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

Ugo Dal Lago Francesco Gavazzo

February 27, 2021

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 behaviors of programs. Our main result is a *sound and complete* characterization of contextual equivalence as *trace equivalence* defined on top of resource transition systems.

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 [34, 11], effectful applicative and normal form/open bisimulation [18, 37], and logic-based equivalences [68, 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* [31, 70] (and its quantitative refinements [30, 29, 13, 4, 20]) has been recognised as a fundamental tool to reason about computational effects [25, 48], as witnessed by a number of programming languages, such as Clean [55], Rust [47], Granule [52], and Linear Haskell [8], 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 [15]. However, it is sound if the programs involved cannot be copied [24, 23]. 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 [63, 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 [69]. We then define and study trace equivalence on resource transition systems. Our main result states 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 5 introduces our vehicle calculus and its operational semantics. Resource-sensitive resource transition systems and their associated notions of equivalence are given in Section 6.

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 equivalence [51, 45], logical relations [62, 56, 67] and, bisimulation-based equivalences [1, 38, 39, 63]. 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 [21, 17, 24]:

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

It is well-known [15] 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). One way to under-

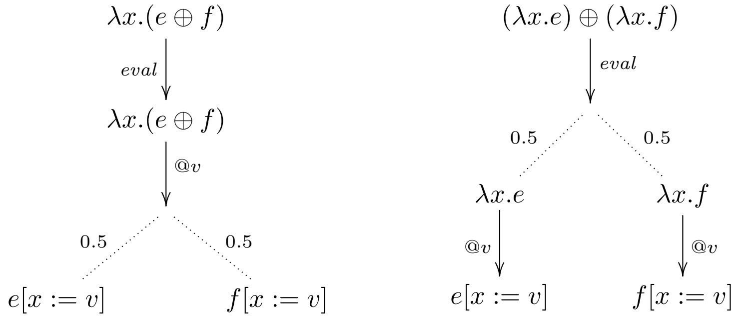

Figure 1: Interactive behaviour of $\lambda x.(e \oplus f)$ and $(\lambda x.e) \oplus (\lambda x.f)$

stand the failure of the equivalence (λ-dist) in *classical* (i.e. resource-agnostic) languages is that several notions of probabilistic program equivalence (such as probabilistic contextual equivalence [21], applicative bisimilarity [15, 21], and logical relations [12]) are sensitive to branching. However, sensitivity to branching does not quite feel like the crux of the failure of 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 pass it an argument, say v, *twice*— 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 [64]. 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 [24, 23], 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 [9]. Such equivalences usually invalidates (!-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 [69].

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].

Definition 3.1 *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:*

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

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 3.1 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 3.2 *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, 71].*

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:*

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

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 [22]. 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* $f(a) = (o', x)$*:*

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

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}*.*

3.1 Algebraic Effects

Following Example 3.2, let us consider a probabilistic program $e \triangleq E[e_1 \oplus e_2]$, where E is an evaluation context. The operational behavior of e is to fairly choose a $e_i \in \{e_1, e_2\}$, 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 behavior 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 3.2. Semantically, operations justifying these equivalences are known as *algebraic operations* [58, 59].

Definition 3.3 *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 3.2, 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 [32].

3.2 Continuity

Another feature shared by all monads in Example 3.2 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 [18] as Σ-*continuity.*

Definition 3.4 *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 3.5 *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 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 Generic Effects

Representing effectful computations as monadic objects has the major advantage of providing semantical information on the effects performed. However, it also has the drawback of lacking a clear distinction between the *effects* produced by a computation and the possible *results* returned. Nonetheless, since effects can only be produced by (algebraic) operations, we can always decouple the effects produced during a computation from its possible results. This is done relying on the notion of a *generic effect* [60], which we introduce by means of an example.

Example 4.1 *Recall that we model probabilistic computations using the subdistribution monad $\mathcal{D}$. When working with (discrete) subdistribution, it is oftentimes convenient to employ syntactic representations of such (sub)distributions, known as* formal sums. *A formal sum (over a set X) is an expression of the form $\sum_{i\in I} p_i; x_i$, where I is a countable set, $p_i \in [0, 1]$, $x_i \in X$, and $\sum_i p_i \leq 1$. The notation $\sum_{i\in I} p_i; x_i$ is meant to recall the semantic counterpart of formal sums, namely subdistributions. However, we should keep in mind that formal sums are purely syntactical expressions. For instance, $\frac{1}{2}; x + \frac{1}{2}; x$ and $1; x$ are two distinct formal sums, although they both denote the Dirac distribution on x. More generally, there is an interpretation function $\mathcal{I}$ mapping each formal sum $\sum_{i\in I} p_i; x_i$ to a subdistribution μ on X defined as $\mu(x) \triangleq \sum_{x_i = x} p_i$. Examining a bit more carefully a formal sum $\sum_{i\in I} p_i; x_i$, we see that the latter consists of an I-indexed sequence $\mathbf{p} = \langle p_i\rangle_{i\in I}$ of elements in $[0, 1]$ together with an I-indexed sequence $\mathbf{x} = \langle x_i\rangle_{i\in I}$ of elements in X. Therefore, a formal sum is just a pair of sequences $(\mathbf{p}, \mathbf{x}) \in [0, 1]^I \times X^I$ such that $\sum_i p_i \leq 1$. But the latter requirement means precisely that $\mathbf{p}$ is actually a subdistribution on I (the one mapping i to p_i). Therefore, we see that formal sums are just elements in $\mathcal{D}(I) \times X^I$. Putting these observations together, we see that for any $\mu \in \mathcal{D}(X)$, there exists a countable set I and an element $\phi \in \mathcal{D}(I) \times X^I$ such that $\mathcal{I}(\phi) = \mu$. As a consequence, stipulating two formal sums $\phi_1, \phi_2 \in \mathcal{D}(I) \times X^I$ to be equal (notation $\phi_1 =_{\mathcal{I}} \phi_2$) if $\mathcal{I}(\phi_1) = \mathcal{I}(\phi_2)$, then we see that $\mathcal{D}(X)$ is isomorphic to the quotient set $(\bigcup_I \mathcal{D}(I) \times X^I)/ =_{\mathcal{I}}$, where I ranges over countable sets.*

Can we generalize Example 4.1 to arbitrary Σ-continuous monads? If monads are countable, [61, 42, 33] (as they are all the monads considered in this work), the answer to this question is in the affirmative. First, let us observe that since the set I in Example 4.1 is countable, we can replace it with an enumeration of its elements. That is, we replace I with sets $\boldsymbol{n}$, where $n \in \mathbb{N}^\infty \triangleq \mathbb{N} \cup \{\omega\}$ and $\boldsymbol{n} \triangleq \{1, \dots, n\}$ if $n \neq \omega$, and $\boldsymbol{n} \triangleq \mathbb{N}_{\geq 1}$, if $n = \omega$.

Theorem 4.1 ([61, 42, 33]) *Let T be countable monad. Then, for any countable set X, all elements in $T(X)$ can be (non-uniquely) presented as elements in*

$$\bigcup_{n \in \mathbb{N}^\infty} T(\boldsymbol{n}) \times X^n$$

Moreover, the map $\mathcal{I} : \bigcup_{n \in \mathbb{N}^\infty} T(\boldsymbol{n}) \times X^n \to T(X)$ mapping $(\gamma, \mathbf{x})$ to $\mathbf{x}^\dagger(\gamma)$ is surjection whose kernel $=_\mathcal{I}$ gives an isomorphism $T(X) \cong \bigcup_{n \in \mathbb{N}^\infty} T(\boldsymbol{n})/{=_\mathcal{I}}$.

Given a pair $(\gamma, \mathbf{x}) \in \bigcup_{n \in \mathbb{N}^\infty} T(\boldsymbol{n})/{=_\mathcal{I}}$, we think about γ as the effect produced during a computation, and about $\mathbf{x}$ as the possible values returned. Elements in $T(\boldsymbol{n})$ are called *generic effects* [60], whereas we refer to the set $\{x_i\}_{i \in \boldsymbol{n}}$ associated to $\mathbf{x}$ as the *support* of ξ.

By Theorem 4.1, we can represent any monadic element as an equivalence class of a pair $(\gamma, \mathbf{x})$. Working with such pairs[1] allows us to simplify proofs. Moreover, elements in $T(\boldsymbol{n})$ form an *operand* [40, **?**]. In particular, they come with a notion of composition that mapping all generic effects $\gamma \in T(\boldsymbol{n})$, $\alpha_1 \in T(\boldsymbol{m_1}), \dots, \alpha_1 \in T(\boldsymbol{m_1})$, to a generic effect $\gamma \circ (\alpha_1, \dots, \alpha_n) \in T(\boldsymbol{l})$, where $l \triangleq \sum_{i \in \boldsymbol{n}} m_i$. Additionally, operands comes with a diagrammatic syntax whereby we write

$$\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}} x_i$$

for the pair $(\gamma, \langle x_i \rangle_{i \in \boldsymbol{n}}) \in T(\boldsymbol{n}) \times X^n$. In a diagram $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{——}}} x_i$, the letter i ranges over elements in $\boldsymbol{n}$ and to each i it is associated the corresponding element x_i. That is, the horizontal bar with subscript i and target x_i stands for the function $i \mapsto x_i$.

Example 4.2 1. *Consider the maybe monad $\mathcal{M}$. We present an object $\mu \in \mathcal{M}(X)$ as a pair in $\mathcal{M}(\boldsymbol{n}) \times X^n$, for some $n \in \mathbb{N}^\infty$. Since $\mathcal{M}(\boldsymbol{n}) \times X^n = (\boldsymbol{n} + \{\uparrow\}) \times X^n$, μ is (presented as) either a pair $(k, \langle x_i \rangle_{i \in \boldsymbol{n}})$ or a pair $(\uparrow, \langle x_i \rangle_{i \in \boldsymbol{n}})$. The former corresponds to the case of convergence to x_k, whereas the latter to divergence. In particular, if ξ is the result of evaluating a λ-term, then we will actually have $n = 1$ (if the term converges) or $n = 0$ (if the term diverges). If $n = 1$, we obtain pairs of the form $(1, \langle x \rangle))$, which we write as $\boxed{\downarrow}\text{———} x$. If $n = 0$, then we can only have the pair $(\perp, \langle\rangle)$, where $\langle\rangle$ is the empty sequence. We write such a pair as $\boxed{\uparrow}$.*

2. *Consider the output monad $\mathcal{O}$. We present an object $\mu \in \mathcal{O}(X)$ as a pair in $\mathcal{O}(\boldsymbol{n}) \times X^n = O^\infty \times (\boldsymbol{n} + \{\uparrow\}) \times X^n$, for some $n \in \mathbb{N}^\infty$. Therefore, μ is presented as either a triple $(o, \uparrow, \langle x_i \rangle_{i \in \boldsymbol{n}})$, or as a triple $(o, k, \langle x_i \rangle_{i \in \boldsymbol{n}})$. The former case means that we have divergence, and that the string o is outputted, whereas the latter case means that we converge to x_k, and that the string o is outputted. As before, if μ is the result of evaluating a (λ-)term, we will have either $n = 1$ (if the term converges) or $n = 0$ (if the term diverges). If $n = 1$, we have triples of the form $(o, 1, x))$ which we write as $\boxed{o}\text{———} x$. If $n = 0$, the we can only have triples of the form $(o, \perp, \langle\rangle)$, which we write as $\boxed{(o, \uparrow)}$.*

3. *The case for the subdistribution monad goes exactly as in Example 4.1. We can present a formal sum $(\langle p_i \rangle_{i \in \boldsymbol{n}}, \langle x_i \rangle_{i \in \boldsymbol{n}})$ as $\boxed{\langle p_i \rangle_{i \in \boldsymbol{n}}}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}} x_i$.*

[1] For simplicity, we work with pairs $(\gamma, \mathbf{x})$ rather than with their equivalence classes: it is a straightforward exercise to check that all definitions we give relying on such pairs do not actually depend on the specific choice of the pair, so that they extend to elements in $\bigcup_{n \in \mathbb{N}^\infty} T(\boldsymbol{n})/{=_\mathcal{I}}$.

Using the diagrammatic syntax, we present the composition of pairs $(\gamma, \mathbf{x})$, $(\alpha_1, \mathbf{y}_1), \ldots, (\alpha_n \mathbf{y}_n)$ (with $\mathbf{x}$ of length n and each $\mathbf{y}_i$ of length m_i) as:

$$\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\boxed{\alpha_i}\overset{\boldsymbol{m_i}}{\underset{j}{\text{———}}} y_j$$

Notice that associativity of composition is built-in the diagrammatic notation and that the latter also manages index dependencies. In fact, in the above diagram we see that $i \in \boldsymbol{n}$ and $j \in \boldsymbol{m_i}$. Moreover, by reading from the right to the left we recover index dependencies: since $j \in \boldsymbol{m_i}$, it depends on $i \in \boldsymbol{n}$. There is a trivial generic effect $\eta \in T(\mathbf{1})$ corresponding to the unit of T which behaves as a neutral element for composition:

$$\boxed{\eta}\text{———}\xi = \xi; \qquad\qquad \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\boxed{\eta}\text{———}x_i = \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}x_i.$$

Moreover, given a function $f : X \to T(Y)$ and $\mu \in T(X)$, we see that if μ is presented as $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}x_i$, then $\mu \gg\!= f$ is presented as $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}f(x_i)$. Finally, we recall a well-known result by Plotkin and Power stating that algebraic operations and generic effects are equivalent notions.

Proposition 4.2 ([60]) *There is a one-to-one correspondence between generic effects in $T(\boldsymbol{n})$ and n-ary algebraic operations on T.*

In light of Proposition 4.2, if we present objects $\mu_i \in T(X)$ as ξ_i, then we write $\boxed{\mathbf{op}}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\xi_i$ for the presentation of $\mathbf{op}(\mu_1, \ldots, \mu_n)$. Notice that if μ_i is presented as ξ_i, then both $\mathbf{op}(\mu_1, \ldots, \mu_n) \gg\!= f$ and $\mathbf{op}(\mu_1 \gg\!= f, \ldots, \mu_n \gg\!= f)$ are presented as $\boxed{\mathbf{op}}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}f(\xi_i)$. That is, the defining identity of algebraic operations (Definition 3.3) is built-in the notation.

Order-Theoretic Properties Since we deal with Σ-continuous monads, we can transfer order-theoretic properties of T to the diagrammatic notation by stipulating that a diagram ξ is below a digram φ (notation $\xi \sqsubseteq \varphi$) if so are the elements presented by those diagrams. In particular, there is a bottom effect $\boxed{\bot} \in T(\mathbf{0})$ corresponding to the bottom element of T satisfying the law $\boxed{\bot} \sqsubseteq \xi$, for any diagram ξ. Additionally,we have the following monotonicity laws, where $f : X \to T(Z)$:

$$(\forall i \in \boldsymbol{n}.\ \xi_i \sqsubseteq \varphi_i) \implies \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\xi_i \sqsubseteq \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\varphi_i$$

$$\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}x_i \sqsubseteq \boxed{\alpha}\overset{\boldsymbol{m}}{\underset{j}{\text{———}}}y_j \implies \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}f(x_i) \sqsubseteq \boxed{\alpha}\overset{\boldsymbol{m}}{\underset{j}{\text{———}}}f(y_j)$$

5 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.

5.1 Syntax

Our vehicle calculus is a linear refinement of fine-grain call-by-value [41], 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:

$$v ::= x \;\big|\; \lambda x.e \;\big|\; !e$$

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

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 $\mathbf{let}\ x = -\ \mathbf{in}\ -$ constructor. The latter comes in two flavors: in the first case, we deal with expressions of the form $\mathbf{let}\ x = e\ \mathbf{in}\ f$, 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 $\mathbf{let}\ !a = v\ \mathbf{in}\ f$, 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 as 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.

5.2 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 [17] 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 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.

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

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 5.1 (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.

5.3 Dynamics

The dynamic semantics of $\Lambda^!$ associates to any *closed computation* e of type σ a monadic element in $T(\mathcal{V}_\sigma)$. Such a dynamics is defined relying on Felleisen's evaluation semantics

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

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

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

Figure 3: Statics of $\Lambda^!$

[26]. Accordingly, we define *evaluation contexts* and *redexes* by the following grammars

$$E ::= [-] \;\Big|\; \mathbf{let}\ x = E\ \mathbf{in}\ e$$

$$r ::= (\lambda x.e)v \;\Big|\; \mathbf{let}\ x = (\mathbf{val}\ v)\ \mathbf{in}\ e \;\Big|\; \mathbf{let}\ !a = !e\ \mathbf{in}\ f \;\Big|\; \mathbf{op}(e_1, \dots, e_n)$$

where $[-]$ acts as a placeholder for a computation. The *pure* reduction relation $\mapsto$ is thus defined:

$$(\lambda x.e)v \mapsto e[x := v] \qquad \mathbf{let}\ x = (\mathbf{val}\ v)\ \mathbf{in}\ e \mapsto e[x := v] \qquad \mathbf{let}\ !a = !e\ \mathbf{in}\ f \mapsto f[a := e]$$

Notice that $\mapsto$ is deterministic and that no (side) effect is produced when performing according $\mapsto$-reductions. We denote by r' the unique term such that $r \mapsto r'$. 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 $[\![-]\!]^{\Lambda}_k$).

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

Figure 4: Operational Semantics of $\Lambda^!$

5.4 Observational Equivalence

In order to compare $\Lambda^!$-terms, we introduce the notion of *contextual equivalence* [51]. To do so, we follow [**?**, 68] 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 $\mathtt{obs}^{\Lambda^*} : T(\Lambda) \to T(1)$ 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 5.2 *Notice that $T(1)$ indeed describes the observations one usually work with in concrete calculi. For instance, $\mathcal{D}(1) \cong [0,1]$, so that $\mathtt{obs}^{\Lambda^*}(\llbracket e \rrbracket)$ gives the probability of convergence of e, and $\mathcal{M}(1) \cong \{\bot, \top\}$, so that $\mathtt{obs}^{\Lambda^*}(\llbracket e \rrbracket) = \top$ 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 5.3 *Define contextual equivalence $\equiv^{\mathsf{ctx}}$ as follows:*

$$\begin{aligned} e \equiv^{\mathsf{ctx}} f &\iff \forall C.\ \mathtt{obs}^{\Lambda^*}\ \llbracket C[e] \rrbracket = \mathtt{obs}^{\Lambda^*}\ \llbracket C[f] \rrbracket \\ v \equiv^{\mathsf{ctx}} w &\iff \mathbf{val}\ v \equiv^{\mathsf{ctx}} \mathbf{val}\ w. \end{aligned}$$

As usual, we can easily show $\equiv^{\mathsf{ctx}}$ to be a congruence relation.

Remark 5.4 *Thinking to a context C as an experiment, we see that C being forced to use its hole $[-]$* linearly*, we are allowed to experiment with a program e more than once only if $e \in \Lambda_{!\sigma}$.*

Contextual equivalence is a powerful notion to discriminate between programs, but are not well-suited to establish equivalences between them. We overcome this deficiency by characterising contextual equivalence as a notion of effectful environmental trace equivalence.

6 Resource-sensitive Semantics and Program Equivalence

The operational semantics of Section 5.3 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 [17].

6.1 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 $|S|$).

Concatenation and insertion Unless ambiguous, we will denote the concatenation of Σ and Σ' as Σ, Σ'. Moreover, for $\Sigma = S_1, \ldots, S_k$ we denote by $|\Sigma| = k$ the length of Σ, and write $\Sigma[S]_p$, with $p \in \{1, \ldots, k+1\}$, for the sequence obtained by inserting S in Σ at position p, i.e. the sequence $S_1, \ldots, S_{p-1}, S, S_p, \ldots, S_k$ of length $k+1$. Notice that $\Sigma[S]_1 = S, \Sigma$ and $\Sigma[S]_{k+1} = \Sigma, S$.

Subsequences and subtraction Oftentimes, given a sequence $\Sigma = S_1, \ldots, S_k$, we will form new sequences out of it by taking elements in Σ at given positions. If $\bar{p} = (p_1, \ldots, p_l)$ is a sequence with elements in $\{1, \ldots, k\}$ without repetitions, then we write $\Sigma_{\bar{p}}$ for the sequence $S_{p_1}, \ldots, S_{p_l}$, and $\Sigma \ominus \bar{p}$ for the sequence obtained from Σ by removing elements in position $p_1, \ldots, p_l$. Observe that if $\varphi : \{1, \ldots, l\} \to \{1, \ldots, l\}$ is a permutation, then $\Sigma \ominus \bar{p} = \Sigma \ominus \varphi(\bar{p})$ (meaning that the operation $\ominus$ has actually a set—rather than a sequence—as left operand), and that we have the identity $\Sigma[S]_p \ominus p = \Sigma$.

Ordered subsequences When building the sequence $\Sigma \ominus \bar{p}$, for Σ and p as above, we preserve the order of Σ. This is not the case for $\Sigma_{\bar{p}}$. To avoid such behaviours, we can consider sequences $\bar{p} = (p_1 < \cdots < p_k)$ with $p_i \in \{1, \ldots, n\}$. We call such sequences valid for Σ (although we should say valid for $|\Sigma|$: indeed, if $\bar{p}$ is valid for Σ, then it is also valid for any Σ' such that $|\Sigma'| = |\Sigma|$).

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. In order to facilitate our analysis, we introduce the following notation. If Θ ends with a closed computation e, then we write $(\Gamma; \Delta; e)$ with Δ finite sequence of closed values (and $\Theta = \Delta, e$). Otherwise, we write $(\Gamma; \Delta)$, with Δ as above. 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 similarity 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 pass 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 $!e$, unbanged 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. Mimicking previous notational conventions, we write $(\sigma_1, \ldots, \sigma_n; \tau_1, \ldots, \tau_m)$ if all τ_is refer to values, and $(\sigma_1, \ldots, \sigma_n; \tau_1, \ldots, \tau_m; \rho)$ if all τ_is refers to values and ρ to a computation.

Formally:

$$\frac{\vdash^{\Lambda} e_1, \ldots, e_n : \sigma_1, \ldots, \sigma_n \quad \vdash^{\mathsf{V}} v_1, \ldots, v_m : \tau_1, \ldots, \tau_m}{\vdash (e_1, \ldots, e_n; v_1, \ldots, v_m) : (\sigma_1, \ldots, \sigma_n; \tau_1, \ldots, \tau_m)}$$

$$\frac{\vdash^{\Lambda} e_1, \ldots, e_n : \sigma_1, \ldots, \sigma_n \quad \vdash^{\mathsf{V}} v_1, \ldots, v_m : \tau_1, \ldots, \tau_m \quad \vdash^{\Lambda} e : \rho}{\vdash (e_1, \ldots, e_n; v_1, \ldots, v_m; e) : (\sigma_1, \ldots, \sigma_n; \tau_1, \ldots, \tau_m; \rho)}$$

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 6.1 *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.

Remark 6.2 *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*

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

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

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

by ℓ and α. That is, there is a partial *function* $\mathtt{b}$ *from configuration types and actions such that if* $\mathtt{b}(\alpha,\ell)$ *is defined and* $K \in \mathcal{C}_\alpha$*, then* $K \xrightarrow{\ell} \kappa$ *with* $\kappa \in T(\mathcal{C}_{\mathtt{b}(\alpha,\ell)})$*. As a consequence, in order to know whether a configuration* K *of type* α *can make a* ℓ*-transition, it is sufficient to check if* $\mathtt{b}(\alpha,\ell)$ *is defined. From now on, we write* $\mathtt{b}(\alpha,\ell)=\beta$ *to mean that* $\mathtt{b}(\alpha,\ell)$ *is defined and equal* β*. As a consequence, we have the rule:*

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

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 [18]. 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 6.3 *A* $\mathcal{K}$*-trace (just trace) is a finite sequence of* $\mathcal{K}$*-actions. That is, a trace* $\mathtt{t}$ *is either the empty sequence (denoted by* ε*), or a sequence of the form* $\ell \cdot \mathtt{u}$*, where* ℓ *is a* $\mathcal{K}$*-action and* $\mathtt{u}$ *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 $\ell \cdot \mathtt{u} \in Tr(K)$ whenever $K \xrightarrow{\ell} \kappa$, for some monadic configuration κ, and $\mathtt{u} \in Tr(L)$, 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 6.4 *It is a straightforward exercise to prove that on the powerset monad* $\mathtt{st}$ *gives the usual notion of 'passing a trace'. 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 6.5 *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 6.6 *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.

$$\begin{tikzcd}[column sep=small]
 & (\emptyset;\emptyset;\mathbf{val}\ \lambda x.(e \oplus f)) \arrow[d, "eval"] & \\
 & (\emptyset;\lambda x.(e \oplus f)) \arrow[d, "{1,v}"] & \\
 & (\emptyset;\emptyset;e[x:=v] \oplus f[x:=v]) \arrow[d, "eval"] & \\
 & \cdot \arrow[dl, dotted, no head, "0.5"'] \arrow[dr, dotted, no head, "0.5"] & \\
(\emptyset;\emptyset;\llbracket e[x:=v] \rrbracket) & & (\emptyset;\emptyset;\llbracket f[x:=v] \rrbracket)
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=small]
 & (\emptyset;\emptyset;(\mathbf{val}\ \lambda x.e) \oplus (\mathbf{val}\ \lambda x.f)) \arrow[d, "eval"] & \\
 & \cdot \arrow[dl, dotted, no head, "0.5"'] \arrow[dr, dotted, no head, "0.5"] & \\
(\emptyset;\lambda x.e) \arrow[d, "{1,v}"'] & & (\emptyset;\lambda x.f) \arrow[d, "{1,v}"] \\
(\emptyset;\emptyset;e[x:=v]) \arrow[d, "eval"'] & & (\emptyset;\emptyset;f[x:=v]) \arrow[d, "eval"] \\
(\emptyset;\emptyset;\llbracket e[x:=v] \rrbracket) & & (\emptyset;\emptyset;\llbracket f[x:=v] \rrbracket)
\end{tikzcd}$$

In light of Theorem 7.8, we can then conclude $g \equiv^{\mathrm{ctx}} h$. Next, we prove that such an equivalence is only linear: $\mathbf{val}\ !(e \oplus f) \not\equiv^{\mathrm{ctx}} (\mathbf{val}\ !e) \oplus (\mathbf{val}\ !f)$. *For that, it is sufficient to instantiate e and f as the identity program* $\mathbf{val}\ (\lambda x.\mathbf{val}\ x)$ *and the purely divergent program Ω, respectively, and to take the context C defined as* $\mathbf{let}\ x = [-]\ \mathbf{in}\ \mathbf{let}\ !a = x\ \mathbf{in}\ (a; a; \mathbf{val}\ v)$, *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.

7 Trace Equivalence: Soundness and Completeness

In this section, we prove the main result of this work, namely *full abstraction* of trace equivalence for contextual equivalence: $\simeq^{\Lambda}_{\mathrm{Tr}} = \equiv^{\mathrm{ctx}}$. That $\equiv^{\mathrm{ctx}}$ is included in $\simeq^{\Lambda}_{\mathrm{Tr}}$ (*completeness*) does not come with much of a surprise. In fact, it is easy to realise that all $\mathcal{K}$-actions (and thus traces) can be implemented by suitable contexts [17]. Proving that $\simeq^{\Lambda}_{\mathrm{Tr}}$ is included in $\equiv^{\mathrm{ctx}}$ (i.e. *soundness*) is, however, more challenging. Our proof builds upon the technique given by Deng and Zhang [24] and Crubillé and Dal Lago [17] to prove similar full abstraction results for trace equivalences and metrics, respectively. Due to the large amount of technicalities, before entering into the technical details of the proof of soundness of trace equivalence, it is instructive to outline the main points of such a proof.

Soundness of trace equivalence means that the inclusion $\simeq^{\Lambda}_{\mathrm{Tr}} \subseteq \equiv^{\mathrm{ctx}}$ holds. To prove that, we have to show that if $e \simeq^{\Lambda}_{\mathrm{Tr}} f$, then we have $\mathtt{obs}^{\Lambda^*} \llbracket C[e] \rrbracket^{\Lambda} = \mathtt{obs}^{\Lambda^*} \llbracket C[e] \rrbracket^{\Lambda}$, 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.

$$\begin{tikzcd}
\Lambda \arrow[rrr, "{C[-]}"] \arrow[d, hook] & & & \Lambda^* \arrow[r, "\mathtt{obs}^{\Lambda^*}"] & T1 \\
\mathcal{K} \arrow[r, hook] & \mathcal{K}^* \arrow[r, "{C[-]}"] & \mathcal{F} \arrow[r, hook] & \mathcal{F}^* \arrow[u, "\mathtt{push}"] \arrow[ur, "\mathtt{obs}^{\mathcal{F}^*}"'] &
\end{tikzcd}$$

Since $\simeq^{\Lambda}_{\mathrm{Tr}}$ is defined in terms of $\simeq^{\mathcal{K}}_{\mathrm{Tr}}$, 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 [19]. 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 for $\kappa = \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{---}}}K_i$, we define $C[\kappa]$ pointwise as $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{---}}}C[K_i]$. 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 $\mathtt{push}\left(\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{---}}}C : \kappa\right) = \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{---}}}C[\kappa]$. We summarize the systems introduce 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 6.1	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*[2] commutative diagram:

$$\begin{tikzcd}
\Lambda \arrow[r, hook] \arrow[d, "\simeq^{\Lambda}_{\mathtt{Tr}}"'] & \mathcal{K} \arrow[r, hook] \arrow[d, "\simeq^{\mathcal{K}}_{\mathtt{Tr}}"'] & \mathcal{K}^* \arrow[r, "{C[-]}"] \arrow[d, "\simeq^{\mathcal{K}^*}_{\mathtt{Tr}}"'] & \mathcal{F} \arrow[r, hook] \arrow[d, "\mathtt{C}(\simeq^{\mathcal{K}^*}_{\mathtt{Tr}})"'] & \mathcal{F}^* \arrow[r, "\mathtt{obs}^{\mathcal{F}^*}"] \arrow[d, "\mathtt{BC}(\simeq^{\mathcal{K}^*}_{\mathtt{Tr}})"'] & T1 \arrow[d, "="'] \\
\Lambda \arrow[r, hook] & \mathcal{K} \arrow[r, hook] & \mathcal{K}^* \arrow[r, "{C[-]}"'] & \mathcal{F} \arrow[r, hook] & \mathcal{F}^* \arrow[r, "\mathtt{obs}^{\mathcal{F}^*}"'] & T1
\end{tikzcd}$$

Here, $\mathtt{C}(R)$ denotes the contextual closure of R, whereas $\mathtt{B}(R)$ is the Barr extension of R [7, 36]. Finally, the map $\mathtt{obs}^{\mathcal{F}^*}$ is obtained postcomposing the observation map $\mathtt{obs}$ with $\mathtt{push}$.

7.1 Determinisation: From $\mathcal{K}$ to $\mathcal{K}^*$

The first step of our strategy is the determinisation of $\mathcal{K}$. We do so by taking advantage of Remark 6.2 and working with a transition system whose states are monadic configurations in $\bigcup_\alpha T(\mathcal{C}_\alpha)$. Without much of a surprise, we extend the notion of a type to monadic configurations by stipulating κ has type α if and only if $\kappa \in T(\mathcal{C}_\alpha)$.

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

Definition 7.1 *System $\mathcal{K}^*$ has elements in $\bigcup_\alpha T(\mathcal{C}_\alpha)$ as states, $\mathcal{K}$-actions as actions, and transition structure thus defined, where $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}K_i \in T(\mathcal{C}_\alpha)$:*

$$\frac{\mathtt{b}(\alpha,\ell)\text{ defined} \qquad \forall i \in n.\ K_i \xrightarrow{\ell} \kappa_i}{\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}K_i \overset{\ell}{\Longrightarrow} \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\kappa_i}$$

Notice that $\mathcal{K}^*$ is indeed a deterministic system and that, by Remark 6.2, the transition structure of $\mathcal{K}^*$ is well-defined. For suppose $\kappa = \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}K_i$ is a $\mathcal{K}^*$-state, and thus an element in $T(\mathcal{C}_\alpha)$ for some configuration type α, and let ℓ be an action such that $\mathtt{b}(\alpha,\ell) = \beta$. Then, for any $i \in n$, we have $K_i \xrightarrow{\ell} \kappa_i$, for some $\kappa_i \in T(\mathcal{C}_\beta)$. As a consequence, we see that $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\kappa_i \in T(\mathcal{C}_\beta)$. Notice also that for $\boxed{\perp_\alpha} \in T(\mathcal{C}_\alpha)$, we have $\boxed{\perp_\alpha} \overset{a}{\Longrightarrow} \boxed{\perp_\beta}$, for any β and action a such that $\mathtt{b}(\alpha,a) = \beta$.

We define a notion of trace equivalence for $\mathcal{K}^*$ pretty much as we did for $\mathcal{K}$. The action sets of $\mathcal{K}$ and $\mathcal{K}^*$ being the same, the set of traces of $\mathcal{K}$ and $\mathcal{K}^*$ are the same as well. Moreover, given a $\mathcal{K}^*$-state κ, the set $Tr(\kappa)$ is defined in the obvious way. Finally, we rely on the structure $(T\{*\}, \mathtt{obs}^{\mathcal{K}^*})$ for observations, where $\mathtt{obs}^{\mathcal{K}^*}$ maps a $\mathcal{K}^*$-state to an element in $T\{*\}$ as usual: $\mathtt{obs}^{\mathcal{K}^*}\left(\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}K_i\right) \triangleq \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}*$.

Definition 7.2 *Let $\kappa \in T(\mathcal{C}_\alpha)$ and $\mathtt{t} \in Tr(\kappa)$. Define the element $\mathtt{st}^*(\kappa,\mathtt{t}) \in T1$ as follows:*

$$\mathtt{st}^*(\kappa,\varepsilon) \triangleq \mathtt{obs}^{\mathcal{K}^*}(\kappa); \qquad\qquad \mathtt{st}^*(\kappa, a\cdot\mathtt{u}) \triangleq \mathtt{st}^*(\rho,\mathtt{u}) \textit{ where } \kappa \overset{a}{\Longrightarrow} \rho$$

Trace equivalence $\simeq_{\mathrm{Tr}}^{\mathcal{K}^*}$ is the relation on $\mathcal{K}^*$-states thus defined:

$$\kappa \simeq_{\mathrm{Tr}}^{\mathcal{K}^*} \rho \iff Tr(\kappa) = Tr(\rho) \wedge \forall \mathtt{t} \in Tr(\kappa).\ \mathtt{st}^*(\kappa,\mathtt{t}) = \mathtt{st}^*(\rho,\mathtt{t}).$$

Lemma 7.1 *Given a $\mathcal{K}^*$-state $\xi = \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}K_i$ and a trace $t \in Tr(\xi)$, we have:*

$$\mathtt{st}^*\left(\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}K_i, t\right) = \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\mathtt{st}(K_i,t)$$

Proof. First of all observe that if $t \in Tr(\xi)$, then $t \in Tr(K_i)$, for any i. In fact, say $\xi \in T(\mathcal{C}_\alpha)$, so that $K_i \in \mathcal{C}_\alpha$ for any i. Since whether $t \in Tr(K_i)$ is determined by the type of K_i, we indeed have $t \in Tr(K_i)$, for any $i \in n$. We now prove the thesis by induction on t. The case for $t = \varepsilon$ is trivial. Suppose $t = a \cdot u$. Since $t \in Tr(\xi)$, we have $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}K_i \overset{a}{\Longrightarrow} \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\boxed{\alpha_i}\overset{\boldsymbol{m_i}}{\underset{j}{\text{———}}}L_j$, with $K_i \xrightarrow{a} \boxed{\alpha_i}\overset{\boldsymbol{m_i}}{\underset{j}{\text{———}}}L_j$. We can thus compute:

$$\begin{aligned}
\mathtt{st}^*\left(\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}K_i, a\cdot u\right) &= \mathtt{st}^*\left(\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\boxed{\alpha_i}\overset{\boldsymbol{m_i}}{\underset{j}{\text{———}}}L_j, u\right)\\
&\overset{IH}{=} \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\boxed{\alpha_i}\overset{\boldsymbol{m_i}}{\underset{j}{\text{———}}}\mathtt{st}(L_j,u)\\
&= \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}}\mathtt{st}(K_i, a\cdot u)
\end{aligned}$$

□

Corollary 7.2 *Given two $\mathcal{K}$-states K, L, we have $K \leq_{\mathrm{Tr}}^{\mathcal{K}} L$ if and only if $\boxed{\eta}\text{———}K \leq_{\mathrm{Tr}}^{\mathcal{K}^*} \boxed{\eta}\text{———}L$.*

Finally, we take advantage of the deterministic nature of $\mathcal{K}^*$ and characterise trace equivalence coinductively as $\mathcal{K}^*$-bisimilarity.

Definition 7.3 *Define $\mathcal{K}^*$-bisimilarity $\simeq^{\mathcal{K}^*}$ as the largest relation R on $\mathcal{K}^*$-states such that:*

- $\xi \, R \, \varphi$ *and* $\xi \overset{a}{\Longrightarrow} \xi'$ *implies* $\varphi \overset{a}{\Longrightarrow} \varphi'$ *and* $\xi' \, R \, \varphi'$
- $\xi \, R \, \varphi$ *implies* $\mathtt{obs}^{\mathcal{K}^*}(\xi) \sqsubseteq \mathtt{obs}^{\mathcal{K}^*}(\varphi)$.

As usual, since $\mathtt{obs}^{\mathcal{K}^*}$ is monotone we can define $\lesssim^{\mathcal{K}^*}$ coinductively as the greatest fixed point of a suitable monotone function. Moreover, $\mathcal{K}^*$ being deterministic, we can recover $\mathcal{K}^*$-bisimilarity as the intersection of $\lesssim^{\mathcal{K}^*}$ and its dual.

Proposition 7.3 $\lesssim^{\mathcal{K}^*} = \leq^{\mathcal{K}^*}_{\mathtt{Tr}}$.

Proof. Obviously $\lesssim^{\mathcal{K}^*}$ is contained in $\leq^{\mathcal{K}^*}_{\mathtt{Tr}}$. For the converse, observe that $\leq^{\mathcal{K}^*}_{\mathtt{Tr}}$ is a simulation. $\square$

Finally, we recover standard inductive reasoning on finite approximations of program semantics by means of finite-step simulation.

Lemma 7.4 *Let $k \geq 0$. Define system $\mathcal{K}_k$ by replacing $[\![e]\!]^\Lambda$ with $[\![e]\!]^\Lambda_k$ in Definition* **??**, *and system $\mathcal{K}^*_k$ by replacing $\mathcal{K}$ with $\mathcal{K}_k$ in Definition 7.1 Let $\lesssim^{\mathcal{K}^*}_k$ be similarity on $\mathcal{K}^*_k$ and define* finite-step similarity $\lesssim^{\mathcal{K}^*}_{\mathtt{fin}}$ *as $\bigcap_k \lesssim^{\mathcal{K}^*}_k$. Then $\lesssim^{\mathcal{K}^*}_{\mathtt{fin}} = \lesssim^{\mathcal{K}^*}$.*

Proof. [Proof sketch] The hard part is proving $\lesssim^{\mathcal{K}^*}_{\mathtt{fin}} \subseteq \lesssim^{\mathcal{K}^*}$. For that we show that the $\mathcal{K}^*$-relation $R \triangleq \{(\bigsqcup_n \kappa_n, \rho) \mid \kappa_n \lesssim^{\mathcal{K}^*}_{\mathtt{fin}} \rho\}$ is a $\mathcal{K}^*$-simulation. To do so we rely on the ω-cppo-enrichment of T and use diagonalisation of double chains (given a sequence $(x_{n,m})_{n,m}$ in a domain, if $n \leq n'$ and $m \leq m'$ imply $x_{n,m} \sqsubseteq x_{n',m'}$, then $\bigsqcup_n \bigsqcup_m x_{n,m} = \bigsqcup_m \bigsqcup_n x_{n,m} = \bigsqcup_k x_{k,k}$). $\square$

7.2 From $\mathcal{K}^*$ to $\mathcal{F}^*$

The next step of our construction is to equip $\mathcal{K}^*$-states with contexts. To do so, we first define the notion of a *context for a configuration.* Without much of a surprise, the latter is modelled as an open term t whose free variables can be instantiated with terms in configurations. However, in order to properly account for configurations of the form $(\Gamma; \Delta; e)$, we have to consider open terms having a free variable (one is enough for our purposes) acting as a placeholder for a *linearly-used* computations.

Definition 7.4 *Let $(\Gamma; \Delta)$ be a configuration with $|\Gamma| = n$ and $|\Delta| = m$. A context for $(\Gamma; \Delta)$ is simply a (n, m)-context, i.e. an open term $a_1, \ldots, a_n \mid x_1, \ldots, x_m \vdash t$. A context for a configuration $(\Gamma; \Delta; e)$ is an open term $a_1, \ldots, a_n \mid x_1, \ldots, x_m \mid z \vdash t$, where z is a* linear *placeholder for a computation.*

Due to space constraints, we do not given an explicit system for sequents of the form $\Sigma \mid \Omega \mid z \vdash t$, as such a system is standard.

Given a monadic configurations κ, we say that t is a context for κ if t is a context for all configurations in $\mathsf{supp}(\kappa)$ (notice that if t is a context for a configuration in the support of κ, then it is a context for all such configurations). If that is the case, then we can pair t and κ together, obtaining a monadic term $t[\kappa] \in T(\Lambda) \cup T(\mathcal{V})$, where for $\kappa = \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{---}}} K_i$ we define $t[\kappa]$ as $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{---}}} t[K_i]$.

In order to study the computational behaviour of a $\mathcal{K}^*$-state paired with a context for it, we define a new system, called $\mathcal{F}$, whose states are figures of the form $t : \kappa$, with t context for κ. If $t[\kappa] \in T(\mathcal{V})$, then we say that $t : \kappa$ is a $\mathcal{F}$-value state (similarity, we have $\mathcal{F}$-computation states when $t[\kappa] \in T(\Lambda)$: by type assumption, these are the only possible cases). The dynamics of $\mathcal{F}$ is given by an evaluation function $[\![-]\!]^{\mathcal{F}}$ mapping $\mathcal{F}$-computation states to monadic $\mathcal{F}$-value states.

In order to facilitate the definition of $[\![-]\!]^{\mathcal{F}}$, it is convenient to first extend the action of $[\![-]\!]^\Lambda$ to *open* terms. We do so following [39].

Definition 7.5 *An evaluation context E is an expression generated by the following grammar:*

$$E ::= [-] \mid \textbf{let } x = E \textbf{ in } e.$$

A stuck expression is an expression of the form $E[\mathsf{s}]$, where s is an expression having the shape of a redex but whose evaluation is stuck, due to the presence of a variable. Formally, stuck expressions are thus defined:

$$\mathsf{s} \;::=\; a \mid z \mid xv \mid \textbf{let } !a = x \textbf{ in } e$$

We are now ready to define the dynamics $[\![-]\!]^{\mathcal{F}}$ of system $\mathcal{F}$. As usual, we define $[\![-]\!]^{\mathcal{F}}$ as $\bigsqcup_{k\geq 0}[\![-]\!]^{\mathcal{F}}_k$, where $[\![t : \kappa]\!]^{\mathcal{F}}_0 \triangleq \boxed{\bot}$. In order to define $[\![t : \kappa]\!]^{\mathcal{F}}_k$, where $t : \kappa$ is a $\mathcal{F}$-computation state, we proceed by cases on t.

Case 1. Suppose t is neither a value nor a stuck term. Then $[\![t : \xi]\!]^{\mathcal{F}}_{k+1}$ simply evaluates t.

$$\begin{aligned}
[\![\textbf{val } t : \xi]\!]^{\mathcal{F}}_{k+1} &\triangleq \boxed{\eta}\text{——} t : \xi\\
[\![E[\textbf{let } x = (\textbf{val } t) \textbf{ in } s] : \xi]\!]^{\mathcal{F}}_{k+1} &\triangleq [\![E[s[x := t]] : \xi]\!]^{\mathcal{F}}_k\\
[\![E[(\lambda x.s)t] : \xi]\!]^{\mathcal{F}}_{k+1} &\triangleq [\![E[s[x := t]] : \xi]\!]^{\mathcal{F}}_k\\
[\![E[\textbf{let } !a = !t \textbf{ in } s] : \xi]\!]^{\mathcal{F}}_{k+1} &\triangleq [\![E[s[a := t]] : \xi]\!]^{\mathcal{F}}_k\\
[\![E[\textbf{op}(t_1, \dots, t_n)]]\!]^{\mathcal{F}}_{k+1} &\triangleq \boxed{[\![\textbf{op}]\!]}\overset{\boldsymbol{n}}{\underset{i}{\text{——}}}[\![E[t_i]]\!]^{\mathcal{F}}_k
\end{aligned}$$

Case 2. Suppose t is of the form $E[z]$. We do a further analysis on the shape of E. In the following, we write $[\![K]\!]^{\mathcal{K}}_k$, where K is a configuration of the form $(\Gamma; \Delta; e)$, for $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{——}}}(\Gamma; \Delta, v_i)$, where $[\![e]\!]^{\Lambda}_k = \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{——}}}v_i$. We extend $[\![-]\!]^{\mathcal{K}}$ to a map $[\![-]\!]^{\mathcal{K}^*}$ acting on $\mathcal{K}^*$-states as usual.

Case 2.1. Consider the case for $z : \xi$. Since z is a context for ξ, ξ must have the form $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{——}}}(\emptyset; \emptyset; e_i)$, so that $[\![\xi]\!]^{\mathcal{K}^*}_k$ have the form $\boxed{\alpha}\overset{\boldsymbol{m}}{\underset{j}{\text{——}}}(\emptyset; v_j)$. Define:

$$[\![z : \xi]\!]^{\mathcal{F}}_{k+1} \triangleq \boxed{\eta}\text{——} x_1 : [\![\xi]\!]^{\mathcal{K}^*}_k.$$

Case 2.2. Consider the case for $E[\textbf{let } x = z \textbf{ in } s] : \xi$. As before, we must have that any configuration K in the support of ξ must have the form $(\Gamma; \Delta; e)$. Therefore, any configuration in the support of $[\![K]\!]^{\mathcal{K}^*}_k$ must have the form $(\Gamma; \Delta, v)$. Let $|\Delta| = n$. Define:

$$[\![E[\textbf{let } x = z \textbf{ in } s] : \xi]\!]^{\mathcal{F}}_{k+1} \triangleq [\![E[s[x := x_{n+1}]] : [\![\xi]\!]^{\mathcal{K}^*}_k]\!]^{\mathcal{F}}_k$$

Case 3. We have to consider those cases $E[\mathsf{s}]$ where the stuck expression s comes from a variable acting as a placeholder for a resource in a configuration of the form $(\Gamma; \Delta)$. In those cases we just mimic transitions in $\mathcal{K}^*$ and update $E[\mathsf{s}]$ accordingly. Notice that this is exactly what we have done in case 2, where we have mimicked *eval* actions.

- Consider the case for $E[a_i]$. Since the latter is a context for ξ, any configuration K in the support of ξ must have the form $(\Gamma[e]_i; \Delta)$. As a consequence, we have the $\mathcal{K}$-transition $K \xrightarrow{!_i} \boxed{\eta}\text{——}(\Gamma[e]_i; \Delta; e)$, and thus a $\mathcal{K}^*$-transition from ξ, say to φ. Define:
$$[\![E[a_i] : \xi]\!]^{\mathcal{F}}_{k+1} \triangleq [\![E[z] : \varphi]\!]^{\mathcal{F}}_k$$

- Consider the case for $E[\mathbf{let}\ !a = x_i\ \mathbf{in}\ t]$. Since the latter is a context for ξ, any configuration K in the support of ξ must have the form $(\Gamma; \Delta[!e]_i)$. As a consequence, we have the $\mathcal{K}$-transition $K \xrightarrow{?_i} \boxed{\eta}\!\!-\!\!-\!\!-(\Gamma, e; \Delta)$, and thus a $\mathcal{K}^*$-transition from ξ, say to φ. Let $|\Gamma| = n$. Define:

$$[\![E[\mathbf{let}\ !a = x_i\ \mathbf{in}\ t] : \xi]\!]^{\mathcal{F}}_{k+1} \triangleq [\![E[\mathbf{let}\ !a = !a_{n+1}\ \mathbf{in}\ t] : \varphi]\!]^{\mathcal{F}}_{k}.$$

- Finally, consider the case for $E[x_i t]$. Since the latter is a valid context for ξ, (i) any configuration K in the support of ξ must have the form $(\Gamma; \Delta[\lambda x.f]_i)$, (ii) there must exist sequences $i \notin \bar{s}, \bar{p}$ such that $\bar{s}, \bar{p}$ are valid for Γ, Δ respectively, and (iii) t is a $(|\bar{s}|, |\bar{p}|)$-value context (and thus t open value). As a consequence, we have the $\mathcal{K}$-transition

$$(\Gamma; \Delta[\lambda x.f]_i) \xrightarrow{(\bar{s},\bar{p},i,t)} \boxed{\eta}\!\!-\!\!-\!\!-(\Gamma; \Delta \ominus \bar{p}; (\lambda x.f)t[\Gamma_{\bar{s}}, \Delta_{\bar{p}}])$$

and thus $\xi \xRightarrow{(\bar{s},\bar{p},i,t)} \varphi$, for a suitable $\mathcal{K}^*$-state φ. Let $|\Delta \ominus \bar{p}| = n$. Define:

$$[\![E[x_i t] : \xi]\!]^{\mathcal{F}}_{k+1} \triangleq [\![E^\star[z] : \varphi]\!]^{\mathcal{F}}_{k}$$

where $E^\star$ is the re-indexing of free variables of E according to $\Delta \ominus \bar{p}$. That is, recall that a context for $(\Gamma; \Delta)$ with $|\Gamma| = n$, $|\Delta| = m$, is a term $a_1, \ldots, a_n \mid x_1, \ldots, x_m \vdash t$ with the intended meaning that, e.g., variable x_i is a placeholder for the i-th value in Δ. Say the latter is v. When passing from Δ to $\Delta \ominus \bar{p}$ we change the position of values in Δ, so that the i-th value in Δ (i.e. v) (to which we associate the variable x_i in t) may not be at position i in $\Delta \ominus \bar{p}$. Therefore, we have to change the index i in x_i to an index j in such a way that x_j is associated to v in $\Delta \ominus \bar{p}$. Such a re-indexing can be easily done observing that if v has position i in Δ, then it has position $i - |\{p \in \bar{p} \mid p < i\}|$ in $\Delta \ominus \bar{p}$.

We summarise the defining rules of $[\![t : \kappa]\!]^{\mathcal{F}}_{k+1}$ in Figure 6, where we employ the notation used in the above discussion.

$$\begin{aligned}
[\![\mathbf{val}\ t : \kappa]\!]^{\mathcal{F}}_{k+1} &\triangleq \boxed{\eta}\!\!-\!\!-\!\!-t : \kappa \\
[\![E[r] : \kappa]\!]^{\mathcal{F}}_{k+1} &\triangleq [\![E[r'] : \kappa]\!]^{\mathcal{F}}_{k} \\
[\![E[\mathbf{op}(t_1, \ldots, t_n)] : \kappa]\!]^{\mathcal{F}}_{k+1} &\triangleq \boxed{[\![\mathbf{op}]\!]}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-}}[\![E[t_i] : \kappa]\!]^{\mathcal{F}}_{k} \\
[\![z : \kappa]\!]^{\mathcal{F}}_{k+1} &\triangleq \boxed{\eta}\!\!-\!\!-\!\!-x_1 : [\![\kappa]\!]^{\mathcal{K}^*}. \\
[\![E[\mathbf{let}\ x = z\ \mathbf{in}\ s] : \kappa]\!]^{\mathcal{F}}_{k+1} &\triangleq [\![E[s[x := x_{n+1}]] : [\![\kappa]\!]^{\mathcal{K}^*}]\!]^{\mathcal{F}}_{k} \\
[\![E[a_i] : \kappa]\!]^{\mathcal{F}}_{k+1} &\triangleq [\![E[z] : \rho]\!]^{\mathcal{F}}_{k}. \\
[\![E[\mathbf{let}\ !a = x_i\ \mathbf{in}\ s] : \kappa]\!]^{\mathcal{F}}_{k+1} &\triangleq [\![E[\mathbf{let}\ !a = !a_{n+1}\ \mathbf{in}\ s] : \rho]\!]^{\mathcal{F}}_{k}. \\
[\![E[x_i t] : \kappa]\!]^{\mathcal{F}}_{k+1} &\triangleq [\![E^\star[z] : \rho]\!]^{\mathcal{F}}_{k}
\end{aligned}$$

Figure 6: Definition of $[\![-]\!]^{\mathcal{F}}_{k+1}$

Finally, we determinise $\mathcal{F}$ building a new system, which we call $\mathcal{F}^*$.

Definition 7.6 *System $\mathcal{F}^*$ has monadic (well-typed) $\mathcal{F}$-states as states, where, as usual, all $\mathcal{F}$-states in the support of a $\mathcal{F}^*$-state ζ have the same type. Given a $\mathcal{F}^*$-state ζ, if* *all $\mathcal{F}$-states in its support are $\mathcal{F}$-value states, then we say that ζ is a $\mathcal{F}^*$-value state (and similarity for computation-states). The dynamics of $\mathcal{F}^*$ is given by the map $[\![-]\!]^{\mathcal{F}^*}$ mapping $\mathcal{F}^*$-computation states to $\mathcal{F}^*$-value states: $\left[\!\!\left[\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}} t_i : \kappa_i\right]\!\!\right]^{\mathcal{F}^*} \triangleq \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}} [\![t_i : \kappa_i]\!]^{\mathcal{F}}$. The map $[\![-]\!]_k^{\mathcal{F}^*}$ is defined as usual.*

We can extract an element in $T(\mathcal{V})$ out of a $\mathcal{F}^*$-value state (and an element in $T(\Lambda)$ out of a $\mathcal{F}^*$-computation state) using the function `push` mapping a $\mathcal{K}^*$-state $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}} t_i : \kappa_i$ to $\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{\text{———}}} t_i[\kappa_i]$. As expected, `push` connects $[\![-]\!]^{\mathcal{F}^*}$ and $[\![-]\!]^{\Lambda^*}$.

Lemma 7.5 *For any $\mathcal{F}^*$-computation state ζ, we have* $\texttt{push}\ [\![\zeta]\!]^{\mathcal{F}^*} = [\![\texttt{push}\ \zeta]\!]^{\Lambda^*}$.

Proof. [Proof sketch] This essentially follows from the way we have defined $[\![-]\!]^{\mathcal{F}}$. In fact, it is sufficient to prove that for any $k \geq 0$, and $\mathcal{F}$-state $t : \kappa$ we have:

$$\begin{aligned} \texttt{push}\ [\![t : \kappa]\!]_k^{\mathcal{F}} &\sqsubseteq [\![t[\kappa]]\!]^{\Lambda^*} \\ [\![t[\kappa]]\!]_k^{\Lambda^*} &\sqsubseteq \texttt{push}\ [\![t : \kappa]\!]^{\mathcal{F}} \end{aligned}$$

The proof proceeds by induction on k. □

We have thus came up with a way to relate system $\mathcal{F}^*$ with monadic terms. We summarise such a relationship in the following commutative diagram, where $\texttt{obs}^{\mathcal{F}^*}$ abbreviates $\texttt{obs}^{\Lambda^*} \circ \texttt{push}$, and we write $\mathcal{F}^*_\Lambda$ for the restriction of $\mathcal{F}^*$ to $\mathcal{F}^*$-computation states (similarity, we use the subscript $\mathcal{V}$ for $\mathcal{F}^*$-value states).

$$\begin{tikzcd}
T\Lambda \arrow[r, "{[\![-]\!]^{\Lambda^*}}"] & T\mathcal{V} \arrow[r, "\texttt{obs}^{\Lambda^*}"] & T1 \\
\mathcal{F}^*_\Lambda \arrow[u, "\texttt{push}"] \arrow[r, "{[\![-]\!]^{\mathcal{F}^*}}"] & \mathcal{F}^*_\mathcal{V} \arrow[u, "\texttt{push}"] \arrow[ur, "\texttt{obs}^{\mathcal{F}^*}"'] &
\end{tikzcd}$$

Notice that the ω-cppo-enrichment of T implies continuity of `push`, and thus of $\texttt{obs}^{\mathcal{F}^*}$, since $\texttt{obs}^{\Lambda^*}$ is continuous.

We are now ready to prove soundness of $\simeq^{\Lambda}_{\text{Tr}}$ for $\equiv^{\text{ctx}}$. Concretely, what we have to prove is that $e \simeq^{\Lambda}_{\text{Tr}} f$ implies $\texttt{obs}^{\Lambda^*}[\![C[e]]\!]^{\Lambda} = \texttt{obs}^{\Lambda^*}[\![C[f]]\!]^{\Lambda}$, for any context C. To prove such a statement we need to study the computational behaviour of $C[e]$ and $C[f]$. The right setting to do so, is, obviously, system $\mathcal{F}^*$. Hence, we need to move from programs to $\mathcal{F}^*$-states. We do so by mapping $C[e]$ to $\zeta_e \triangleq \boxed{\eta}\text{———}C : \boxed{\eta}\text{———}(\emptyset; \emptyset; e)$ (ans similarity we map $C[f]$ to a $\mathcal{F}^*$-state ζ_f). By Lemma 7.5, we recover $[\![C[e]]\!]^{\Lambda}$ as $\texttt{push}\ [\![\zeta_e]\!]^{\mathcal{F}^*}$, and thus $\texttt{obs}^{\Lambda^*}[\![C[e]]\!]^{\Lambda}$ as $\texttt{obs}^{\mathcal{F}^*}[\![\zeta_e]\!]^{\mathcal{F}^*}$. We thus find ourselves in a situation of the form:

$$\begin{array}{ccc}
\boxed{\eta}\text{———}\Big(C : \boxed{\eta}\text{———}(\emptyset; \emptyset; e)\Big) & \texttt{F}(\leq^{\Lambda}_{\text{Tr}}) & \boxed{\eta}\text{———}\Big(C : \boxed{\eta}\text{———}(\emptyset; \emptyset; f)\Big) \\
\downarrow & & \downarrow \\
\left[\!\!\left[\boxed{\eta}\text{———}\Big(C : \boxed{\eta}\text{———}(\emptyset; \emptyset; e)\Big)\right]\!\!\right]^{\mathcal{F}^*} & ? & \left[\!\!\left[\boxed{\eta}\text{———}\Big(C : \boxed{\eta}\text{———}(\emptyset; \emptyset; f)\Big)\right]\!\!\right]^{\mathcal{F}^*}
\end{array}$$

Here, $\texttt{F}(\leq^{\Lambda}_{\text{Tr}})$ is a lifting of $\leq^{\Lambda}_{\text{Tr}}$ to $\mathcal{F}^*$-states, and the question mark ? stands for a relation on $\mathcal{F}^*$-states such that:

1. Whenever $Z\ \texttt{F}(\leq^{\Lambda}_{\text{Tr}})\ U$, we have $[\![Z]\!]^{\mathcal{F}^*}\ ?\ [\![U]\!]^{\mathcal{F}^*}$
2. $Z\ ?\ U$ implies $\texttt{obs}^{\mathcal{F}^*}(Z) \sqsubseteq \texttt{obs}^{\mathcal{F}^*}(U)$.

Obviously, if we are able to find such a relation ? (as well as a suitable lifting $\mathtt{F}$), we can conclude the wished thesis.

Let us examine how the definition of the definition of $[\![-]\!]^{\mathcal{F}^*}$ proceeds. First, we evaluate the context C, this way obtaining the diagram:

$$
\begin{array}{ccc}
\boxed{\eta}-\!\!-\!\!-\Big(C : \boxed{\eta}-\!\!-\!\!-(\emptyset;\emptyset;e)\Big) & \mathtt{F}(\leq^{\Lambda}_{\mathrm{Tr}}) & \boxed{\eta}-\!\!-\!\!-\Big(C : \boxed{\eta}-\!\!-\!\!-(\emptyset;\emptyset;f)\Big) \\
\downarrow & & \downarrow \\
\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-\!\!-}}\Big(s_i : \boxed{\eta}-\!\!-\!\!-(\emptyset;\emptyset;e)\Big) & ? & \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-\!\!-}}\Big(s_i : \boxed{\eta}-\!\!-\!\!-(\emptyset;\emptyset;f)\Big)
\end{array}
$$

Next, we have the interaction between e (resp. f) and the new contexts s_is:

$$
\begin{array}{ccc}
\boxed{\eta}-\!\!-\!\!-\Big(C : \boxed{\eta}-\!\!-\!\!-(\emptyset;\emptyset;e)\Big) & \mathtt{F}(\leq^{\Lambda}_{\mathrm{Tr}}) & \boxed{\eta}-\!\!-\!\!-\Big(C : \boxed{\eta}-\!\!-\!\!-(\emptyset;\emptyset;f)\Big) \\
\downarrow & & \downarrow \\
\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-\!\!-}}\Big(s_i : \boxed{\eta}-\!\!-\!\!-(\emptyset;\emptyset;e)\Big) & ? & \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-\!\!-}}\Big(s_i : \boxed{\eta}-\!\!-\!\!-(\emptyset;\emptyset;f)\Big) \\
\downarrow & & \downarrow \\
\boxed{\alpha}\overset{\boldsymbol{m}}{\underset{j}{-\!\!-\!\!-}}\Big(s'_j : \boxed{\gamma_1}\overset{\boldsymbol{n_1}}{\underset{i}{-\!\!-\!\!-}}K_i\Big) & ? & \boxed{\alpha}\overset{\boldsymbol{m}}{\underset{j}{-\!\!-\!\!-}}\Big(s'_j : \boxed{\gamma_2}\overset{\boldsymbol{n_2}}{\underset{i}{-\!\!-\!\!-}}L_i\Big)
\end{array}
$$

where $\boxed{\gamma_1}\overset{\boldsymbol{n_1}}{\underset{i}{-\!\!-\!\!-}}K_i \leq^{\mathcal{K}^*}_{\mathrm{Tr}} \boxed{\gamma_2}\overset{\boldsymbol{n_2}}{\underset{i}{-\!\!-\!\!-}}L_i$, and thus $\boxed{\gamma_1}\overset{\boldsymbol{n_1}}{\underset{i}{-\!\!-\!\!-}}K_i \lesssim^{\mathcal{K}^*} \boxed{\gamma_2}\overset{\boldsymbol{n_2}}{\underset{i}{-\!\!-\!\!-}}L_i$. At this point the invariant should be clear. All $\mathcal{F}^*$-states in the diagram have the same 'outermost effect', argumentwise equal contexts, and $\mathcal{K}^*$-similar inner ($\mathcal{K}^*$-)states. This directly leads us to the following definition.

We thus have two $\mathcal{F}^*$-states, ζ_e and ζ_f, which are related by the obvious lifting of $\simeq^{\Lambda}_{\mathrm{Tr}}$ to $\mathcal{F}^*$-states. Such a lifting, however, is not preserved by the dynamics of $\mathcal{F}^*$. In order to conclude the wished thesis, we need a stronger relation. In fact, it is sufficient to find a relation R on $\mathcal{F}^*$-states such that

(i) $\zeta_e \, R \, \zeta_f$,

(ii) R is closed under the dynamics of $\mathcal{F}^*$ (meaning that $\zeta \, R \, \theta$ implies $[\![\zeta]\!]^{\mathcal{F}^*} \, R \, [\![\theta]\!]^{\mathcal{F}^*}$), and

(iii) R respects $\mathtt{obs}^{\mathcal{F}^*}$, meaning that $\zeta \, R \, \theta$ implies $\mathtt{obs}^{\mathcal{F}^*}(\zeta) = \mathtt{obs}^{\mathcal{F}^*}(\theta)$.

Actually, since $\mathtt{obs}^{\mathcal{F}^*}$ is continuous, we can replace implication (ii) with

(ii') $\zeta \, R \, \theta$ implies $\forall k \geq 0.\ [\![\zeta]\!]^{\mathcal{F}^*}_k \, R \, [\![\theta]\!]^{\mathcal{F}^*}_k$.

Indeed, if that is the case, then by (iii) we have $\mathtt{obs}^{\mathcal{F}^*} \ [\![\zeta]\!]^{\mathcal{F}^*}_k = \mathtt{obs}^{\mathcal{F}^*} \ [\![\theta]\!]^{\mathcal{F}^*}_k$, for any $k \geq 0$ We conclude:

$$
\mathtt{obs}^{\mathcal{F}^*} \ [\![\zeta]\!]^{\mathcal{F}^*} = \mathtt{obs}^{\mathcal{F}^*} \ \bigsqcup_k [\![\zeta]\!]^{\mathcal{F}^*}_k = \bigsqcup_k \mathtt{obs}^{\mathcal{F}^*} \ [\![\zeta]\!]^{\mathcal{F}^*}_k = \bigsqcup_k \mathtt{obs}^{\mathcal{F}^*} \ [\![\theta]\!]^{\mathcal{F}^*}_k = \mathtt{obs}^{\mathcal{F}^*} \ [\![\theta]\!]^{\mathcal{F}^*}
$$

Coming back to the task of finding the desired R, we see that the latter is essentially given by the following diagram, which is obtained by very definition of $[\![-]\!]^{\mathcal{F}^*}$.

$$
\begin{array}{ccc}
\boxed{\eta}\!-\!\!-\!\!-C : \boxed{\eta}\!-\!\!-\!\!-(\emptyset;\emptyset;e) & R & \boxed{\eta}\!-\!\!-\!\!-C : \boxed{\eta}\!-\!\!-\!\!-(\emptyset;\emptyset;f) \\
\downarrow & & \downarrow \\
\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-\!\!-}}t_i : \boxed{\eta}\!-\!\!-\!\!-(\emptyset;\emptyset;e) & R & \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-\!\!-}}t_i : \boxed{\eta}\!-\!\!-\!\!-(\emptyset;\emptyset;f) \\
\downarrow & & \downarrow \\
\boxed{\alpha}\overset{\boldsymbol{m}}{\underset{j}{-\!\!-\!\!-}}s_j : \boxed{\Xi_j}\overset{\boldsymbol{l_j}}{\underset{a}{-\!\!-\!\!-}}K_a & R & \boxed{\alpha}\overset{\boldsymbol{m}}{\underset{j}{-\!\!-\!\!-}}s_j : \boxed{\Upsilon_j}\overset{\boldsymbol{h_j}}{\underset{b}{-\!\!-\!\!-}}L_b
\end{array}
$$

Suppose we begin by evaluating the context C in isolation, i.e. without interacting with e (resp. f) This is nothing but the application of the first five rules in Figure 6, and corresponds to moving from the first to second raw in the above diagram. Notice that now states consist of equal outermost (generic) effects and argumentwise equal open terms.

Next, we have the interaction between e (resp. f) and the new contexts t_i. By inspection of the rules in Figure 6, we see that e and f, as well as the monadic configurations coming from their evaluation, do not affect the outermost generic effects, which, instead, are modified by the terms t_i only, thus remaining equals. Additionally, the open terms t_i can be modified by e and f (and their monadic configurations coming from their evaluation) only by renaming variables and by replacing stuck terms with variables. Such modifications correspond to $\mathcal{K}^*$-actions only, and e and f having the same traces, they can make the same modifications. We thus see that what we have reached are $\mathcal{F}^*$-states with the same outermost generic effect, argumentwise equal open terms, and, thanks to Proposition 7.3, $\simeq^{\mathcal{K}^*}_{\mathtt{Tr}}$-related inner $\mathcal{K}$-states (meaning that $\boxed{\Xi_j}\overset{\boldsymbol{l_j}}{\underset{a}{-\!\!-\!\!-}}K_a \simeq^{\mathcal{K}^*}_{\mathtt{Tr}} \boxed{\Upsilon_j}\overset{\boldsymbol{h_j}}{\underset{b}{-\!\!-\!\!-}}L_b$ in the diagram above). This is our invariant.

By inspecting rules in Figure 6, we see that the desired relation is thus defined as follows:

Definition 7.7 *Let R be a $\mathcal{K}^*$-relation. Define the $\mathcal{F}^*$-relation $\mathtt{BC}(R)$, called the* Barr and contextual closure *of R, as:*

$$
\mathtt{BC}(R) \triangleq \Big\{\Big(\boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-\!\!-}}t_i;\kappa_i, \boxed{\gamma}\overset{\boldsymbol{n}}{\underset{i}{-\!\!-\!\!-}}t_i;\rho_i\Big) \mid \gamma \in T(\boldsymbol{n}) \wedge \forall i.\ \kappa_i\ R\ \rho_i\Big\}.
$$

Lemma 7.6 (Main Lemma) *For all $\mathcal{F}^*$-states ζ, θ:*

$$
\zeta\ \mathtt{BC}(\simeq^{\mathcal{K}^*}_{\mathtt{Tr}})\ \theta \implies \forall k \geq 0.\ [\![\zeta]\!]^{\mathcal{F}^*}_k\ \mathtt{BC}(\simeq^{\mathcal{K}^*}_{\mathtt{Tr}})\ [\![\theta]\!]^{\mathcal{F}^*}_k.
$$

The proof of Lemma 7.6 follows the informal intuition given in the above discussion, and proceeds by induction on k taking advantage of the equality $\simeq^{\mathcal{K}^*}_{\mathtt{Tr}} = \simeq^{\mathcal{K}^*}$.

Finally, since $e \simeq^{\Lambda}_{\mathtt{Tr}} f$ obviously implies $\zeta_e\ \mathtt{BC}(R)\ \zeta_f$, we obtain soundness of $\simeq^{\Lambda}_{\mathtt{Tr}}$ for contextual equivalence.

Theorem 7.7 $\simeq^{\Lambda}_{\mathtt{Tr}} \subseteq \equiv^{\mathrm{ctx}}$.

As already anticipated, $\simeq^{\Lambda}_{\mathtt{Tr}}$ is also complete for $\equiv^{\mathrm{ctx}}$. This is proved by showing that $\equiv^{\mathrm{ctx}} \subseteq \simeq^{\Lambda}_{\mathtt{Tr}}$, which is itself proved by noticing that any $\mathcal{K}$-action can be encoded as a context.

Theorem 7.8 $\equiv^{\mathrm{ctx}} = \simeq^{\Lambda}_{\mathtt{Tr}}$.

We omit the proof of this result, as the encoding of $\mathcal{K}$-actions as contexts is essentially the same one of [17].

8 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, 27, 13, 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 [28].

Finally, as a long term future work, the authors would like to whether the ideas presented in this paper can be adapted to deal with quantum languages [65, 66], 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 [16].

8.1 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 [10], of applicative bisimilarity by Bierman [9] and Crole[3] [14], of trace equivalence by Deng and Zhang [24, 23], as well as of a number of possible worlds-indexed equivalences (e.g. [2, 35]). 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 notion 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 [18], 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[4].

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 worst, natural extensions of such notions to languages with copying result in equivalences validating (!-dist). Crubillé and Dal Lago [17] 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.

[3] Crole's applicative bisimilarity, however, does not deal with copying.

[4] Besides, notice that bisimilarity being sensitive to branching, it naturally invalidates (λ-dist).

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] 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.

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

[10] 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.

[11] 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.

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

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

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

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

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

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

[18] 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.

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

[20] Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. In *TProc. of TLCA 2009*, pages 80–94, 2009.

[21] 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.

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

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

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

[25] 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.

[26] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[27] 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.

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

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

[30] 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.

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

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

[33] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electr. Notes Theor. Comput. Sci.*, 172:437–458, 2007.

[34] 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.

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

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

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

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

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

[40] Tom Leinster. *Higher Operads, Higher Categories.* London Mathematical Society Lecture Note Series. Cambridge University Press, 2004.

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

[42] F. E. J. Linton. Some aspects of equational categories. In S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl, editors, *Proceedings of the Conference on Categorical Algebra*, pages 84–94. Springer Berlin Heidelberg, 1966.

[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 Power. Countable lawvere theories and computational effects. *Electr. Notes Theor. Comput. Sci.*, 161:59–71, 2006.

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

[63] 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.

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

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

[66] 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.

[67] 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.

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

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

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

[71] 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.