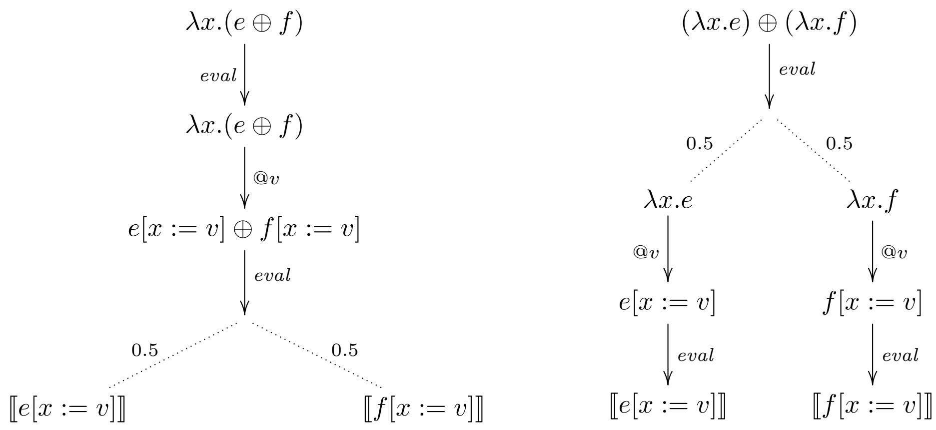


Figure 1 Interactive behaviour of $\lambda x.(e \oplus f)$ and $(\lambda x.e) \oplus (\lambda x.f)$

in *classical* (i.e. resource-agnostic) languages is that several notions of probabilistic program equivalence (such as probabilistic contextual equivalence [24], applicative bisimilarity [16, 24], and logical relations [13]) are sensitive to branching. However, sensitivity to branching does not quite feel like the crux of the failure of distributivity of abstraction over choice in classical languages. In fact, what we see is that $\lambda x.(e \oplus f)$ waits for an input, and then resolves the probabilistic choice. Dually, $(\lambda x.e) \oplus (\lambda x.f)$ first resolves the choice, and then waits for an input. As a consequence, if we evaluate these programs, $\lambda x.(e \oplus f)$ essentially does nothing, whereas $(\lambda x.e) \oplus (\lambda x.f)$ probabilistically chooses if continuing with either $\lambda x.e$ or $\lambda x.f$. At this point, there is a crucial difference between the programs obtained: $\lambda x.(e \oplus f)$ still has to resolve the probabilistic choice. If we were allowed to use it twice by passing it an argument v — this way resolving the choice twice — then we could observe a (probabilistic) behaviour different from both the one of $\lambda x.e$ and of $\lambda x.f$. Indeed, assuming $f[x := v]$ to diverge and $e[x := v]$ to converge (with probability 1), then, we would converge (to $e[x := v]$) with probability 0.25, in the former case, and with probability 0.5, in the latter case. To observe such a behaviour, however, it is crucial to *copy* $\lambda x.(e \oplus f)$. Otherwise, we could only interact with it by passing it an argument only *once*, this way validating (λ-dist).

Summing up, to invalidate (λ-dist) one has to be able to *copy* the results of the evaluation of the programs involved. This observation suggests that the deep reason why (λ-dist) fails relies on the copying capabilities of the calculus [63]. If the calculus at hand is linear (and thus offers no copying capability), we should then expect (λ-dist) to hold, while $!\lambda x.(e \oplus f) \simeq !(\lambda x.e) \oplus !(\lambda x.f)$ (and thus ultimately (!-dist)) to fail. This agrees with a recent result by Deng and Zhang [27, 26], who observed that if a calculus does not have copying capabilities, then contextual equivalence (which is *a fortiori* linear) validates (λ-dist). More generally, Deng and Zhang showed that *linear contextual equivalence*, i.e. contextual equivalence where contexts test their arguments linearly (viz. exactly once), coincides with *linear trace equivalence* in probabilistic languages.

But what about (!-dist)? Unfortunately, linear trace equivalence has been designed for linear languages *without* copying, only. Moreover, straightforward extensions of linear trace equivalence to languages with copying would actually validate (!-dist), trace equivalence being insensitive to branching. The situation does not change much if one looks at different forms of equivalence, such as Bierman's applicative bisimilarity [10]. Such equivalences usually invalidate (!-dist), but they all invalidate (λ-dist), too. We interpret all of this as a symptom of the lack of intensional structure in the aforementioned notions of equivalence. Ultimately, this can be traced back to the very operational semantics of the calculus, which is meant to be an abstract description of the input-output behaviour of programs, but gives no insight into their *intensional* structure, i.e. linearity and copying in our case [68].

We propose to overcome this deficiency by giving calculi a *resource-sensitive* operational semantics on top of which notions of program equivalence accounting for both intensional and extensional aspects of programs can be naturally defined. We do so by shifting from program-based transition systems to transition systems whose states are *tuples* $(\Gamma; \Delta)$, where Γ is a sequence of *non-linear* (hence copyable) programs and Δ is a sequence of *linear* values, as states. Accordingly, fixed a tuple $(\Gamma; \Delta)$ and a program e, we evaluate e, say obtaining a value v, and add v to the linear environment Δ, this way describing the *extensional* behaviour of the program. There are two *intensional* actions we can make on tuples. If Δ contains a value of the form $!e$, then we can remove $!e$ from Δ and add e to Γ. Dually, once we have a program e in Γ, we can decide to evaluate it—and thus to possibly produce a new linear value—*without* removing it from Γ, this way reflecting its non-linear nature. Finally, we can interact with a value $\lambda x.f$ by passing it an argument built using programs in Γ and values in Δ. As the latter are linear, we will then remove them from Δ.

We conclude this section by remarking that although here we have focused on probabilistic languages, a similar analysis can be made for languages exhibiting different kinds of effects, such as input-output behaviours as well as combinations of effects (e.g. probabilistic nondeterminism and global stores).

3 Preliminaries: Monads and Algebraic Effects

Starting with the seminal work by Moggi [49, 50], *monads* have become a standard formalism to model and study computational effects in higher-order sequential languages. Instead of working with monads, we opt for the equivalent notion of a *Kleisli triple* [43]. Additionally, instead of defining monads on arbitrary categories, we tacitly restrict our analysis to the category of sets and functions.

▶ **Definition 1.** *A* Kleisli triple *is triple* $(T, \eta, \gg\!=)$ *consisting of a map associating to any set* X *a set* $T(X)$*, a set-indexed family of functions* $\eta_X : X \to T(X)$*, and a map* $\gg\!=$*, called* bind*, associating to each function* $f : X \to T(Y)$ *a function* $\gg\!=f : T(X) \to T(Y)$*. Additionally, these data must obey the following laws, for* f *and* g *functions with appropriate (co)domains:*

$$\gg\!=\eta = id; \qquad \gg\!=f \circ \eta = f; \qquad \gg\!=g \circ \gg\!=f = \gg\!=(\gg\!=g \circ f).$$

Following standard practice, we write $m \gg\!= f$ *for* $\gg\!=f(m)$*.*

The computational interpretation behind Kleisli triples is the following: if A is a set (or type) of values, then $T(A)$ represent the set of computations returning values in A. Accordingly, for each set A there is a function $\eta_A : A \to T(A)$ that regards a value $a \in A$ as a trivial computation returning a (and producing no effect). The map η corresponds to the programming constructor **return**. Similarly, $\mu \gg\!= f$ is the *sequential composition* of a computation $\mu \in T(A)$ with a function $f : A \to T(B)$, and corresponds to the sequencing constructor **let** $x = -$ **in** $-$. Following this interpretation, we can read the identities in Definition 1 as stipulating that η indeed produces no effect, and that sequencing is associative.

Monads alone are not enough to produce actual effectful computations, as they only provide primitives to produce trivial effects (via the map η) and to (sequentially) compose them (via binding). For this reason, we endow monads T with (finitary) operations, i.e. with set-indexed families of functions $\mathbf{op}_X : T(X)^n \to T(X)$, where $n \in \mathbb{N}$ is the arity of the operation **op**.

▶ **Example 2.** Here are examples of monads modeling some of the computational effects discussed in Section 1. Further examples, such as global stores and exceptions can be found in, e.g., [49, 70].

1. We model possibly divergent computations using the maybe monad $\mathcal{M}(X) \triangleq X + \{\uparrow\}$. An element in $\mathcal{M}(A)$ is either an element $a \in A$ (meaning that we have a terminating computation returning a), or the element $\uparrow$ (meaning that the computation diverges). Given $a \in A$, the map η_A simply (left) injects a in $\mathcal{M}(A)$, whereas $\gg\!=f$ sends a terminating computation returning a to $f(a)$, and divergence to divergence:

$$\mathbf{inr}\ (a) \gg\!= f \triangleq f(a); \qquad \mathbf{inr}\ (\uparrow) \gg\!= f \triangleq \mathbf{inr}\ (\uparrow).$$

   As non-termination is an intrinsic feature of complete programming languages, we do not consider explicit operations to produce divergence.
2. We model probabilistic computations using the (discrete) subdistribution monad $\mathcal{D}$. Recall that a discrete subdistribution over a *countable* set X is a function $\mu : X \to [0, 1]$ such that $\sum_x \mu(x) \leq 1$. An element element $\mu \in \mathcal{D}(A)$ gives for any $a \in A$ the probability $\mu(a)$ of returning a. Notice that working with subdistribution we can easily model divergent computations [25]. Given $a \in A$, $\eta_A(a)$ is the Dirac distribution on a (mapping a to 1 and all other elements to 0), whereas for $\mu \in \mathcal{D}(A)$ and $f : A \to \mathcal{D}(B)$ we define $(\mu \gg\!= f)(b) \triangleq \sum_a \mu(a) \cdot f(a)(b)$. Finally, we generate probabilistic computations using a binary fair probabilistic choice operation $\oplus$ thus defined: $(\mu \oplus \nu)(x) \triangleq 0.5 \cdot \mu(x) + 0.5 \cdot \nu(x)$.

3. We model computations with output using the output monad $\mathcal{O}(X) \triangleq O^\infty \times (X + \{\uparrow\})$, where O^∞ is the set of finite and infinite strings over a fixed output alphabet O and $\uparrow$ is a special symbol denoting divergence. An element of $\mathcal{O}(A)$ is either a pair $(o, \mathbf{inl}\ a)$, with $a \in A$, or a pair $(o, \mathbf{inr}\ \uparrow)$. The former case denotes convergence to a outputting o (in which case o is a *finite* string), whereas the former denotes divergence outputting o (in which case o can be either finite or *infinite*). Given $a \in A$, the pair $(\varepsilon, \mathbf{inr}\ a)$ represents the trivial computation that returns a and outputs nothing (ε denotes the empty string). Further, sequential composition of computations is defined using string concatenation as follows, where $f(a) = (o', x)$:

$$(o, \mathbf{inr}\ \uparrow) \gg\!= f \triangleq (o, \mathbf{inr}\ \uparrow); \qquad (o, \mathbf{inl}\ a) \gg\!= f \triangleq (oo', x).$$

Finally, we produce outputs using (a O-indexed family of) unary operations $\mathbf{print}_c$ mapping (o, x) to (co, x).
4. We model computations with input using the input monad $\mathcal{I}(X) = \mu\alpha.(X + \{\uparrow\}) + \alpha^I$, where I is an input alphabet (for simplicity, we take $I = \{\mathit{true}, \mathit{false}\}$). An element in $\mathcal{I}(A)$ is a binary tree whose leaves are labeled either by elements in A or by the divergent symbol $\uparrow$. The trivial computation returning a is the single leaf labeled by a, whereas given a tree $t \in \mathcal{I}(A)$ and a map $f : A \to \mathcal{I}(B)$, the tree $t \gg\!= f$ is defined by replacing the leaves of t labeled by elements $a \in A$ with $f(a)$. Finally, we consider a binary input operation whereby $\mathbf{read}(t_{true}, t_{false})$ is the tree whose left child is t_{true} and whose right child is t_{false}.

We restrict our analysis to monads T preserving weak pullbacks, and thus preserving injections. As a consequence, if $i : A \hookrightarrow X$ is the subset inclusion map, then $T(i) : T(A) \hookrightarrow T(X)$ is an injection, which can be regarded as monadic inclusion. Intuitively, given an element $\mu \in T(X)$, we think about the *smallest* set $i : A \hookrightarrow X$ such that $\mu \in T(A)$ as the *support* of μ, and denote such a set as $\mathsf{supp}(\mu)$. Of course, in general the support of an element μ may not exist and therefore we restrict our analysis to monads coming with a notion of *countable* support.

▶ **Definition 3.** *We say that a monad is* countable *if for any set X and any element $\mu \in T(X)$, there exists the smallest countable set $i : Y \hookrightarrow X$, denoted by $\mathsf{supp}(\mu)$, such that $\mu \in T(Y)$ (i.e. there exists $\nu \in T(Y)$ such that $\mu = T(i)(\nu)$).*

All monads in Example 2 are countable (for instance, the subdistribution monad $\mathcal{D}$ is countable by definition). An example of a non-countable monad is the powerset monad $\mathcal{P}$. Nonetheless, since we will apply monads to countable sets only (viz. sets of λ-terms and variations thereof), we can regard $\mathcal{P}$ to be countable by taking its countable restriction.

3.1 Algebraic Effects

Following Example 2, let us consider a probabilistic program $e \triangleq E[e_1 \oplus e_2]$, where E is an evaluation context. The operational behaviour of e is to fairly choose $e_i \in \{e_1, e_2\}$, and then execute $E[e_i]$. That is, $E[e_1 \oplus e_2]$ evaluates to $E[e_1]$ (resp. $E[e_2]$) with probability 0.5. But that is exactly the behaviour of $E[e_1] \oplus E[e_2]$, so that we have the program equivalence $E[e_1 \oplus e_2] \equiv E[e_1] \oplus E[e_2]$. It does not take much to realize that a similar equivalence holds for all operations in Example 2. Semantically, operations justifying these equivalences are known as *algebraic operations* [58, 59].

▶ **Definition 4.** *An n-ary (set-indexed family of) operation(s)* $\mathbf{op}_X : T(X)^n \to T(X)$ *is an* algebraic operation *on T, if for all X, Y, $f : X \to T(Y)$, and $\mu_1, \ldots, \mu_n \in T(X)$, we have:*

$$(\mathbf{op}_X(\mu_1, \ldots, \mu_n)) \gg\!= f = \mathbf{op}_Y(\mu_1 \gg\!= f, \ldots, \mu_n \gg\!= f).$$

Using algebraic operations we can model a large class of effects, including those of Example 2, pure nondeterminism (using the powerset monad and set-theoretic union as binary nondeterminism choice), imperative computations (using the global states monad and operations for reading and updating stores), as well as combinations thereof [35].

3.2 Continuity

Another feature shared by all monads in Example 2 is that they all endow sets $T(X)$ with an ω-complete pointed partial order (ω-cppo, for short) structure making $\gg\!=$ strict, monotone, and continuous in both arguments, and algebraic operations monotone and continuous in all arguments. This property has been formalized in [21] as Σ-*continuity*.

▶ **Definition 5.** *Let T be a monad and Σ be a set of algebraic operations on T. We say that T is* Σ-continuous *if for any set X, $T(X)$ carries an ω-cppo structure such that $\gg\!=$ is strict, monotone, and continuous in both arguments, and (algebraic) operations in Σ are monotone and continuous in all arguments.*

▶ **Example 6.**
1. The maybe monad is $\emptyset$-continuous, with $\mathcal{M}(X)$ endowed with the flat order.
2. The subdistribution monad is $\{\oplus\}$-continuous, with subdistributions ordered pointwise (i.e. $\mu \leq \nu$ if and only if $\mu(x) \leq \nu(x)$, for any $x \in X$).
3. Let $\Sigma \triangleq \{\mathbf{print}_c \mid c \in O\}$. Then, the output monad is Σ-continuous, with $\mathcal{O}(A)$ endowed with the order: $(o, x) \sqsubseteq (o', x')$ if and only if either $x = \mathbf{inr} \uparrow$ and $o \sqsubseteq o'$ or $x = \mathbf{inl}\ a = x'$ and $o = o'$.
4. The input monad is $\{\mathbf{read}\}$-continuous with respect to the standard tree ordering.

4 A Linear Calculus with Algebraic Effects

In this section, we introduce a core *linear* call-by-value calculus with *algebraic operations* and *explicit copying* and its *resource-agnostic* operational semantics. The syntax of the calculus is parametric with respect to a signature Σ of operation symbols (notation $\mathbf{op} \in \Sigma$), whereas its dynamics relies on a Σ-continuous monad T, which we assume to be fixed.

4.1 Syntax

Our vehicle calculus is a linear refinement of fine-grain call-by-value [42], which we call $\Lambda^!$. The syntax of $\Lambda^!$ is given by two syntactic classes, *values* (notation $v, w, \ldots$) and *computations* (notation $e, f, \ldots$), which are thus defined:

$$v ::= x \mid \lambda x.e \mid\ !e$$
$$e ::= a \mid \mathbf{val}\ v \mid vv \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \mathbf{op}(e, \ldots, e) \mid \mathbf{let}\ !a = v\ \mathbf{in}\ e.$$

The letter x denotes a *linear* variable, and thus acts as a placeholder for a *value* which has to be used exactly once. Dually, the letter a denotes a *non-linear* variable, and thus acts as a placeholder for a *computation* which can be used *ad libitum*.

Following the fine-grain discipline, we require computations to be explicitly sequenced by means of the **let** $x = -$ **in** $-$ constructor. The latter comes in two flavors: in the first case, we deal with expressions of the form **let** $x = e$ **in** f, where x is a *linear* variable in f (and thus used once). The intuitive semantics of such an expression is to evaluate e, and then bind the result of the evaluation to x in f. As x is linear in f, the result of e cannot be copied. In the second case, we deal with expressions of the form **let** $!a = v$ **in** f, where a is a *non-linear* variable in f (and thus it can be used as will). As we are going to see, for such an expression to be meaningful, we need v to be a banged computation $!e$. The intuitive semantics of such an expression is thus to 'unbang' $!e$, and then bind e to a in f, this way enabling f to copy e at will.

When the distinction between values and computations is not relevant, we generically refer to *terms*, and denote them as $t, s, \ldots$. We adopt standard syntactic conventions as in [5]. In particular, we work with terms modulo renaming of bound variables, and denote by $t[x := v]$ (resp. $t[a := e]$) the result of capture-avoiding substitution of the value v (resp. computation e) for the variable x (resp. a) in t.

4.2 Statics

The syntax of $\Lambda^!$ allows one to write undesired programs, such as programs having runtime errors (e.g. $(!e)v$) and programs that should be forbidden by any reasonable type system (such as $(\mathbf{val}\ !e) \oplus (\mathbf{val}\ \lambda x.f)$). To overcome this problem, we follow [18] and endow $\Lambda^!$ with a simply-typed system with recursive types, using the system in, e.g., [6]. Types are defined by the following grammar:

$$\sigma ::= \mathsf{x} \;\big|\; !\sigma \;\big|\; \sigma \multimap \sigma \;\big|\; \mu\mathsf{x}.\sigma \multimap \sigma \;\big|\; \mu\mathsf{x}.!\sigma$$

where x is a type variable. Types are defined up to equality, as defined in Figure 2, where $\sigma[\tau/\mathsf{x}]$ denotes the substitution of τ for all the (free) occurrences of x in σ. In the third rule in Figure 2, we require ρ to be productive in x, meaning that each free occurrence of x in ρ is under the scope of either $\multimap$ or $!$.

$$\frac{}{\mu\mathsf{x}.\sigma \multimap \tau = \sigma[\mu\mathsf{x}.\sigma \multimap \tau/\mathsf{x}] \multimap \tau[\mu\mathsf{x}.\sigma \multimap \tau/\mathsf{x}]} \qquad \frac{}{\mu\mathsf{x}.!\sigma = !\sigma[\mu\mathsf{x}.!\sigma/\mathsf{x}]} \qquad \frac{\sigma = \rho[\sigma/\mathsf{x}] \quad \tau = \rho[\tau/\mathsf{x}]}{\sigma = \tau}$$

Figure 2 Type Equality

In order to define the collection of well-typed expressions, we consider sequents $\Sigma \mid \Omega \vdash^{\mathsf{v}} v : \sigma$ and $\Sigma \mid \Omega \vdash^{\Lambda} e : \sigma$, where Ω is a linear environment, i.e. a set without repetitions of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, and Σ is a *non-linear* environment, i.e. a set without repetitions of the form $a_1 : \tau_1, \ldots, a_n : \tau_n$. Rules for derivable sequents are given in Figure 3. We write $\mathcal{V}_\sigma$ and Λ_σ for the collection of closed values and computations of type σ, respectively. We write $\mathcal{V}$ and Λ when types are not relevant.

▶ Remark 7 (Notational Convention). In order to facilitate the communication of the main ideas behind this work and to lighten the (quite heavy) notation we will employ in the next sections, we avoid to mention types (and ignore them in the notation) whenever possible. Nonetheless, the reader should keep in mind that from now on we work with typable terms only. We refer to such an assumption as the *type assumption*.

$$\frac{}{\Sigma \mid x : \sigma \vdash^{\mathsf{v}} x : \sigma} \qquad \frac{}{a : \sigma, \Sigma \mid \emptyset \vdash^{\Lambda} a : \sigma} \qquad \frac{\Sigma \mid x : \sigma, \Omega \vdash^{\Lambda} e : \tau}{\Sigma \mid \Omega \vdash^{\mathsf{v}} \lambda x.e : \sigma \multimap \tau} \qquad \frac{\Sigma \mid \Omega \vdash^{\mathsf{v}} v : \sigma}{\Sigma \mid \Omega \vdash^{\Lambda} \mathbf{val}\ v : \sigma}$$

$$\frac{\Sigma \mid \Omega \vdash^{\mathsf{v}} v : \sigma \multimap \tau \quad \Sigma \mid \Omega' \vdash^{\mathsf{v}} w : \sigma}{\Sigma \mid \Omega, \Omega' \vdash^{\Lambda} vw : \tau} \qquad \frac{\Sigma \mid \emptyset \vdash^{\Lambda} e : \sigma}{\Sigma \mid \emptyset \vdash^{\mathsf{v}} !e : !\sigma} \qquad \frac{\Sigma \mid \Omega \vdash^{\mathsf{v}} v : !\sigma \quad \Sigma, a : \sigma \mid \Omega' \vdash^{\Lambda} e : \tau}{\Sigma \mid \Omega, \Omega' \vdash^{\Lambda} \mathbf{let}\ !a = v\ \mathbf{in}\ e : \tau}$$

$$\frac{\Sigma \mid \Omega \vdash^{\Lambda} e : \sigma \quad \Sigma \mid \Omega', x : \sigma \vdash^{\Lambda} f : \tau}{\Sigma \mid \Omega, \Omega' \vdash^{\Lambda} \mathbf{let}\ x = e\ \mathbf{in}\ f : \tau} \qquad \frac{\Sigma \mid \Omega \vdash^{\Lambda} e_1 : \sigma \quad \ldots \quad \Sigma \mid \Omega \vdash^{\Lambda} e_n : \sigma}{\Sigma \mid \Omega \vdash^{\Lambda} \mathbf{op}(e_1, \ldots, e_n) : \sigma}$$

Figure 3 Statics of $\Lambda^!$

4.3 Dynamics

The dynamic semantics of $\Lambda^!$ associates to any *closed computation* e of type σ a monadic element in $T(\mathcal{V}_\sigma)$. The dynamics of $\Lambda^!$ is defined in Figure 4 by means of an $\mathbb{N}$-indexed family of evaluation functions mapping a *closed* computation $e \in \Lambda_\sigma$ to an element $[\![e]\!]^{\Lambda}_k \in T(\mathcal{V}_\sigma)$, where we stipulate $[\![e]\!]^{\Lambda}_0 \triangleq \bot$. Since $([\![e]\!]^{\Lambda}_k)_{k \geq 0}$ forms an ω-chain in $T(\mathcal{V})$, we define $[\![e]\!]^{\Lambda} \triangleq \bigsqcup_{k \geq 0} [\![e]\!]^{\Lambda}_k$. Notice that thanks to the type assumption, we ignore programs causing runtime errors. Finally, we lift $[\![-]\!]^{\Lambda}$ to monadic computations, i.e. to elements $\xi \in T(\Lambda)$ by setting $[\![\xi]\!]^{\Lambda^*} \triangleq \xi \gg= (e \to [\![e]\!]^{\Lambda})$ (and similarity for $[\![-]\!]^{\Lambda}_k$).

$$\begin{aligned}
[\![\mathbf{val}\ v]\!]^{\Lambda}_{k+1} &\triangleq \eta(v) \\
[\![(\lambda x.e)v]\!]^{\Lambda}_{k+1} &\triangleq [\![e[x := v]]\!]^{\Lambda}_k \\
[\![\mathbf{let}\ x = e\ \mathbf{in}\ f]\!]^{\Lambda}_{k+1} &\triangleq [\![e]\!]^{\Lambda}_k \gg= (v \to [\![f[x := v]]\!]^{\Lambda}_k) \\
[\![\mathbf{let}\ !a = !e\ \mathbf{in}\ f]\!]^{\Lambda}_{k+1} &\triangleq [\![f[a := e]]\!]^{\Lambda}_k \\
[\![\mathbf{op}(e_1, \ldots, e_n)]\!]^{\Lambda}_{k+1} &\triangleq [\![\mathbf{op}]\!]([\![e_1]\!]^{\Lambda}_k, \ldots, [\![e_n]\!]^{\Lambda}_k)
\end{aligned}$$

Figure 4 Operational Semantics of $\Lambda^!$

4.4 Observational Equivalence

In order to compare $\Lambda^!$-terms, we introduce the notion of *contextual equivalence* [51]. To do so, we follow [67, 22] and postulate that once an observer executes a program, she can only observe the effects produced by the evaluation of the program. For instance, in a pure (resp. probabilistic) calculus one observes pure (resp. the probability of) convergence. Following this postulate, we define an observation function $\mathtt{obs}^{\Lambda^*} : T(\mathcal{V}) \to T(1)$ as $T(!_{\mathcal{V}})$, where $1 = \{*\}$ is the one-element set and $!_{\mathcal{V}} : \mathcal{V} \to 1$ is the terminal arrow. As a consequence, we see that $\mathtt{obs}^{\Lambda^*}$ is strict and continuous, so that we have, e.g., $\mathtt{obs}^{\Lambda^*}(\bigsqcup_k \xi_k) = \bigsqcup_k \mathtt{obs}^{\Lambda^*}(\xi_k)$.

▶ **Example 8.** Notice that $T(1)$ indeed describes the observations one usually works with in concrete calculi. For instance, $\mathcal{D}(1) \cong [0, 1]$, so that $\mathtt{obs}^{\Lambda^*}([\![e]\!])$ gives the probability of convergence of e, and $\mathcal{M}(1) \cong \{\bot, \top\}$, so that $\mathtt{obs}^{\Lambda^*}([\![e]\!]) = \top$ if and only if e converges.

In order to define contextual equivalence, we need to introduce the notion of a $\Lambda^!$-context. The latter is simply a $\Lambda^!$-term with a single *linear* hole $[-]$ acting as a placeholder for a computation (we regard a value v as the computation **val** v). We do not give an explicit definition of contexts, the latter being standard.

▶ **Definition 9.** *Define contextual equivalence* $\equiv^{\mathtt{ctx}}$ *as follows:*

$$v \equiv^{\mathtt{ctx}} w \iff \mathbf{val}\ v \equiv^{\mathtt{ctx}} \mathbf{val}\ w \qquad e \equiv^{\mathtt{ctx}} f \iff \forall C.\ \mathtt{obs}^{\Lambda^*}\ [\![C[e]]\!] = \mathtt{obs}^{\Lambda^*}\ [\![C[f]]\!].$$

The universal quantification over contexts guarantees $\equiv^{\mathtt{ctx}}$ to be a congruence relation. However, it also makes $\equiv^{\mathtt{ctx}}$ difficult to be used in practice. We overcome this deficiency by characterising contextual equivalence as a suitable notion of trace equivalence.

5 Resource-Sensitive Semantics and Program Equivalence

The operational semantics of Section 4.3 is *resource-agnostic*, meaning that linearity *de facto* plays no role in the definition of the dynamics of a program. To overcome this deficiency, we endow $\Lambda^!$ with a resource-sensitive operational semantics: we give the latter by means of a suitable transition systems, which we dub resource transition systems. *Resource transition systems* (RTSs, for short) provide an operational semantics for $\Lambda^!$-programs accounting for both their intensional and extensional behaviour. Those are defined as first-order transition systems in the spirit of [44], and generalise the Markov chains of [18].

5.1 Auxiliary Notions

In order to properly handle resources, it is useful to introduce some notation on sequences. Let S, S' be sequences over objects $s_1, s_2, \ldots$. Unless ambiguous, we denote the concatenation of S and S' as S, S'. Moreover, for $S = s_1, \ldots, s_k$ we denote by $|S| = k$ the length of S, and write $S[s]_i$, with $i \in \{1, \ldots, k+1\}$, for the sequence obtained by inserting s in S at position i, i.e. the sequence $s_1, \ldots, s_{i-1}, s, s_i, \ldots, s_k$ of length $k+1$. Given a sequence $S = s_1, \ldots, s_k$, we will form new sequences out of it by taking elements in S at given positions. If $\bar{c} = c_1, \ldots, c_n$ is a sequence with elements in $\{1, \ldots, k\}$ without repetitions, then we write $S_{\bar{c}}$ for the sequence $s_{c_1}, \ldots, s_{c_n}$, and $S \ominus \bar{c}$ for the sequence obtained from S by removing elements in positions $c_1, \ldots, c_n$. In order to preserve the order of S, we often consider sequences $\bar{c} = (c_1 < \cdots < c_n)$ with $c_i \in \{1, \ldots, k\}$. We call such sequences valid for S (although we should say valid for $|S|$).

5.1.0.1 System $\mathcal{K}$

The resource-sensitive operational semantics of $\Lambda^!$ is given by the RTS $\mathcal{K}$. Following [44], $\mathcal{K}$-states are defined as *configurations* $(\Gamma; \Theta)$, i.e. pairs of sequences of terms, where Γ is a (finite) sequence of (closed) computations and Θ is a (finite) sequence of (closed) terms in which only the last one need not be a value. To facilitate our analysis, we write $(\Gamma; \Delta; e)$ if $\Theta = \Delta, e$, with Δ finite sequence of closed values and $e \in \Lambda$. Otherwise, we write $(\Gamma; \Delta)$, with Δ as above.

In a configuration $(\Gamma; \Delta; e)$ (and similarly in $(\Gamma; \Delta)$), Γ represents the non-linear resources available, which are (closed) computations: the environment can freely duplicate and evaluate them, as well as use them *ad libitum* to build arguments to be passed as input to other programs. Once a resource in Γ has been used, it *remains* in Γ, this way reflecting its non-linear nature. Dually, Δ represents the linear resources available, which are closed values. Values in Δ being closed, they are either abstractions or banged computations. In the latter case, the environment can take a value $!e$, unbang it, and put e in Γ. In the former case, the environment can pass to a value $\lambda x.f$ an input argument made out of a context C (provided by the very environment) using values and computations in Γ, Δ. Since resources in Δ are linear, once they are used by C, they must be removed from Δ. Finally, the program e is the tested program. The environment can only evaluate it, possibly producing effects and values (linear resources). Once a linear resource v has been produced, it is put in Δ.

The calculus $\Lambda^!$ being typed, it is convenient to extend the notion of a type to configurations by defining a configuration type (notation $\alpha, \beta, \ldots$) as a pair of sequences $(\sigma_1, \ldots, \sigma_n; \tau_1, \ldots, \tau_m)$ of ordinary types. We say that a configuration $K = (\Gamma; \Theta)$ has type $\alpha = (\sigma_1, \ldots, \sigma_n; \tau_1, \ldots, \tau_m)$ (and write $\vdash K : \alpha$) if each computation e_i at position i in Γ has type σ_i, and each term t_i at position i in Θ has type τ_i.

Notice that configuration types almost completely describe the structure of configurations. However, they do not allow one to see whether the last argument in the second component Θ of a configuration $(\Gamma; \Theta)$ is a value (so that the type will be inhabitated by configurations of the form $(\Gamma; \Delta)$) or a computation (so that the type will be inhabitated by configurations of the form $(\Gamma; \Delta; e)$). To avoid this issue, we add a special label to the last type τ_m of the second component of a configuration type, this way specifying whether τ_m refers to a value or to a computation.

We denote by $\mathcal{C}_\alpha$ the collection of configurations of type α. Notice that if $K, L \in \mathcal{C}_\alpha$, then they have the same structure. In particular, terms in K and L at the same position have the same type and belong to the same syntactic class. As usual, following the type assumption, we will omit configuration types whenever possible.

States of $\mathcal{K}$ are thus (typable) configurations, whereas its dynamics is based on three kind of actions: *evaluation*, *duplication*, and *resource-based application*, which are *extensional*, *intensional*, and *mixed extensional-intensional* actions, respectively. Formally, we consider transitions from (typable) configurations, i.e. elements in $\bigcup_\alpha \mathcal{C}_\alpha$ to monadic configurations in $\bigcup_\alpha T(\mathcal{C}_\alpha)$, i.e. monadic configurations κ such that all configurations in the support of κ have the same type. This ensures that all configurations in $\mathsf{supp}(\kappa)$ can make the same actions. As usual, such a property follows by typing, hence by the type assumption. We now spell out the main ideas behind the dynamics of $\mathcal{K}$.

- Given a configuration $(\Gamma; \Delta; e)$, the environment simply evaluates e. That is, we have the transition:
$$(\Gamma; \Delta; e) \xrightarrow{eval} [\![e]\!] \gg= (v \to \eta(\Gamma; \Delta, v)).$$
- Given a configuration of the form $(\Gamma; \Delta[!e]_l)$, the environment adds e to the non-linear environment, and removes $!e$ from the linear one. We thus have the transition:
$$(\Gamma; \Delta[!e]_l) \xrightarrow{?_l} \eta(\Gamma, e; \Delta).$$
- In a configuration of the form $(\Gamma[e]_l; \Delta)$, the environment has the non-linear resource e at its disposal, which can be duplicated (and eventually evaluated via an *eval* action). We model such a behaviour as the following transition (notice that e is not removed from $\Gamma[e]_l$):
$$(\Gamma[e]_l; \Delta) \xrightarrow{!_l} \eta(\Gamma[e]_l; \Delta; e).$$
- For the last action, namely resource-based application, we consider open terms as playing the role of contexts. An open term is simply a term $\Sigma \mid \Omega \vdash t$. We refer to an open term $a_1, \ldots, a_n \mid x_1, \ldots, x_m \vdash t$ as a (n, m)-(value/computation) context, depending on whether t is a value or a computation. Given sequences $\Gamma = e_1, \ldots, e_n$, $\Delta = v_1, \ldots, v_m$, we write $t[\Gamma, \Delta]$ for the substitution of variables in t with the corresponding elements in Γ, Δ. As usual, following the type-assumption we assume types of variables to match types

of the substituted terms. Given sequences $\bar{\imath}$, $\bar{\jmath}$ of length n, m valid for Γ, Δ, respectively, we can build a new (closed) term out of Γ, Δ and a (n, m)-context t as $t[\Gamma_{\bar{\imath}}, \Delta_{\bar{\jmath}}]$. Since resources in Δ are linear, the construction of $t[\Gamma_{\bar{\imath}}, \Delta_{\bar{\jmath}}]$ affects Δ, this way leaving only resources $\Delta \ominus \bar{\jmath}$ available. We formalise this behaviour as the transition:

$$\frac{t\ (n,m)\text{-value context} \quad |\bar{\imath}| = n, |\bar{\jmath}| = m \quad \bar{\imath}, \bar{\jmath} \text{ valid for } \Gamma, \Delta}{(\Gamma; \Delta[\lambda x.f]_l) \xrightarrow{(\bar{\imath},\bar{\jmath},l,t)} \eta(\Gamma; \Delta \ominus \bar{\jmath}; f[x := t[\Gamma_{\bar{\imath}}, \Delta_{\bar{\jmath}}]])}$$

▶ **Definition 10.** *System $\mathcal{K}$ is the (resource) transition system having typable configurations as states, actions*

$$\{eval, ?_l, !_l, , (\bar{\imath}, \bar{\jmath}, l, t), \alpha \mid l \in \mathbb{N}, t\ (n,m)\text{-value context}, |\bar{\imath}| = n, |\bar{\jmath}| = m\}$$

where α ranges over configuration types, and dynamics defined by the transition rules in Figure 5, where we employ the notation of previous discussion.

$$(\Gamma; \Delta; e) \xrightarrow{eval} [\![e]\!] \gg\!= v \to \eta(\Gamma; \Delta, v) \qquad (\Gamma; \Delta[!e]_l) \xrightarrow{?_l} \eta(\Gamma, e; \Delta).$$

$$(\Gamma[e]_l; \Delta) \xrightarrow{!_l} \eta(\Gamma[e]_l; \Delta; e) \qquad (\Gamma; \Delta[\lambda x.f]_l) \xrightarrow{(\bar{\imath},\bar{\jmath},l,t)} \eta(\Gamma; \Delta \ominus \bar{\jmath}; f[x := t[\Gamma_{\bar{\imath}}, \Delta_{\bar{\jmath}}]])$$

Figure 5 Transition rules for $\mathcal{K}$

▶ Remark 11. Notice that given $K \in \mathcal{C}_\alpha$, K can always make a α-transition, this way making its type visible. Additionally, we see that the transition structure of $\mathcal{K}$ is *type-driven*. That is, given a configuration $K \in \mathcal{C}_\alpha$ and a $\mathcal{K}$-action ℓ, α and ℓ alone determine whether K can make an ℓ-transition. Moreover, if that is the case, then there is a unique κ such that $K \xrightarrow{\ell} \kappa$. Besides, $\kappa \in T(\mathcal{C}_\beta)$ for some configuration type β which is *uniquely* determined by ℓ and α. That is, there is a *partial* function b from configuration types and actions such that if $\mathsf{b}(\alpha, \ell)$ is defined and $K \in \mathcal{C}_\alpha$, then $K \xrightarrow{\ell} \kappa$ with $\kappa \in T(\mathcal{C}_{\mathsf{b}(\alpha,\ell)})$. From now on, we write $\mathsf{b}(\alpha, \ell) = \beta$ to mean that $\mathsf{b}(\alpha, \ell)$ is defined and equal β. As a consequence, we have the rule:

$$K \in \mathcal{C}_\alpha \wedge \mathsf{b}(\alpha, \ell) = \beta \implies \exists! \kappa \in T(\mathcal{C}_\beta).\ K \xrightarrow{\ell} \kappa.$$

Having defined system $\mathcal{K}$, there are at least two natural ways to compare its states. The first one is by means of *bisimilarity*, which can be defined in a standard way [21]. Unfortunately, bisimilarity being sensitive to branching, it is bound not to work well for our purposes, as already extensively discussed. The second natural way to compare $\mathcal{K}$-states is by means of *trace equivalence* which, contrary to bisimilarity, is not sensitive to branching, and thus qualifies as a suitable candidate program equivalence for our purposes.

▶ **Definition 12.** *A $\mathcal{K}$-trace (just trace) is a finite sequence of $\mathcal{K}$-actions. That is, a trace* t *is either the empty sequence (denoted by ε), or a sequence of the form $\ell \cdot \mathsf{u}$, where ℓ is a $\mathcal{K}$-action and* u *a trace.*

We are interested in observing the behaviour of $\mathcal{K}$-states on those traces that are coherent with their type. Therefore, given a $\mathcal{K}$-state K, we define the set $Tr(K)$ of its traces by stipulating that $\varepsilon \in Tr(K)$, for any K, and that $\ell \cdot \mathsf{u} \in Tr(K)$ whenever $K \xrightarrow{\ell} \kappa$, for some monadic configuration κ, and $\mathsf{u} \in Tr(L)$, for any $L \in \mathsf{supp}(\kappa)$. Notice that the latter clause is meaningful, since $Tr(K)$ is actually determined by the type of K (rather than by K itself), and if $K \xrightarrow{\ell} \kappa$, then all configurations in the support of κ have the same type.

Now, given a $\mathcal{K}$-state K, and a trace $\mathtt{t} \in Tr(K)$, the observable behaviour of K on $\mathtt{t}$ is the element in $T(1)$ computed using the map $\mathtt{st}$ thus defined:

$$\mathtt{st}(K, \varepsilon) \triangleq \eta(*); \qquad \mathtt{st}(K, \ell \cdot \mathtt{u}) \triangleq \kappa \gg\!= (L \to \mathtt{st}(L, \mathtt{u})) \text{ where } K \xrightarrow{\ell} \kappa.$$

▶ **Example 13.** Let us consider the (sub)distribution monad $\mathcal{D}$, and let K be a configuration. Recall that $\mathcal{D}(1) \cong [0,1]$, and notice that $\mathtt{st}(K, \varepsilon) = 1$. Suppose now $K \xrightarrow{eval} \sum_{i \in n} p_i \cdot L_i$. Then, we see that $\mathtt{st}(K, eval \cdot \mathtt{u}) = \sum_{i \in n} p_i \cdot \mathtt{st}(L_i, \mathtt{u}) \in [0,1]$, meaning that $\mathtt{st}(K, \mathtt{t})$ gives the probability that K passes the trace $\mathtt{t}$.

▶ **Definition 14.** *The relation $\simeq^{\mathcal{K}}_{\mathrm{Tr}}$ on $\mathcal{K}$-states is thus defined:*

$$K \simeq^{\mathcal{K}}_{\mathrm{Tr}} L \iff Tr(K) = Tr(L) \wedge \forall \mathtt{t} \in Tr(K).\ \mathtt{st}(K, \mathtt{t}) = \mathtt{st}(L, \mathtt{t})$$

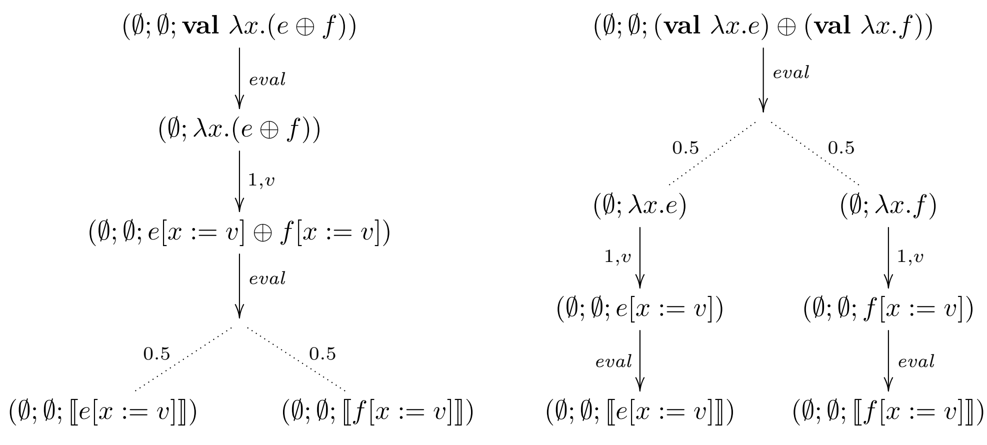
We extend the action of $\simeq^{\mathcal{K}}_{\mathrm{Tr}}$ to $\Lambda^!$-terms by regarding a computation e as the configuration $(\emptyset; \emptyset; e)$, and a value v as the computation **val** v. We denote the resulting notion $\simeq^{\Lambda}_{\mathrm{Tr}}$.

Having added $\simeq^{\mathcal{K}}_{\mathrm{Tr}}$ to our arsenal of operational techniques, it is time to investigate its structural properties and its relationship with contextual equivalence. Before doing so, however, we take a fresh look at our running example.

▶ **Example 15.** Let us use the machinery developed so far to review our introductory examples. First, we show

$$\mathbf{val}\ \lambda x.(e \oplus f) \simeq^{\Lambda}_{\mathrm{Tr}} (\mathbf{val}\ \lambda x.e) \oplus (\mathbf{val}\ \lambda x.f).$$

Let us call g the former program, and h the latter. To see that $g \simeq^{\Lambda}_{\mathrm{Tr}} h$, we simply observe that $Tr(\emptyset; \emptyset; g) = Tr(\emptyset; \emptyset; h)$ and that for any $\mathtt{t} \in Tr(g)$, the probability that $(\emptyset; \emptyset; g)$ passes $\mathtt{t}$ coincides with the one of $(\emptyset; \emptyset; h)$. All of this can be easily observed by inspecting the following transition systems.



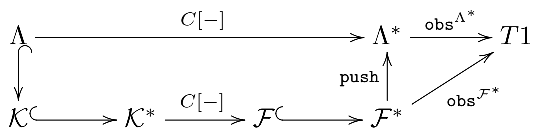
In light of Theorem 17, we can then conclude $g \equiv^{\mathrm{ctx}} h$. Next, we prove that such an equivalence is only linear: **val** $!(e \oplus f) \not\equiv^{\mathrm{ctx}}$ (**val** $!e) \oplus$ (**val** $!f$). For that, it is sufficient to instantiate e and f as the identity program **val** ($\lambda x.$**val** x) and the purely divergent program Ω, respectively, and to take the context C defined as **let** $x = [-]$ **in let** $!a = x$ **in** $(a; a;$ **val** $v)$, where v is closed value, and $e; f$ denotes trivial sequencing. Indeed, what C does is to evaluate its input and then test the result thus obtained *twice*.

5.2 Full Abstraction of Trace Equivalence

In this section, we outline the proof of *full abstraction* of trace equivalence for contextual equivalence. Our proof of full abstraction builds upon the technique given by Deng and Zhang [27] and Crubillé and Dal Lago [18] to prove similar full abstraction results for trace equivalences and metrics, respectively. Due to the large amount of technicalities, the full proof of full abstraction of trace equivalence goes beyond the scope of this paper, so that here we only outline its main points (see [20] for details). Let us begin by showing that trace equivalence is *sound* for contextual equivalence.

▶ **Proposition 16.** $\simeq^{\Lambda}_{\mathtt{Tr}} \subseteq \equiv^{\mathtt{ctx}}$.

To prove Proposition 16, we have to show that if $e \simeq^{\Lambda}_{\mathtt{Tr}} f$, then we have $\mathtt{obs}^{\Lambda^*}[\![C[e]]\!]^{\Lambda} = \mathtt{obs}^{\Lambda^*}[\![C[e]]\!]^{\Lambda}$, for any context C. Our proof proceeds by progressively building systems with increasingly more complex state spaces, but with finer dynamics. We summarise our strategy in the following diagram.



Since $\simeq^{\Lambda}_{\mathtt{Tr}}$ is defined in terms of $\simeq^{\mathcal{K}}_{\mathtt{Tr}}$, we consider configurations—$\mathcal{K}$-states—and contexts for them, where a context for a $\mathcal{K}$-state K is just a standard multiple-holes context whose holes have to be filled with with terms in K. The first step of our strategy is the *determinization* of $\mathcal{K}$. This is achieved by lifting the state space of $\mathcal{K}$ from configurations to monadic configurations. The dynamics of $\mathcal{K}$ is then lifted relying on the (strong) monad structure of T in a standard way [22]. We call the resulting system $\mathcal{K}^*$. The advantage of working with $\mathcal{K}^*$ is that $\mathcal{K}^*$-bisimilarity and $\mathcal{K}^*$-trace equivalence coincide, $\mathcal{K}^*$ being deterministic. In general, most of the transition systems we rely on can be ultimately described as systems $\mathcal{S} = (X, \delta)$ made of a state space X and a dynamics $\delta : X \to T(X)^A$, for some set A of actions. The determinization of $\mathcal{S}$, which we usually denote by $\mathcal{S}^*$, has $T(X)$ as state space and dynamics $\delta^* : T(X) \to T(X)^A$ defined as the strong Kleisli extension of δ (modulo (un)currying).

Having determinized $\mathcal{K}$, we reach a situation where we have to study the computational behaviour of a monadic configuration κ — i.e. a $\mathcal{K}^*$-state — and a context C for the configurations in the support of κ. To do so, we build a further system, called $\mathcal{F}$, whose states are pairs $C : \kappa$ made of a monadic configuration κ and a context C for it. The dynamics of $\mathcal{F}$ is given by an evaluation function which, when applied to a $\mathcal{F}$-state $C : \kappa$, gives the same result of evaluating the *monadic computation* $C[\kappa] \in T(\Lambda)$, where $C[\kappa] = \kappa \gg\!= (K \to \eta(C[K]))$. Such a dynamics explicitly separates the computational steps acting on C only from those making C and κ interact. This feature is crucial, as it shows that any interaction between C and κ corresponds to a $\mathcal{K}^*$-action, so that equivalent $\mathcal{K}^*$-states will have the same $\mathcal{F}$-dynamics when paired with the same context. That gives us a finer analysis of the computational behaviour of the compound monadic computation $C[\kappa]$, and ultimately of a compound computation $C[e]$. As we did for $\mathcal{K}$, it is actually convenient to determinise $\mathcal{F}$. We call the resulting system $\mathcal{F}^*$. Finally, from $\mathcal{F}^*$ we can come back to $T(\Lambda)$ using the map $\mathtt{push} : \mathcal{F}^* \to T(\Lambda)$ defined by $\mathtt{push}(\xi) \triangleq \xi \gg\!= (C : \kappa \mapsto C[\kappa])$. We summarize the systems introduced so far in the following table.

System	$\mathcal{K}$	$\mathcal{K}^*$	$\mathcal{F}$	$\mathcal{F}^*$
States	Configurations K	Monadic configurations κ	Pairs $C : \kappa$	Monadic pairs
Dynamics	Definition 10	Kleisli lifting of $\mathcal{K}$	$[\![C[\kappa]]\!]^*$	Kleisli lifting of $\mathcal{F}$

What remains to be clarified is how relations between computations can be transformed into relations on the aforementioned systems. The answer to this question is given by the following *lax*[1] commutative diagram:

$$
\begin{array}{ccccccccccc}
\Lambda & \hookrightarrow & \mathcal{K} & \hookrightarrow & \mathcal{K}^* & \xrightarrow{C[-]} & \mathcal{F} & \hookrightarrow & \mathcal{F}^* & \xrightarrow{\mathtt{obs}^{\mathcal{F}^*}} & T1 \\
{\scriptstyle \simeq^{\Lambda}_{\mathrm{Tr}}}\downarrow & & {\scriptstyle \simeq^{\mathcal{K}}_{\mathrm{Tr}}}\downarrow & & {\scriptstyle \simeq^{\mathcal{K}^*}_{\mathrm{Tr}}}\downarrow & & {\scriptstyle \mathtt{C}(\simeq^{\mathcal{K}^*}_{\mathrm{Tr}})}\downarrow & & {\scriptstyle \mathtt{BC}(\simeq^{\mathcal{K}^*}_{\mathrm{Tr}})}\downarrow & & {\scriptstyle =}\downarrow \\
\Lambda & \hookrightarrow & \mathcal{K} & \hookrightarrow & \mathcal{K}^* & \xrightarrow[C[-]]{} & \mathcal{F} & \hookrightarrow & \mathcal{F}^* & \xrightarrow[\mathtt{obs}^{\mathcal{F}^*}]{} & T1
\end{array}
$$

Here, $\mathtt{C}(R)$ denotes the contextual closure of R, whereas $\mathtt{B}(R)$ is the Barr extension of R [7, 38]. Finally, the map $\mathtt{obs}^{\mathcal{F}^*}$ is obtained postcomposing the observation map `obs` with `push`. Let us now move to full abstraction.

▶ **Theorem 17.** $\equiv^{\mathrm{ctx}} = \simeq^{\Lambda}_{\mathrm{Tr}}$.

To prove Theorem 17 it is sufficient to show $\equiv^{\mathrm{ctx}} \subseteq \simeq^{\Lambda}_{\mathrm{Tr}}$. The latter is proved by noticing that any $\mathcal{K}$-action can be encoded as a context. The encoding of $\mathcal{K}$-actions as contexts is essentially the same one of the one given by Crubillé and Dal Lago [18].

6 Conclusion and Future Work

In this paper, we have introduced resource transition systems as an operational account of both intensional and extensional behaviours of linear effectful programs with explicit copying. On top of resource transition systems, we have defined trace equivalence and showed that the latter is fully abstract for contextual equivalence.

Although the present paper focuses on linearity (and effects), the authors believe that resource transition systems can be extended to deal with finer notions of context dependence such as *structural coeffects* [53, 29, 14, 52]. To do so, one should modify resource transition systems by considering sequences of terms indexed by elements of a resource algebra (the latter being a preordered semiring), and let transitions update resources. Thus, for instance, from a sequence $(\Gamma, \langle e\rangle_{r+1}, \Delta)$, meaning that e is available according to the resource $r+1$, we have a transition to $(\Gamma, \langle e\rangle_r, \Delta; e)$. The authors also believe that resource transition systems can be used to generalise Crubillé and Dal Lago probabilistic program metric to arbitrary algebraic effects. To do so, one would simply replace ordinary relations with relations taking values over quantales [30, 31]. In the same direction, it would be interesting to study whether resource transition systems give fully abstract equivalences in presence of *continuous*, rather than discrete, probability (applicative bisimilarity, for instance, has been proved to be sound but not fully abstract on higher-order calculi with sampling from continuous distributions [39]).

Finally, as a long term future work, the authors would like to study whether the ideas presented in this paper can be adapted to deal with quantum languages [64, 65], where the interaction between linearity and effects plays a central role. In fact, although we have not discussed tensor product types (which play a crucial role in a quantum setting), it is not hard to see that resource transition systems can be extended to deal with such types [17].

[1] Each square gives a set-theoretic inclusion. For instance, the leftmost square states that $\simeq^{\Lambda}_{\mathrm{Tr}} \subseteq \simeq^{\mathcal{K}}_{\mathrm{Tr}}$.

6.1 Related Work

This is not the first work on operationally-based notions of program equivalence for linear calculi. In particular, notions of equivalences have been defined by means of logical relations by Bierman, Pitts, and Russo [11], of applicative bisimilarity by Bierman [10] and Crole[2] [15], of trace equivalence by Deng and Zhang [27, 26], as well as of a number of possible worlds-indexed equivalences (e.g. [2, 37]). As already remarked, one of the advantages of resource transition systems (and their associated trace equivalence) compared, e.g., with logical relations, is that they they provide a *first-order* account of program equality.

Among first-order notions of program equivalence, Bierman's applicative bisimilarity plays a prominent role. The latter is a lightweight extensional equivalence extending Abramsky's applicative bisimilarity [1] to a *pure* linear λ-calculus with explicit copying. Bierman's applicative bisimilarity can be readily extended to calculi with algebraic effects along the lines of [21], this way obtaining a notion of equivalence invalidating (!-dist). However, such a notion of bisimilarity stipulates that two programs $!e$ and $!f$ are bisimilar if and only if e and f are, this way making bisimilarity insensitive to linearity, and thus invalidating (λ-dist) as well.[3]

Deng and Zhang's linear trace equivalence has been designed to study the interaction of linearity and (both pure and probabilistic) nondeterminism. The latter equivalence, in fact, validates (λ-dist). However, linear trace equivalence does not deal with (explicit) copying: even worse, natural extensions of such notions to languages with copying result in equivalences validating (!-dist). Crubillé and Dal Lago [18] solved that problem by introducing a tuple-based applicative bisimilarity for a calculus with probabilistic nondeterminism and explicit copying. Our notion of a resource transition system can be seen as a generalisation of the Markov chain underlying tuple based applicative bisimilarity to arbitrary algebraic effects.

References

1. Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.
2. Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundam. Informaticae*, 77(4):397–449, 2007.
3. Andrew W. Appel and Daddiv A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.
4. Robert Atkey. Syntax and semantics of quantitative type theory. In *Proc. of LICS 2018*, pages 56–65, 2018.
5. Hendrik P. Barendregt. *The lambda calculus: its syntax and semantics.* Studies in logic and the foundations of mathematics. North-Holland, 1984.
6. Hendrik P. Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types.* Perspectives in logic. Cambridge University Press, 2013.
7. Michael Barr. Relational algebras. *Lect. Notes Math.*, 137:39–55, 1970.
8. Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Proc. of LICS 1996*, pages 420–431, 1996.
9. Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *PACMPL*, 2(POPL):5:1–5:29, 2018.

---

[2] Crole's applicative bisimilarity, however, does not deal with copying.
[3] Besides, notice that bisimilarity being sensitive to branching, it naturally invalidates (λ-dist).

10 Gavin M. Bierman. Program equivalence in a linear functional language. *J. Funct. Program.*, 10(2):167–190, 2000.

11 Gavin M. Bierman, Andrew M. Pitts, and Claudio V. Russo. Operational properties of lily, a polymorphic linear lambda calculus with recursion. *Electr. Notes Theor. Comput. Sci.*, 41(3):70–88, 2000.

12 Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *PACMPL*, 2(POPL):8:1–8:30, 2018.

13 Ales Bizjak and Lars Birkedal. Step-indexed logical relations for probability. In *Proc. of FOSSACS 2015*, pages 279–294, 2015.

14 Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In *Proc. of ESOP 2014*, pages 351–370, 2014.

15 Roy L. Crole. Completeness of bisimilarity for contextual equivalence in linear theories. *Logic Journal of the IGPL*, 9(1):27–51, 2001.

16 Raphaëlle Crubillé and Ugo Dal Lago. On probabilistic applicative bisimulation and call-by-value lambda-calculi. In *Proc. of ESOP 2014*, pages 209–228, 2014.

17 Raphaëlle Crubillé and Ugo Dal Lago. Metric reasoning about lambda-terms: The affine case. In *Proc. of LICS 2015*, pages 633–644, 2015.

18 Raphaëlle Crubillé and Ugo Dal Lago. Metric reasoning about lambda-terms: The general case. In *Proc. of ESOP 2017*, pages 341–367, 2017.

19 Ugo Dal Lago and Francesco Gavazzo. Effectful normal form bisimulation. In *Proc. of ESOP 2019*, pages 263–292, 2019.

20 Ugo Dal Lago and Francesco Gavazzo. Resource transition systems and full abstraction for linear higher-order effectful programs (extended version). 2021. URL: `http://www.cs.unibo.it/~dallago/resbranch.pdf`.

21 Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. Effectful applicative bisimilarity: Monads, relators, and howe's method. In *Proc. of LICS 2017*, pages 1–12, 2017.

22 Ugo Dal Lago, Francesco Gavazzo, and Ryo Tanaka. Effectful applicative similarity for call-by-name lambda calculi. *Theor. Comput. Sci.*, 813:234–247, 2020.

23 Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. In *Proc. of TLCA 2009*, pages 80–94, 2009.

24 Ugo Dal Lago, Davide Sangiorgi, and Michele Alberti. On coinductive equivalences for higher-order probabilistic functional programs. In *Proc. of POPL 2014*, pages 297–308, 2014.

25 Ugo Dal Lago and Margherita Zorzi. Probabilistic operational semantics for the lambda calculus. *RAIRO - Theor. Inf. and Applic.*, 46(3):413–450, 2012.

26 Yuxin Deng and Yuan Feng. Bisimulations for probabilistic linear lambda calculi. In *Proc. of TASE 2017*, pages 1–8, 2017.

27 Yuxin Deng and Yu Zhang. Program equivalence in linear contexts. *Theor. Comput. Sci.*, 585:71–90, 2015.

28 Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. Linear-use CPS translations in the enriched effect calculus. *Logical Methods in Computer Science*, 8(4), 2012.

29 Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. Combining effects and coeffects via grading. In *Proc. of ICFP 2016*, pages 476–489, 2016.

30 Francesco Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. In *Proc. of LICS 2018*, pages 452–461, 2018.

31 Francesco Gavazzo. *Coinductive Equivalences and Metrics for Higher-order Languages with Algebraic Effects*. PhD thesis, University of Bologna, Italy, 2019. URL: `http://amsdottorato.unibo.it/9075/`.

32 Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Proc. of ESOP 2014*, pages 331–350, 2014.

33 J-Y. Girard, A. Scedrov, and P.J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97:1–66, 1992.

34 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

35 Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.

36 Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *Proc. of LICS 2010*, pages 209–218. IEEE Computer Society, 2010.

37 Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Proc. of POPL 2015*, pages 17–30, 2015.

38 Alexander Kurz and Jiri Velebil. Relation lifting, a survey. *J. Log. Algebr. Meth. Program.*, 85(4):475–499, 2016.

39 Ugo Dal Lago and Francesco Gavazzo. On bisimilarity in lambda calculi with continuous probabilistic choice. In *Proc. of MFPS 2019*, pages 121–141, 2019.

40 Søren B. Lassen. Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. *Electr. Notes Theor. Comput. Sci.*, 20:346–374, 1999.

41 Søren B. Lassen. Eager normal form bisimulation. In *Proceedings of LICS 2005*, pages 345–354, 2005.

42 Paul B. Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003.

43 Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

44 Jean-Marie Madiot, Damien Pous, and Davide Sangiorgi. Bisimulations up-to: Beyond first-order transition systems. In *Proc. of CONCUR 2014*, pages 93–108, 2014.

45 Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *J. Funct. Program.*, 1(3):287–327, 1991.

46 Cristina Matache and Sam Staton. A sound and complete logic for algebraic effects. In *Proc. of FOSSACS 2019*, pages 382–399, 2019.

47 Nicholas D. Matsakis and Felix S. Klock II. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104, 2014.

48 Rasmus Ejlers Møgelberg and Sam Staton. Linear usage of state. *Logical Methods in Computer Science*, 10(1), 2014.

49 Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of LICS 1989*, pages 14–23. IEEE Computer Society, 1989.

50 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

51 J. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1969.

52 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP):110:1–110:30, 2019.

53 Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In *Proc. of ICFP 2014*, pages 123–135, 2014.

54 Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.

55 Marinus J. Plasmeijer. CLEAN: a programming environment based on term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 2:215–221, 1995.

56 Gordon Plotkin. Lambda-definability and logical relations. Technical Report SAI-RM-4, School of A.I., University of Edinburgh, 1973.

57 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In *Proc. of FOSSACS 2001*, pages 1–24, 2001.

58 Gordon D. Plotkin and John Power. Semantics for algebraic operations. *Electr. Notes Theor. Comput. Sci.*, 45:332–345, 2001.

59 Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Proc. of FOSSACS 2002*, pages 342–356, 2002.

60 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

61 John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

62 Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011.

63 Davide Sangiorgi and Valeria Vignudelli. Environmental bisimulations for probabilistic higher-order languages. In *Proceedings of POPL 2016*, pages 595–607, 2016.

64 Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.

65 Peter Selinger and Benoît Valiron. A linear-non-linear model for a computational call-by-value lambda calculus (extended abstract). In *Proc. of FOSSACS 2008*, pages 81–96, 2008.

66 Kurt Sieber. Reasoning about sequential functions via logical relations. In Johnstone P. T. Fourman, M. P. and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 258–269. Cambridge University Press, 1992.

67 Alex Simpson and Niels Voorneveld. Behavioural equivalence via modalities for algebraic effects. In *Proc. of ESOP 2018*, pages 300–326, 2018.

68 David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theor. Comput. Sci.*, 227(1-2):231–248, 1999.

69 Philip Wadler. Linear types can change the world! In *Programming concepts and methods, 1990*, page 561, 1990.

70 Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, pages 24–52, 1995.