



HAL
open science

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

Ugo Dal Lago, Francesco Gavazzo

► **To cite this version:**

Ugo Dal Lago, Francesco Gavazzo. Resource Transition Systems and Full Abstraction for Linear Higher-Order Effectful Programs. FSCD 2021 - 6th International Conference on Formal Structures for Computation and Deduction, Jul 2021, Buenos Aires, Argentina. hal-03520742

HAL Id: hal-03520742

<https://inria.hal.science/hal-03520742v1>

Submitted on 11 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Ugo Dal Lago @ ORCID

University of Bologna, Italy

INRIA Sophia Antipolis, France

Francesco Gavazzo @ ORCID

University of Bologna, Italy

INRIA Sophia Antipolis, France

Abstract

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

2012 ACM Subject Classification Theory of computation \rightarrow Operational semantics

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

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

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

1 Introduction

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

Algebraic Effects Since the early days of programming language semantics, the study of computational effects, i.e. those aspects of computations that go beyond the pure process of computing, has been of paramount importance. Starting with the seminal work by Moggi [47, 48], modelling and understanding computational effects in terms of monads [41] has been a standard practice in the denotational semantics of higher-order sequential languages. More recently, Plotkin and Power [58, 55, 56] 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

44 the development of, e.g., effectful logical relations [35, 12], effectful applicative and normal
 45 form/open bisimulation [21, 19], and logic-based equivalences [65, 44].

46 **Linearity and Copying** The analysis of effectful computations in terms of monads and
 47 algebraic effects is, in its very essence, *extensional*: ultimately, a program represents a function
 48 from inputs to monadic outputs. However, when reasoning about computational effects, also
 49 *intensional* aspects of programs may be relevant. In particular, *linearity* [33, 67, 8] (and
 50 its quantitative refinements [32, 31, 14, 4, 23]) has been recognised as a fundamental tool
 51 to reason about computational effects [28, 46], as witnessed by a number of programming
 52 languages, such as Clean [53], Rust [45], Granule [50], and Linear Haskell [9], which explicitly
 53 rely on linearity to structure and manage effects. Indeed, the interaction between linearity,
 54 copying, and computational effects deeply influences program equivalence: there are effectful
 55 programs that cannot be discriminated without allowing the environment to copy them, and
 56 thus program transformations which are *sound* if linearity is guaranteed, but *unsound* in
 57 presence of copying.

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

67 **Our Contribution** In this paper we introduce *resource transition systems* as an intensional,
 68 resource-sensitive operational semantics for linear languages with algebraic operations and
 69 explicit copying. Resource transition systems combine standard *extensional* properties of
 70 effectful computations with linearity and copying, whose nature is, instead, *intensional*. We
 71 model the former using monads—as one does for ordinary effectful semantics—and the latter
 72 by shifting from program-based transition systems to *tuple-based* transition systems, as one
 73 does in environmental bisimulation [60, 42]. Indeed, a resource transition system can be
 74 thought of as an ordinary transition system whose states are built over tuples of copyable
 75 programs and linear values representing the available resources produced by a program
 76 while interacting with the external environment. Another possible way to look at resource
 77 transition systems is as an interactive semantics defined on top of the so-called storage model
 78 [66]. We then define and study trace equivalence on resource transition systems. Our main
 79 result states that trace equivalence is *sound* and *complete* for contextual equivalence. To the
 80 best of the authors’ knowledge, this is the first full abstraction result for a linear λ -calculus
 81 with arbitrary algebraic effects and explicit copying.

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

2 Effects, Linearity, and Program Equivalence

In this section, we give a gentle introduction to program equivalence in presence of linearity, explicit copying, and effects. In this work, we are concerned with *operationally-based* equivalences, example of those being contextual and CIU equivalences [49, 43], logical relations [59, 54, 64] and, bisimulation-based equivalences [1, 38, 39, 60]. 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 [52] 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 [42], viz. notions of trace equivalence and bisimilarity.

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

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

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

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

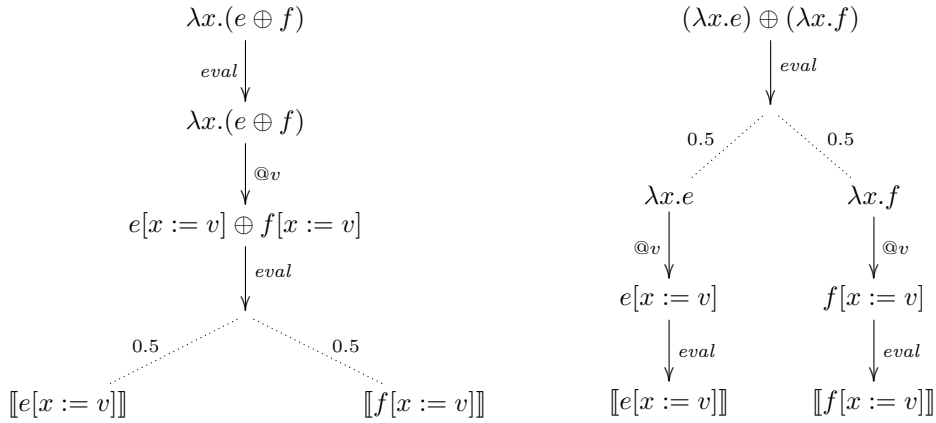


Figure 1 Interactive behaviour of $\lambda x.(e \oplus f)$ and $(\lambda x.e) \oplus (\lambda x.f)$

in *classical* (i.e. resource-agnostic) languages is that several notions of probabilistic program equivalence (such as probabilistic contextual equivalence [24], applicative bisimilarity [16, 24], and logical relations [13]) are sensitive to branching. However, sensitivity to branching does not quite feel like the crux of the failure of distributivity of abstraction over choice in classical languages. In fact, what we see is that $\lambda x.(e \oplus f)$ waits for an input, and then resolves the probabilistic choice. Dually, $(\lambda x.e) \oplus (\lambda x.f)$ first resolves the choice, and then waits

119 for an input. As a consequence, if we evaluate these programs, $\lambda x.(e \oplus f)$ essentially does
 120 nothing, whereas $(\lambda x.e) \oplus (\lambda x.f)$ probabilistically chooses if continuing with either $\lambda x.e$ or
 121 $\lambda x.f$. At this point, there is a crucial difference between the programs obtained: $\lambda x.(e \oplus f)$
 122 still has to resolve the probabilistic choice. If we were allowed to use it twice by passing it an
 123 argument v — this way resolving the choice twice — then we could observe a (probabilistic)
 124 behaviour different from both the one of $\lambda x.e$ and of $\lambda x.f$. Indeed, assuming $f[x := v]$ to
 125 diverge and $e[x := v]$ to converge (with probability 1), then, we would converge (to $e[x := v]$)
 126 with probability 0.25, in the former case, and with probability 0.5, in the latter case. To
 127 observe such a behaviour, however, it is crucial to *copy* $\lambda x.(e \oplus f)$. Otherwise, we could only
 128 interact with it by passing it an argument only *once*, this way validating (λ -dist).

129 Summing up, to invalidate (λ -dist) one has to be able to *copy* the results of the evaluation
 130 of the programs involved. This observation suggests that the deep reason why (λ -dist)
 131 fails relies on the copying capabilities of the calculus [61]. If the calculus at hand is linear
 132 (and thus offers no copying capability), we should then expect (λ -dist) to hold, while
 133 $!\lambda x.(e \oplus f) \simeq !(\lambda x.e) \oplus !(\lambda x.f)$ (and thus ultimately (!-dist)) to fail. This agrees with a
 134 recent result by Deng and Zhang [27, 26], who observed that if a calculus does not have
 135 copying capabilities, then contextual equivalence (which is *a fortiori* linear) validates (λ -dist).
 136 More generally, Deng and Zhang showed that *linear contextual equivalence*, i.e. contextual
 137 equivalence where contexts test their arguments linearly (viz. exactly once), coincides with
 138 *linear trace equivalence* in probabilistic languages.

139 But what about (!-dist)? Unfortunately, linear trace equivalence has been designed for
 140 linear languages *without* copying, only. Moreover, straightforward extensions of linear trace
 141 equivalence to languages with copying would actually validate (!-dist), trace equivalence
 142 being insensitive to branching. The situation does not change much if one looks at different
 143 forms of equivalence, such as Bierman’s applicative bisimilarity [10]. Such equivalences
 144 usually invalidate (!-dist), but they all invalidate (λ -dist), too. We interpret all of this as a
 145 symptom of the lack of intensional structure in the aforementioned notions of equivalence.
 146 Ultimately, this can be traced back to the very operational semantics of the calculus, which
 147 is meant to be an abstract description of the input-output behaviour of programs, but gives
 148 no insight into their *intensional* structure, i.e. linearity and copying in our case [66].

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

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

3 Preliminaries: Monads and Algebraic Effects

Starting with the seminal work by Moggi [47, 48], *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* [41]. Additionally, instead of defining monads on arbitrary categories, we tacitly restrict our analysis to the category of sets and functions.

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

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

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 $\eta_A : A \rightarrow T(A)$ 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 $f : A \rightarrow T(B)$, and corresponds to the sequencing constructor **let** $x = -$ **in** $-$. Following this interpretation, we can read the identities in Definition 1 as stipulating that η indeed produces no effect, and that sequencing is associative.

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

► **Example 2.** Here are examples of monads modeling some of the computational effects discussed in Section 1. Further examples, such as global stores and exceptions can be found in, e.g., [47, 68].

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:

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

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 $\mu : X \rightarrow [0, 1]$ such that $\sum_x \mu(x) \leq 1$. An element $\mu \in \mathcal{D}(A)$ gives for any $a \in A$ the probability $\mu(a)$ of returning a . Notice that working with subdistribution we can easily model divergent computations [25]. Given $a \in A$, $\eta_A(a)$ is the Dirac distribution on a (mapping a to 1 and all other elements to 0), whereas for $\mu \in \mathcal{D}(A)$ and $f : A \rightarrow \mathcal{D}(B)$ 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: $(\mu \oplus \nu)(x) \triangleq 0.5 \cdot \mu(x) + 0.5 \cdot \nu(x)$.

212 3. We model computations with output using the output monad $\mathcal{O}(X) \triangleq O^\infty \times (X + \{\uparrow\})$,
 213 where O^∞ is the set of finite and infinite strings over a fixed output alphabet O and \uparrow is
 214 a special symbol denoting divergence. An element of $\mathcal{O}(A)$ is either a pair $(o, \mathbf{inl} a)$, with
 215 $a \in A$, or a pair $(o, \mathbf{inr} \uparrow)$. The former case denotes convergence to a outputting o (in
 216 which case o is a *finite* string), whereas the former denotes divergence outputting o (in
 217 which case o can be either finite or *infinite*). Given $a \in A$, the pair $(\varepsilon, \mathbf{inr} a)$ represents
 218 the trivial computation that returns a and outputs nothing (ε denotes the empty string).
 219 Further, sequential composition of computations is defined using string concatenation as
 220 follows, where $f(a) = (o', x)$:

$$221 \quad (o, \mathbf{inr} \uparrow) \gg= f \triangleq (o, \mathbf{inr} \uparrow); \quad (o, \mathbf{inl} a) \gg= f \triangleq (oo', x).$$

223 Finally, we produce outputs using (a O -indexed family of) unary operations \mathbf{print}_c
 224 mapping (o, x) to (co, x) .

225 4. We model computations with input using the input monad $\mathcal{I}(X) = \mu\alpha.(X + \{\uparrow\}) + \alpha^I$,
 226 where I is an input alphabet (for simplicity, we take $I = \{true, false\}$). An element in
 227 $\mathcal{I}(A)$ is a binary tree whose leaves are labeled either by elements in A or by the divergent
 228 symbol \uparrow . The trivial computation returning a is the single leaf labeled by a , whereas
 229 given a tree $t \in \mathcal{I}(A)$ and a map $f : A \rightarrow \mathcal{I}(B)$, the tree $t \gg= f$ is defined by replacing
 230 the leaves of t labeled by elements $a \in A$ with $f(a)$. Finally, we consider a binary input
 231 operation whereby $\mathbf{read}(t_{true}, t_{false})$ is the tree whose left child is t_{true} and whose right
 232 child is t_{false} .

233 We restrict our analysis to monads T preserving weak pullbacks, and thus preserving
 234 injections. As a consequence, if $i : A \hookrightarrow X$ is the subset inclusion map, then $T(i) : T(A) \hookrightarrow$
 235 $T(X)$ is an injection, which can be regarded as monadic inclusion. Intuitively, given an
 236 element $\mu \in T(X)$, we think about the *smallest* set $i : A \hookrightarrow X$ such that $\mu \in T(A)$ as the
 237 *support* of μ , and denote such a set as $\mathbf{supp}(\mu)$. Of course, in general the support of an
 238 element μ may not exist and therefore we restrict our analysis to monads coming with a
 239 notion of *countable* support.

240 ► **Definition 3.** We say that a monad is *countable* if for any set X and any element
 241 $\mu \in T(X)$, there exists the smallest countable set $i : Y \hookrightarrow X$, denoted by $\mathbf{supp}(\mu)$, such that
 242 $\mu \in T(Y)$ (i.e. there exists $\nu \in T(Y)$ such that $\mu = T(i)(\nu)$).

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

247 3.1 Algebraic Effects

248 Following Example 2, let us consider a probabilistic program $e \triangleq E[e_1 \oplus e_2]$, where E is
 249 an evaluation context. The operational behaviour of e is to fairly choose $e_i \in \{e_1, e_2\}$, and
 250 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.
 251 But that is exactly the behaviour of $E[e_1] \oplus E[e_2]$, so that we have the program equivalence
 252 $E[e_1 \oplus e_2] \equiv E[e_1] \oplus E[e_2]$. It does not take much to realize that a similar equivalence holds
 253 for all operations in Example 2. Semantically, operations justifying these equivalences are
 254 known as *algebraic operations* [56, 57].

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

$$257 \quad (\mathbf{op}_X(\mu_1, \dots, \mu_n)) \gg= f = \mathbf{op}_Y(\mu_1 \gg= f, \dots, \mu_n \gg= f). \\ 258$$

259 Using algebraic operations we can model a large class of effects, including those of
 260 Example 2, pure nondeterminism (using the powerset monad and set-theoretic union as
 261 binary nondeterminism choice), imperative computations (using the global states monad and
 262 operations for reading and updating stores), as well as combinations thereof [34].

263 3.2 Continuity

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

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

- 272 ▶ **Example 6.** 1. The maybe monad is \emptyset -continuous, with $\mathcal{M}(X)$ endowed with the flat
 273 order.
 274 2. The subdistribution monad is $\{\oplus\}$ -continuous, with subdistributions ordered pointwise
 275 (i.e. $\mu \leq \nu$ if and only if $\mu(x) \leq \nu(x)$, for any $x \in X$).
 276 3. Let $\Sigma \triangleq \{\mathbf{print}_c \mid c \in O\}$. Then, the output monad is Σ -continuous, with $\mathcal{O}(A)$
 277 endowed with the order: $(o, x) \sqsubseteq (o', x')$ if and only if either $x = \mathbf{inr} \uparrow$ and $o \sqsubseteq o'$ or
 278 $x = \mathbf{inl} a = x'$ and $o = o'$.
 279 4. The input monad is $\{\mathbf{read}\}$ -continuous with respect to the standard tree ordering.

280 4 A Linear Calculus with Algebraic Effects

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

285 4.1 Syntax

286 Our vehicle calculus is a linear refinement of fine-grain call-by-value [40], which we call $\Lambda^!$.
 287 The syntax of $\Lambda^!$ is given by two syntactic classes, *values* (notation v, w, \dots) and *computations*
 288 (notation e, f, \dots), which are thus defined:

$$289 \quad v ::= x \mid \lambda x.e \mid !e \\ 290 \quad e ::= a \mid \mathbf{val} v \mid vv \mid \mathbf{let} x = e \mathbf{in} e \mid \mathbf{op}(e, \dots, e) \mid \mathbf{let} !a = v \mathbf{in} e. \\ 291$$

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

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

305 When the distinction between values and computations is not relevant, we generically
 306 refer to *terms*, and denote them as t, s, \dots . We adopt standard syntactic conventions as in
 307 [5]. In particular, we work with terms modulo renaming of bound variables, and denote by
 308 $t[x := v]$ (resp. $t[a := e]$) the result of capture-avoiding substitution of the value v (resp.
 309 computation e) for the variable x (resp. a) in t .

310 4.2 Statics

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

$$316 \quad \sigma ::= x \mid !\sigma \mid \sigma \multimap \sigma \mid \mu x.\sigma \multimap \sigma \mid \mu x.! \sigma$$

318 where x is a type variable. Types are defined up to equality, as defined in Figure 2, where
 319 $\sigma[\tau/x]$ denotes the substitution of τ for all the (free) occurrences of x in σ . In the third rule
 320 in Figure 2, we require ρ to be productive in x , meaning that each free occurrence of x in ρ
 321 is under the scope of either \multimap or $!$.

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

■ **Figure 2** Type Equality

322 In order to define the collection of well-typed expressions, we consider sequents $\Sigma \mid \Omega \vdash^v$
 323 $v : \sigma$ and $\Sigma \mid \Omega \vdash^\Lambda e : \sigma$, where Ω is a linear environment, i.e. a set without repetitions of the
 324 form $x_1 : \sigma_1, \dots, x_n : \sigma_n$, and Σ is a *non-linear* environment, i.e. a set without repetitions of
 325 the form $a_1 : \tau_1, \dots, a_n : \tau_n$. Rules for derivable sequents are given in Figure 3. We write \mathcal{V}_σ
 326 and Λ_σ for the collection of closed values and computations of type σ , respectively. We write
 327 \mathcal{V} and Λ when types are not relevant.

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

$$\begin{array}{c}
\frac{}{\Sigma \mid x : \sigma \vdash^v x : \sigma} \quad \frac{}{a : \sigma, \Sigma \mid \emptyset \vdash^{\wedge} a : \sigma} \quad \frac{\Sigma \mid x : \sigma, \Omega \vdash^{\wedge} e : \tau}{\Sigma \mid \Omega \vdash^v \lambda x. e : \sigma \multimap \tau} \quad \frac{\Sigma \mid \Omega \vdash^v v : \sigma}{\Sigma \mid \Omega \vdash^{\wedge} \mathbf{val} v : \sigma} \\
\frac{\Sigma \mid \Omega \vdash^v v : \sigma \multimap \tau \quad \Sigma \mid \Omega' \vdash^v w : \sigma}{\Sigma \mid \Omega, \Omega' \vdash^{\wedge} vw : \tau} \quad \frac{\Sigma \mid \emptyset \vdash^{\wedge} e : \sigma}{\Sigma \mid \emptyset \vdash^v !e : !\sigma} \quad \frac{\Sigma \mid \Omega \vdash^v v : !\sigma \quad \Sigma, a : \sigma \mid \Omega' \vdash^{\wedge} e : \tau}{\Sigma \mid \Omega, \Omega' \vdash^{\wedge} \mathbf{let} !a = v \mathbf{ in} e : \tau} \\
\frac{\Sigma \mid \Omega \vdash^{\wedge} e : \sigma \quad \Sigma \mid \Omega', x : \sigma \vdash^{\wedge} f : \tau}{\Sigma \mid \Omega, \Omega' \vdash^{\wedge} \mathbf{let} x = e \mathbf{ in} f : \tau} \quad \frac{\Sigma \mid \Omega \vdash^{\wedge} e_1 : \sigma \quad \dots \quad \Sigma \mid \Omega \vdash^{\wedge} e_n : \sigma}{\Sigma \mid \Omega \vdash^{\wedge} \mathbf{op}(e_1, \dots, e_n) : \sigma}
\end{array}$$

■ **Figure 3** Statics of $\Lambda^!$

4.3 Dynamics

333 The dynamic semantics of $\Lambda^!$ associates to any *closed computation* e of type σ a monadic
334 element in $T(\mathcal{V}_\sigma)$. The dynamics of $\Lambda^!$ is defined in Figure 4 by means of an \mathbb{N} -indexed family of
335 evaluation functions mapping a *closed* computation $e \in \Lambda_\sigma$ to an element $\llbracket e \rrbracket_k^\wedge \in T(\mathcal{V}_\sigma)$, where
336 we stipulate $\llbracket e \rrbracket_0^\wedge \triangleq \perp$. Since $(\llbracket e \rrbracket_k^\wedge)_{k \geq 0}$ forms an ω -chain in $T(\mathcal{V})$, we define $\llbracket e \rrbracket^\wedge \triangleq \bigsqcup_{k \geq 0} \llbracket e \rrbracket_k^\wedge$.
337 Notice that thanks to the type assumption, we ignore programs causing runtime errors.
338 Finally, we lift $\llbracket - \rrbracket^\wedge$ to monadic computations, i.e. to elements $\xi \in T(\Lambda)$ by setting
339 $\llbracket \xi \rrbracket^{\wedge*} \triangleq \xi \gg= (e \rightarrow \llbracket e \rrbracket^\wedge)$ (and similarity for $\llbracket - \rrbracket_k^\wedge$).
340

$$\begin{array}{l}
\llbracket \mathbf{val} v \rrbracket_{k+1}^\wedge \triangleq \eta(v) \\
\llbracket (\lambda x. e) v \rrbracket_{k+1}^\wedge \triangleq \llbracket e[x := v] \rrbracket_k^\wedge \\
\llbracket \mathbf{let} x = e \mathbf{ in} f \rrbracket_{k+1}^\wedge \triangleq \llbracket e \rrbracket_k^\wedge \gg= (v \rightarrow \llbracket f[x := v] \rrbracket_k^\wedge) \\
\llbracket \mathbf{let} !a = !e \mathbf{ in} f \rrbracket_{k+1}^\wedge \triangleq \llbracket f[a := e] \rrbracket_k^\wedge \\
\llbracket \mathbf{op}(e_1, \dots, e_n) \rrbracket_{k+1}^\wedge \triangleq \llbracket \mathbf{op} \rrbracket(\llbracket e_1 \rrbracket_k^\wedge, \dots, \llbracket e_n \rrbracket_k^\wedge)
\end{array}$$

■ **Figure 4** Operational Semantics of $\Lambda^!$

4.4 Observational Equivalence

341 In order to compare $\Lambda^!$ -terms, we introduce the notion of *contextual equivalence* [49]. To do
342 so, we follow [65, 22] and postulate that once an observer executes a program, she can only
343 observe the effects produced by the evaluation of the program. For instance, in a pure (resp.
344 probabilistic) calculus one observes pure (resp. the probability of) convergence. Following
345 this postulate, we define an observation function $\mathbf{obs}^{\wedge*} : T(\mathcal{V}) \rightarrow T(1)$ as $T(!\gamma)$, where
346 $1 = \{*\}$ is the one-element set and $!\gamma : \mathcal{V} \rightarrow 1$ is the terminal arrow. As a consequence, we
347 see that $\mathbf{obs}^{\wedge*}$ is strict and continuous, so that we have, e.g., $\mathbf{obs}^{\wedge*}(\bigsqcup_k \xi_k) = \bigsqcup_k \mathbf{obs}^{\wedge*}(\xi_k)$.
348

349 ► **Example 8.** Notice that $T(1)$ indeed describes the observations one usually works with
350 in concrete calculi. For instance, $\mathcal{D}(1) \cong [0, 1]$, so that $\mathbf{obs}^{\wedge*}(\llbracket e \rrbracket)$ gives the probability of
351 convergence of e , and $\mathcal{M}(1) \cong \{\perp, \top\}$, so that $\mathbf{obs}^{\wedge*}(\llbracket e \rrbracket) = \top$ if and only if e converges.

352 In order to define contextual equivalence, we need to introduce the notion of a $\Lambda^!$ -context.
353 The latter is simply a $\Lambda^!$ -term with a single *linear* hole $[-]$ acting as a placeholder for a

354 computation (we regard a value v as the computation $\mathbf{val} v$). We do not give an explicit
 355 definition of contexts, the latter being standard.

356 ► **Definition 9.** Define contextual equivalence \equiv^{ctx} as follows:

$$357 \quad v \equiv^{\text{ctx}} w \iff \mathbf{val} v \equiv^{\text{ctx}} \mathbf{val} w \quad e \equiv^{\text{ctx}} f \iff \forall C. \text{obs}^{\Lambda^*} \llbracket C[e] \rrbracket = \text{obs}^{\Lambda^*} \llbracket C[f] \rrbracket.$$

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

362 5 Resource-Sensitive Semantics and Program Equivalence

363 The operational semantics of Section 4.3 is *resource-agnostic*, meaning that linearity *de facto*
 364 plays no role in the definition of the dynamics of a program. To overcome this deficiency, we
 365 endow $\Lambda^!$ with a resource-sensitive operational semantics: we give the latter by means of a
 366 suitable transition systems, which we dub resource transition systems. *Resource transition*
 367 *systems* (RTSs, for short) provide an operational semantics for $\Lambda^!$ -programs accounting for
 368 both their intensional and extensional behaviour. Those are defined as first-order transition
 369 systems in the spirit of [42], and generalise the Markov chains of [18].

370 5.1 Auxiliary Notions

371 In order to properly handle resources, it is useful to introduce some notation on sequences.
 372 Let S, S' be sequences over objects s_1, s_2, \dots . Unless ambiguous, we denote the concatenation
 373 of S and S' as S, S' . Moreover, for $S = s_1, \dots, s_k$ we denote by $|S| = k$ the length of
 374 S , and write $S[s]_i$, with $i \in \{1, \dots, k+1\}$, for the sequence obtained by inserting s in S
 375 at position i , i.e. the sequence $s_1, \dots, s_{i-1}, s, s_i, \dots, s_k$ of length $k+1$. Given a sequence
 376 $S = s_1, \dots, s_k$, we will form new sequences out of it by taking elements in S at given
 377 positions. If $\bar{c} = c_1, \dots, c_n$ is a sequence with elements in $\{1, \dots, k\}$ without repetitions,
 378 then we write $S_{\bar{c}}$ for the sequence s_{c_1}, \dots, s_{c_n} , and $S \ominus \bar{c}$ for the sequence obtained from S
 379 by removing elements in positions c_1, \dots, c_n . In order to preserve the order of S , we often
 380 consider sequences $\bar{c} = (c_1 < \dots < c_n)$ with $c_i \in \{1, \dots, k\}$. We call such sequences valid for
 381 S (although we should say valid for $|S|$).

382 5.1.0.1 System \mathcal{K}

383 The resource-sensitive operational semantics of $\Lambda^!$ is given by the RTS \mathcal{K} . Following [42],
 384 \mathcal{K} -states are defined as *configurations* $(\Gamma; \Theta)$, i.e. pairs of sequences of terms, where Γ is a
 385 (finite) sequence of (closed) computations and Θ is a (finite) sequence of (closed) terms in
 386 which only the last one need not be a value. To facilitate our analysis, we write $(\Gamma; \Delta; e)$
 387 if $\Theta = \Delta, e$, with Δ finite sequence of closed values and $e \in \Lambda$. Otherwise, we write $(\Gamma; \Delta)$,
 388 with Δ as above.

389 In a configuration $(\Gamma; \Delta; e)$ (and similarly in $(\Gamma; \Delta)$), Γ represents the non-linear resources
 390 available, which are (closed) computations: the environment can freely duplicate and evaluate
 391 them, as well as use them *ad libitum* to build arguments to be passed as input to other
 392 programs. Once a resource in Γ has been used, it *remains* in Γ , this way reflecting its
 393 non-linear nature. Dually, Δ represents the linear resources available, which are closed values.
 394 Values in Δ being closed, they are either abstractions or banged computations. In the latter
 395 case, the environment can take a value $!e$, unbang it, and put e in Γ . In the former case, the

environment can pass to a value $\lambda x.f$ an input argument made out of a context C (provided by the very environment) using values and computations in Γ, Δ . Since resources in Δ are linear, once they are used by C , they must be removed from Δ . Finally, the program e is the tested program. The environment can only evaluate it, possibly producing effects and values (linear resources). Once a linear resource v has been produced, it is put in Δ .

The calculus $\Lambda^!$ being typed, it is convenient to extend the notion of a type to configurations by defining a configuration type (notation α, β, \dots) as a pair of sequences $(\sigma_1, \dots, \sigma_n; \tau_1, \dots, \tau_m)$ of ordinary types. We say that a configuration $K = (\Gamma; \Theta)$ has type $\alpha = (\sigma_1, \dots, \sigma_n; \tau_1, \dots, \tau_m)$ (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 inhabited by configurations of the form $(\Gamma; \Delta)$) or a computation (so that the type will be inhabited by configurations of the form $(\Gamma; \Delta; e)$). To avoid this issue, we add a special label to the last type τ_m of the second component of a configuration type, this way specifying whether τ_m refers to a value or to a computation.

We denote by \mathcal{C}_α 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 $\text{supp}(\kappa)$ 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:

$$(\Gamma; \Delta; e) \xrightarrow{\text{eval}} \llbracket e \rrbracket \gg= (v \rightarrow \eta(\Gamma; \Delta, v)).$$

- 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 $a_1, \dots, a_n \mid x_1, \dots, x_m \vdash t$ as a (n, m) -(value/computation) context, depending on whether t is a value or a computation. Given sequences $\Gamma = e_1, \dots, e_n$, $\Delta = v_1, \dots, v_m$, 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

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

$$435 \frac{t \text{ (} n, m \text{)-value context } \quad |\bar{i}| = n, |\bar{j}| = m \quad \bar{i}, \bar{j} \text{ valid for } \Gamma, \Delta}{(\Gamma; \Delta[\lambda x.f]_l) \xrightarrow{(\bar{i}, \bar{j}, l, t)} \eta(\Gamma; \Delta \ominus \bar{j}; f[x := t[\Gamma_{\bar{i}}, \Delta_{\bar{j}}]])}$$

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

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

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

$$\begin{array}{ll} (\Gamma; \Delta; e) \xrightarrow{eval} \llbracket e \rrbracket \gg v \rightarrow \eta(\Gamma; \Delta, v) & (\Gamma; \Delta[!e]_l) \xrightarrow{?_l} \eta(\Gamma, e; \Delta). \\ (\Gamma[e]_l; \Delta) \xrightarrow{!_l} \eta(\Gamma[e]_l; \Delta; e) & (\Gamma; \Delta[\lambda x.f]_l) \xrightarrow{(\bar{i}, \bar{j}, l, t)} \eta(\Gamma; \Delta \ominus \bar{j}; f[x := t[\Gamma_{\bar{i}}, \Delta_{\bar{j}}]]) \end{array}$$

■ **Figure 5** Transition rules for \mathcal{K}

438 ► **Remark 11.** Notice that given $K \in \mathcal{C}_\alpha$, K can always make a α -transition, this way making
 439 its type visible. Additionally, we see that the transition structure of \mathcal{K} is *type-driven*. That
 440 is, given a configuration $K \in \mathcal{C}_\alpha$ and a \mathcal{K} -action ℓ , α and ℓ alone determine whether K
 441 can make an ℓ -transition. Moreover, if that is the case, then there is a unique κ such that
 442 $K \xrightarrow{\ell} \kappa$. Besides, $\kappa \in T(\mathcal{C}_\beta)$ for some configuration type β which is *uniquely* determined by
 443 ℓ and α . That is, there is a *partial* function \mathbf{b} from configuration types and actions such that
 444 if $\mathbf{b}(\alpha, \ell)$ is defined and $K \in \mathcal{C}_\alpha$, then $K \xrightarrow{\ell} \kappa$ with $\kappa \in T(\mathcal{C}_{\mathbf{b}(\alpha, \ell)})$. From now on, we write
 445 $\mathbf{b}(\alpha, \ell) = \beta$ to mean that $\mathbf{b}(\alpha, \ell)$ is defined and equal β . As a consequence, we have the rule:

$$446 K \in \mathcal{C}_\alpha \wedge \mathbf{b}(\alpha, \ell) = \beta \implies \exists! \kappa \in T(\mathcal{C}_\beta). K \xrightarrow{\ell} \kappa.$$

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

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

456 We are interested in observing the behaviour of \mathcal{K} -states on those traces that are coherent
 457 with their type. Therefore, given a \mathcal{K} -state K , we define the set $Tr(K)$ of its traces by
 458 stipulating that $\varepsilon \in Tr(K)$, for any K , and that $\ell \cdot \mathbf{u} \in Tr(K)$ whenever $K \xrightarrow{\ell} \kappa$, for some
 459 monadic configuration κ , and $\mathbf{u} \in Tr(L)$, for any $L \in \text{supp}(\kappa)$. Notice that the latter clause

460 is meaningful, since $Tr(K)$ is actually determined by the type of K (rather than by K itself),
 461 and if $K \xrightarrow{\ell} \kappa$, then all configurations in the support of κ have the same type.

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

$$464 \quad \mathbf{st}(K, \varepsilon) \triangleq \eta(*); \quad \mathbf{st}(K, \ell \cdot \mathbf{u}) \triangleq \kappa \gg= (L \rightarrow \mathbf{st}(L, \mathbf{u})) \text{ where } K \xrightarrow{\ell} \kappa.$$

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

► **Definition 14.** The relation $\simeq_{\text{tr}}^{\mathcal{K}}$ on \mathcal{K} -states is thus defined:

$$K \simeq_{\text{tr}}^{\mathcal{K}} L \iff Tr(K) = Tr(L) \wedge \forall \mathbf{t} \in Tr(K). \mathbf{st}(K, \mathbf{t}) = \mathbf{st}(L, \mathbf{t})$$

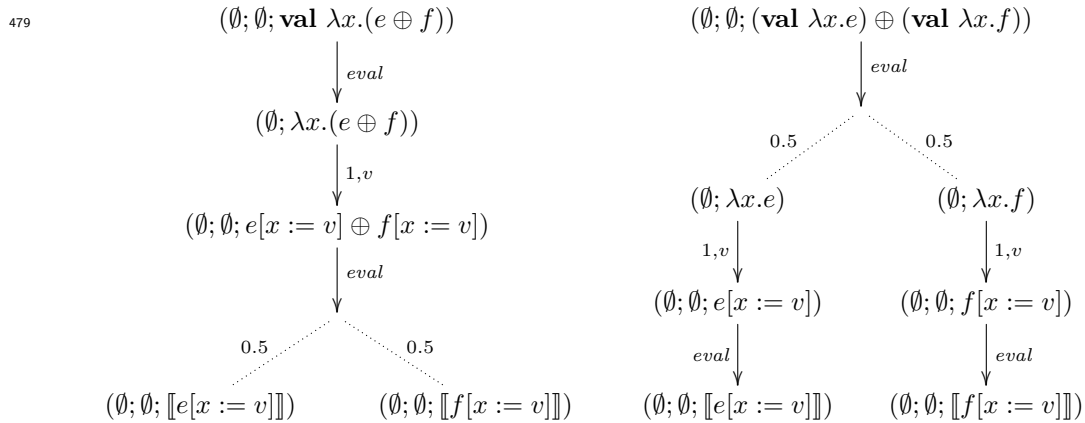
470 We extend the action of $\simeq_{\text{tr}}^{\mathcal{K}}$ to Λ^1 -terms by regarding a computation e as the configuration
 471 $(\emptyset; \emptyset; e)$, and a value v as the computation $\mathbf{val} v$. We denote the resulting notion $\simeq_{\text{tr}}^{\Lambda}$.

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

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

$$\mathbf{val} \lambda x.(e \oplus f) \simeq_{\text{tr}}^{\Lambda} (\mathbf{val} \lambda x.e) \oplus (\mathbf{val} \lambda x.f).$$

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



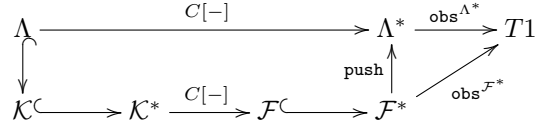
480 In light of Theorem 17, we can then conclude $g \equiv^{\text{ctx}} h$. Next, we prove that such an
 481 equivalence is only linear: $\mathbf{val} !(e \oplus f) \not\equiv^{\text{ctx}} (\mathbf{val} !e) \oplus (\mathbf{val} !f)$. For that, it is sufficient to
 482 instantiate e and f as the identity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

486 **5.2 Full Abstraction of Trace Equivalence**

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

494 ► **Proposition 16.** $\simeq_{\text{tr}}^{\Lambda} \subseteq \equiv^{\text{ctx}}$.

To prove Proposition 16, we have to show that if $e \simeq_{\text{tr}}^{\Lambda} f$, then we have $\text{obs}^{\Lambda^*} \llbracket C[e] \rrbracket^{\Lambda} = \text{obs}^{\Lambda^*} \llbracket C[f] \rrbracket^{\Lambda}$, 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.



495 Since $\simeq_{\text{tr}}^{\Lambda}$ is defined in terms of $\simeq_{\text{tr}}^{\mathcal{K}}$, we consider configurations— \mathcal{K} -states—and contexts for
 496 them, where a context for a \mathcal{K} -state K is just a standard multiple-holes context whose holes
 497 have to be filled with with terms in K . The first step of our strategy is the *determinization*
 498 of \mathcal{K} . This is achieved by lifting the state space of \mathcal{K} from configurations to monadic
 499 configurations. The dynamics of \mathcal{K} is then lifted relying on the (strong) monad structure of T
 500 in a standard way [22]. We call the resulting system \mathcal{K}^* . The advantage of working with \mathcal{K}^*
 501 is that \mathcal{K}^* -bisimilarity and \mathcal{K}^* -trace equivalence coincide, \mathcal{K}^* being deterministic. In general,
 502 most of the transition systems we rely on can be ultimately described as systems $\mathcal{S} = (X, \delta)$
 503 made of a state space X and a dynamics $\delta : X \rightarrow T(X)^A$, for some set A of actions. The
 504 determinization of \mathcal{S} , which we usually denote by \mathcal{S}^* , has $T(X)$ as state space and dynamics
 505 $\delta^* : T(X) \rightarrow T(X)^A$ defined as the strong Kleisli extension of δ (modulo (un)currying).

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

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

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

$$\begin{array}{ccccccccc}
 \Lambda & \hookrightarrow & \mathcal{K}^{\mathcal{C}} & \longrightarrow & \mathcal{K}^* & \xrightarrow{\mathcal{C}[-]} & \mathcal{F}^{\mathcal{C}} & \longrightarrow & \mathcal{F}^* & \xrightarrow{\text{obs}^{\mathcal{F}^*}} & T1 \\
 \simeq_{\text{Tr}}^{\Lambda} \downarrow & & \simeq_{\text{Tr}}^{\mathcal{K}} \downarrow & & \simeq_{\text{Tr}}^{\mathcal{K}^*} \downarrow & & \mathcal{C}(\simeq_{\text{Tr}}^{\mathcal{K}^*}) \downarrow & & \text{BC}(\simeq_{\text{Tr}}^{\mathcal{K}^*}) \downarrow & & = \downarrow \\
 \Lambda & \hookrightarrow & \mathcal{K}^{\mathcal{C}} & \longrightarrow & \mathcal{K}^* & \xrightarrow{\mathcal{C}[-]} & \mathcal{F}^{\mathcal{C}} & \longrightarrow & \mathcal{F}^* & \xrightarrow{\text{obs}^{\mathcal{F}^*}} & T1
 \end{array}$$

522 Here, $\mathcal{C}(R)$ denotes the contextual closure of R , whereas $\text{B}(R)$ is the Barr extension of R
 523 [7, 37]. Finally, the map $\text{obs}^{\mathcal{F}^*}$ is obtained postcomposing the observation map obs with
 524 **push**. Let us now move to full abstraction.

525 ► **Theorem 17.** $\equiv^{\text{ctx}} = \simeq_{\text{Tr}}^{\Lambda}$.

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

529 6 Conclusion and Future Work

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

534 Although the present paper focuses on linearity (and effects), the authors believe that
 535 resource transition systems can be extended to deal with finer notions of context dependence
 536 such as *structural coeffects* [51, 29, 14, 50]. To do so, one should modify resource transition
 537 systems by considering sequences of terms indexed by elements of a resource algebra (the
 538 latter being a preordered semiring), and let transitions update resources. Thus, for instance,
 539 from a sequence $(\Gamma, \langle e \rangle_{r+1}, \Delta)$, meaning that e is available according to the resource $r + 1$, we
 540 have a transition to $(\Gamma, \langle e \rangle_r, \Delta; e)$. The authors also believe that resource transition systems
 541 can be used to generalise Crubillé and Dal Lago probabilistic program metric to arbitrary
 542 algebraic effects. To do so, one would simply replace ordinary relations with relations taking
 543 values over quantales [30].

544 Finally, as a long term future work, the authors would like to study whether the ideas
 545 presented in this paper can be adapted to deal with quantum languages [62, 63], where the
 546 interaction between linearity and effects plays a central role. In fact, although we have not
 547 discussed tensor product types (which play a crucial role in a quantum setting), it is not
 548 hard to see that resource transition systems can be extended to deal with such types [17].

549 6.1 Related Work

550 This is not the first work on operationally-based notions of program equivalence for linear
 551 calculi. In particular, notions of equivalences have been defined by means of logical relations
 552 by Bierman, Pitts, and Russo [11], of applicative bisimilarity by Bierman [10] and Crole²

¹ Each square gives a set-theoretic inclusion. For instance, the leftmost square states that $\simeq_{\text{Tr}}^{\Lambda} \subseteq \simeq_{\text{Tr}}^{\mathcal{K}}$.

² Crole's applicative bisimilarity, however, does not deal with copying.

553 [15], of trace equivalence by Deng and Zhang [27, 26], as well as of a number of possible
 554 worlds-indexed equivalences (e.g. [2, 36]). As already remarked, one of the advantages of
 555 resource transition systems (and their associated trace equivalence) compared, e.g., with
 556 logical relations, is that they they provide a *first-order* account of program equality.

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

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

573 ——— References ———

- 574 1 Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in*
 575 *Functional Programming*, pages 65–117. Addison Wesley, 1990.
- 576 2 Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3 : A linear language with locations.
 577 *Fundam. Informaticae*, 77(4):397–449, 2007.
- 578 3 Andrew W. Appel and Daddiv A. McAllester. An indexed model of recursive types for
 579 foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.
- 580 4 Robert Atkey. Syntax and semantics of quantitative type theory. In *Proc. of LICS 2018*, pages
 581 56–65, 2018.
- 582 5 Hendrik P. Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and
 583 the foundations of mathematics. North-Holland, 1984.
- 584 6 Hendrik P. Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*.
 585 Perspectives in logic. Cambridge University Press, 2013.
- 586 7 Michael Barr. Relational algebras. *Lect. Notes Math.*, 137:39–55, 1970.
- 587 8 Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Proc. of*
 588 *LICS 1996*, pages 420–431, 1996.
- 589 9 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and
 590 Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language.
 591 *PACMPL*, 2(POPL):5:1–5:29, 2018.
- 592 10 Gavin M. Bierman. Program equivalence in a linear functional language. *J. Funct. Program.*,
 593 10(2):167–190, 2000.
- 594 11 Gavin M. Bierman, Andrew M. Pitts, and Claudio V. Russo. Operational properties of lily,
 595 a polymorphic linear lambda calculus with recursion. *Electr. Notes Theor. Comput. Sci.*,
 596 41(3):70–88, 2000.
- 597 12 Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care:
 598 relational interpretation of algebraic effects and handlers. *PACMPL*, 2(POPL):8:1–8:30, 2018.

³ Besides, notice that bisimilarity being sensitive to branching, it naturally invalidates (λ -dist).

- 599 13 Ales Bizjak and Lars Birkedal. Step-indexed logical relations for probability. In *Proc. of*
600 *FOSSACS 2015*, pages 279–294, 2015.
- 601 14 Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative
602 coefficient calculus. In *Proc. of ESOP 2014*, pages 351–370, 2014.
- 603 15 Roy L. Crole. Completeness of bisimilarity for contextual equivalence in linear theories. *Logic*
604 *Journal of the IGPL*, 9(1):27–51, 2001.
- 605 16 Raphaëlle Crubillé and Ugo Dal Lago. On probabilistic applicative bisimulation and call-by-
606 value lambda-calculi. In *Proc. of ESOP 2014*, pages 209–228, 2014.
- 607 17 Raphaëlle Crubillé and Ugo Dal Lago. Metric reasoning about lambda-terms: The affine case.
608 In *Proc. of LICS 2015*, pages 633–644, 2015.
- 609 18 Raphaëlle Crubillé and Ugo Dal Lago. Metric reasoning about lambda-terms: The general
610 case. In *Proc. of ESOP 2017*, pages 341–367, 2017.
- 611 19 Ugo Dal Lago and Francesco Gavazzo. Effectful normal form bisimulation. In *Proc. of ESOP*
612 *2019*, pages 263–292, 2019.
- 613 20 Ugo Dal Lago and Francesco Gavazzo. Resource transition systems and full abstraction for
614 linear higher-order effectful programs (extended version). 2021. URL: [http://www.cs.unibo.](http://www.cs.unibo.it/~dallago/resbranch.pdf)
615 [it/~dallago/resbranch.pdf](http://www.cs.unibo.it/~dallago/resbranch.pdf).
- 616 21 Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. Effectful applicative bisimilarity:
617 Monads, relators, and howe’s method. In *Proc. of LICS 2017*, pages 1–12, 2017.
- 618 22 Ugo Dal Lago, Francesco Gavazzo, and Ryo Tanaka. Effectful applicative similarity for
619 call-by-name lambda calculi. *Theor. Comput. Sci.*, 813:234–247, 2020.
- 620 23 Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. In *Proc. of TLCA 2009*,
621 pages 80–94, 2009.
- 622 24 Ugo Dal Lago, Davide Sangiorgi, and Michele Alberti. On coinductive equivalences for
623 higher-order probabilistic functional programs. In *Proc. of POPL 2014*, pages 297–308, 2014.
- 624 25 Ugo Dal Lago and Margherita Zorzi. Probabilistic operational semantics for the lambda
625 calculus. *RAIRO - Theor. Inf. and Applic.*, 46(3):413–450, 2012.
- 626 26 Yuxin Deng and Yuan Feng. Bisimulations for probabilistic linear lambda calculi. In *Proc. of*
627 *TASE 2017*, pages 1–8, 2017.
- 628 27 Yuxin Deng and Yu Zhang. Program equivalence in linear contexts. *Theor. Comput. Sci.*,
629 585:71–90, 2015.
- 630 28 Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. Linear-use CPS translations in the
631 enriched effect calculus. *Logical Methods in Computer Science*, 8(4), 2012.
- 632 29 Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvert, and Tarmo
633 Uustalu. Combining effects and coeffects via grading. In *Proc. of ICFP 2016*, pages 476–489,
634 2016.
- 635 30 Francesco Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs:
636 Applicative distances. In *Proc. of LICS 2018*, pages 452–461, 2018.
- 637 31 Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Proc. of*
638 *ESOP 2014*, pages 331–350, 2014.
- 639 32 J-Y. Girard, A. Scedrov, and P.J. Scott. Bounded linear logic: A modular approach to
640 polynomial-time computability. *Theor. Comput. Sci.*, 97:1–66, 1992.
- 641 33 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
- 642 34 Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor.
643 *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.
- 644 35 Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for
645 algebraic effects. In *Proc. of LICS 2010*, pages 209–218. IEEE Computer Society, 2010.
- 646 36 Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and
647 dependent types. In *Proc. of POPL 2015*, pages 17–30, 2015.
- 648 37 Alexander Kurz and Jiri Velebil. Relation lifting, a survey. *J. Log. Algebr. Meth. Program.*,
649 85(4):475–499, 2016.

- 650 38 Søren B. Lassen. Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up
651 to context. *Electr. Notes Theor. Comput. Sci.*, 20:346–374, 1999.
- 652 39 Søren B. Lassen. Eager normal form bisimulation. In *Proceedings of LICS 2005*, pages 345–354,
653 2005.
- 654 40 Paul B. Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value
655 programming languages. *Inf. Comput.*, 185(2):182–210, 2003.
- 656 41 Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
- 657 42 Jean-Marie Madiot, Damien Pous, and Davide Sangiorgi. Bisimulations up-to: Beyond
658 first-order transition systems. In *Proc. of CONCUR 2014*, pages 93–108, 2014.
- 659 43 Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *J.*
660 *Funct. Program.*, 1(3):287–327, 1991.
- 661 44 Cristina Matache and Sam Staton. A sound and complete logic for algebraic effects. In *Proc.*
662 *of FOSSACS 2019*, pages 382–399, 2019.
- 663 45 Nicholas D. Matsakis and Felix S. Klock II. The rust language. In *Proceedings of the 2014*
664 *ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland,*
665 *Oregon, USA, October 18-21, 2014*, pages 103–104, 2014.
- 666 46 Rasmus Ejlers Møgelberg and Sam Staton. Linear usage of state. *Logical Methods in Computer*
667 *Science*, 10(1), 2014.
- 668 47 Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of LICS 1989*, pages
669 14–23. IEEE Computer Society, 1989.
- 670 48 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
- 671 49 J. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1969.
- 672 50 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program
673 reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP):110:1–110:30, 2019.
- 674 51 Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-
675 dependent computation. In *Proc. of ICFP 2014*, pages 123–135, 2014.
- 676 52 Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical*
677 *Structures in Computer Science*, 10(3):321–359, 2000.
- 678 53 Marinus J. Plasmeijer. CLEAN: a programming environment based on term graph rewriting.
679 *Electr. Notes Theor. Comput. Sci.*, 2:215–221, 1995.
- 680 54 Gordon Plotkin. Lambda-definability and logical relations. Technical Report SAI-RM-4,
681 School of A.I., University of Edinburgh, 1973.
- 682 55 Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In *Proc. of FOSSACS*
683 *2001*, pages 1–24, 2001.
- 684 56 Gordon D. Plotkin and John Power. Semantics for algebraic operations. *Electr. Notes Theor.*
685 *Comput. Sci.*, 45:332–345, 2001.
- 686 57 Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Proc. of*
687 *FOSSACS 2002*, pages 342–356, 2002.
- 688 58 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied*
689 *Categorical Structures*, 11(1):69–94, 2003.
- 690 59 John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages
691 513–523, 1983.
- 692 60 Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for
693 higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011.
- 694 61 Davide Sangiorgi and Valeria Vignudelli. Environmental bisimulations for probabilistic higher-
695 order languages. In *Proceedings of POPL 2016*, pages 595–607, 2016.
- 696 62 Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical
697 control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.
- 698 63 Peter Selinger and Benoît Valiron. A linear-non-linear model for a computational call-by-value
699 lambda calculus (extended abstract). In *Proc. of FOSSACS 2008*, pages 81–96, 2008.

- 700 **64** Kurt Sieber. Reasoning about sequential functions via logical relations. In Johnstone P. T.
701 Fourman, M. P. and A. M. Pitts, editors, *Applications of Categories in Computer Science*,
702 volume 177 of *London Mathematical Society Lecture Note Series*, pages 258–269. Cambridge
703 University Press, 1992.
- 704 **65** Alex Simpson and Niels Voorneveld. Behavioural equivalence via modalities for algebraic
705 effects. In *Proc. of ESOP 2018*, pages 300–326, 2018.
- 706 **66** David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theor. Comput.*
707 *Sci.*, 227(1-2):231–248, 1999.
- 708 **67** Philip Wadler. Linear types can change the world! In *Programming concepts and methods*,
709 *1990*, page 561, 1990.
- 710 **68** Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*,
711 *First International Spring School on Advanced Functional Programming Techniques, Båstad*,
712 *Sweden, May 24-30, 1995, Tutorial Text*, pages 24–52, 1995.