Lazy Intermediate Representations for Algebraic Effects

Simon Castellan
Inria, Univ. Rennes, CNRS, IRISA, France

Hugo Paquet
Inria, École Normale Supérieure – PSL University, CNRS, France

Abstract

A lazy program interpreter postpones computation until the result is actually needed. This is typically more efficient than an eager (or call-by-value) interpreter, but a concern is that the semantics is not generally preserved.

We propose a new semantic analysis of lazy evaluation that relies on a subtle combination of name generation and read-only state. Our perspective is that laziness arises from a hybrid evaluation strategy, in which only the name generation follows call-by-value.

This semantic model suggests better intermediate representations of sum and product types in a lazy interpreter, along with equations that justify further optimizations. We illustrate this with an implementation in OCaml. Our motivation is practical: the origin of this work is a real-world application of discrete probabilistic programming, in which large algebraic data types cause significant performance issues with a call-by-value interpreter. Our lazy semantics justifies better optimized representations, and provides principled foundations for other methods involving laziness in probabilistic programming.

2012 ACM Subject Classification Theory of computation → Categorical semantics; Theory of computation → Denotational semantics; Theory of computation → Probabilistic computation

Keywords and phrases Categorical semantics, lazy evaluation, interpreter, probabilistic programming

Digital Object Identifier 10.4230/LIPIcs.LICS.2026.80

Acknowledgements We thank Aurore Alcolei and Tom Hirschowitz for their deep involvement in this project, and for many helpful discussions. We are also grateful for conversations with John M. Li, Jack Czenszak, and Steven Holtzen: our paper shares many ideas with their work [29], to appear at PLDI 2026; understanding their perspective has helped us improve the final version of this paper.We acknowledge support from the grant ANR Flores (ANR-23-SSAI-0001).

1 Introduction

The lazy paradigm of program evaluation consists in deferring any computation until the result is needed. Laziness can provide a crucial efficiency gain, but for programs with side-effects, it also brings significant semantic complexity.

We propose a new semantic analysis of laziness, which aims to reduce the gap between the abstract semantics of laziness and an efficient lazy program interpreter. More concretely, we consider lazy evaluation for a functional language with expressive data types and an algebraic signature of effects. Our main contribution is a new categorical semantics, derived from a particular combination of monads, which exposes the mechanisms of lazy evaluation and justifies structured intermediate representations, appropriate for an efficient lazy interpreter.

In this introduction we give a sense of what efficient laziness involves for algebraic effects. Our running example, which we discuss next, is inspired by a real-world system: a botanical knowledge base that relies on discrete probabilistic programming and complex algebraic data types.

© Simon Castellan and Hugo Paquet;
licensed under Creative Commons License CC-BY 4.0
41st Annual Symposium on Logic in Computer Science (LICS 2026).
Editors: Claudia Faggian and Joost-Pieter Katoen; Article No. 80; pp. 80:1–80:27
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 The need for laziness: a practical example

At a high level, the Botascopia project [7, 8] is a large-scale database of plant species that makes available a range of tools for identifying and reasoning about plants. The goal is to facilitate the transfer of botanical knowledge between diverse communities of users.

Of interest to this paper is the practical challenge that arises when mediating between different descriptions of the same plant (e.g. the scientific description of a plant *vs.* a vernacular description for a lay audience). Botascopia aims to do this kind of translation in a systematic way, as we explain now.

Plant species as probabilistic programs. In Botascopia, each species of plants is formally encoded as a probabilistic program that returns an element of a large algebraic data type `botany`. The probability coefficients capture possible variation within the species (a concept known as 'species polymorphism').[1]

The type `botany` encodes all possible traits and features of a plant. It is a complex algebraic data type, built out of sums and products (i.e. tagged unions and records). We now look at a small simplified fragment, concerned with the botanical concept of inflorescence.

```
type color = Red | White | Yellow
type numPetals = 1 | 2 | ... | 10 | 11 | 12
type flower = { c: color ; n: numPetals }
(* A daisy is a capitulum *)
type inflorescence =
  Capitulum of { ligules: flower option; tubes: flower option }
| Solitary of flower
type botany = { inflorescence; ... }
```

An inflorescence is either a solitary flower, or a 'capitulum': a cluster of small flowers appearing as a single flower, made up of either 'ligulate' (petal-looking) or 'tubulate' (tube-shaped) flowers, or both. An example inflorescence is the `pilewort`, a solitary yellow flower with a number of petals uniformly distributed between 7 and 12.

```
pilewort = Solitary { c = Yellow; n = rand(7,12) }
```

The challenge of probabilistic translation. Since a non-specialist may not know about inflorescences and capitula, Botascopia provides a simplified type `vernacular` as a substitute for `botany`, together with simplification function `botany -> vernacular`. We display the fragment for inflorescences.

```
type vernacular = { flower; ... }
```

```
let translation (i: inflorescence): flower =
 match i with
| Solitary x -> x
| Capitulum {ligules = None; tubes = Some x} -> x
| Capitulum {ligules = Some x; tubes = None} -> x
| Capitulum {ligules = Some x; tubes = Some y} ->
  { n = x.number; c = if flip then x.c else y.c }
```

[1] This variation could refer to possible differences between individuals of the same species (some *Primula vulgaris* have white flowers, others have yellow flowers), or distinct traits appearing on a single individual (e.g. leaves of a different shape depending on whether they appear on the stem or the basal).

The translation is itself probabilistic. This is to represent a fuzzy understanding: for a capitulum composed of both ligulate and tubulate flowers, a non-specialist will typically count the petals of ligulate flowers, but may pick either color as the main color of the capitulum.

The challenge is to compute the outcome distribution of the program `translation pilewort`. In a call-by-value (or *eager*) probabilistic language the evaluation calls `translation` for every possible deterministic value of `pilewort` (e.g. `Solitary{ c = Yellow; n = 9 }`) even though all executions follow the same control flow path. Worse, the number of calls to `translation` will grow exponentially with the number of fields in the description record, which is a serious limitation: the current `botany` type in use in Botascopia has nearly a thousand traits and most species exhibit some form of polymorphism.

On the other hand, a lazy interpreter for the language does not need to enumerate all possible executions in this way. Instead, it observes that any randomness in `pilewort` is nested inside the constructor `Solitary`, and evaluates `translation pilewort` in constant time by immediately returning the underlying `flower`.

In this paper we give a semantic analysis of this kind of behaviour. We show that a lazy probabilistic interpreter computes the correct outcome distribution in all cases, provided appropriate intermediate representations are used when probabilistic choice is nested in or interleaved with type constructors.

1.2 Semantic issues around effects and laziness

Passing around computations instead of values affects the semantics. Consider the term $\texttt{let}\, x = \texttt{flip}\, \texttt{in}\, \langle x, x\rangle$ that draws a random boolean and returns the result duplicated in a pair. Only correlated pairs $\langle \texttt{True}, \texttt{True}\rangle$ and $\langle \texttt{False}, \texttt{False}\rangle$ should be observable, but by naively substituting the whole computation `flip` for x we end up with independent draws.

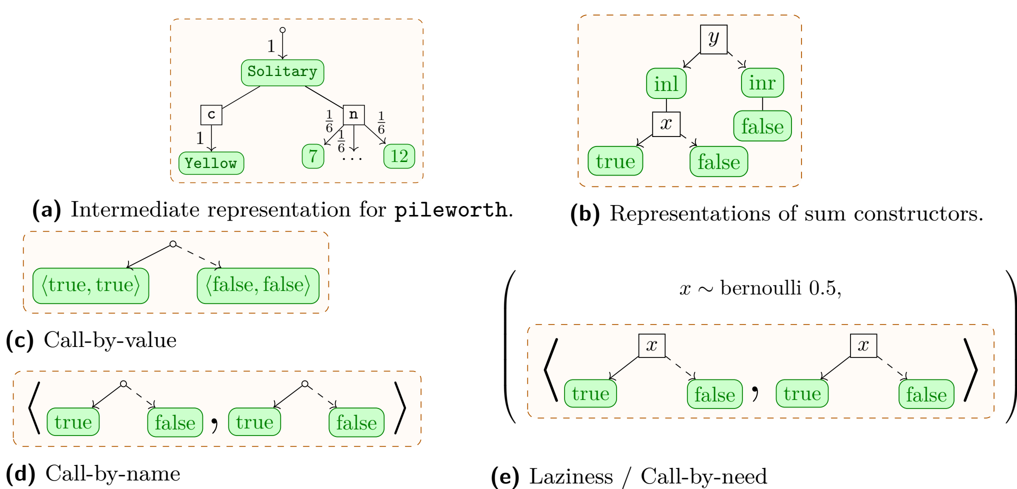
This example is a typical illustration of the difference between call-by-value and call-by-name. This is well-understood semantically [44]. Our interest in this paper is in lazy evaluation (a.k.a. *call-by-need*), which in some sense lies in between: for the duplicated coin flip, lazy evaluation agrees with call-by-value; however for a program like $\texttt{fst}\ \langle \texttt{print 'a'}, \texttt{print 'b'}\rangle$ it agrees with call-by-name and only `a` is printed. (The result of the right-hand computation is not *needed*.)

We emphasize that, for a practical language with effects, a fully lazy evaluation of this kind is perhaps inappropriate, despite the efficiency gain, because of the semantic unpredictability. (Haskell is a popular lazy language, but built-in monads typically enforce call-by-value behaviour by default, and the language offers operators for overriding laziness.)

However, lazy evaluation shines for probabilistic programming, where it is used heavily in practice [11, 4, 13, 36]. Our model makes precise the folklore understanding that laziness is safe for any affine effects, such as probabilistic choice, for which removing “unused” parts of a program leaves the semantics unchanged.

1.3 Correct intermediate representations

There is a significant gap between the folklore semantics of laziness (using affine monads, see §3.1) and the implementation of a lazy interpreter. A key contribution of this paper is a new semantic model that exposes the basic operations and intermediate representations involved in a realistic lazy interpreter. To illustrate these representations we return briefly to the Botascopia example.



(a) Intermediate representation for `pileworth`.

(b) Representations of sum constructors.

(c) Call-by-value

(d) Call-by-name

(e) Laziness / Call-by-need

Figure 1 Program representations and evaluation strategies.

Tree-like diagrams for probabilistic computations. The computation corresponding to `pilewort` can be represented as a tree-like structure that interleaves type constructors and probabilistic branching (Figure 1a). There are two kinds of nodes and edges in this tree: green constructor nodes, corresponding to sum type constructors, whose incoming edges labelled with probability; and product (or field) nodes, corresponding to a field in a record type (or a component in a pair) and with incoming undirected arrow without probability.

This representation follows the type structure of `inflorescence`, and in fact arises automatically from the distribution monad in the call-by-name model of probabilistic programming (§4.5). A language interpreter relying on this kind of representation is easy to write, following the semantic model, but will suffer from the semantic issues outlined in §1.2.

Lazy interpretation using choice names. We propose to adapt this representation for a lazy interpreter. The key insight is to generate a name for each call to a random primitive and to refer to these names in the tree structure. In other words probability is considered as a combination of two effects: name generation and state access, where the state consists of a set of names pointing to basic distributions.[2] To illustrate this informally, we contrast the lazy representation based on named variables with the usual representations of call-by-value and call-by-name for the program $\texttt{let}\, x = \texttt{flip}\, \texttt{in}\, \langle x, x\rangle$ in Figure 1c, 1d, 1e. Under the lazy representation, the two booleans are appropriately correlated because they are determined by a single value x, and the representation allows for a faster computation of the marginals.

Decision diagrams and types. This idea leads us to a sophisticated form of multi-valued decision diagrams, with n-ary internal branching (§4.4). As the example of Figure 1 suggests, programs of product type are efficiently represented as families of these structures. Meanwhile sum types require diagrams whose leaves refer to a type constructor and point to another diagram representing the nested computation. An example is given in Figure 1b for the term $(\texttt{inl}\ (\texttt{True}) +_x \texttt{inl}\ (\texttt{False})) +_y \texttt{inr}\ (\texttt{False})$ where the node for $\texttt{inl}\ (\cdot)$ has been pulled

[2] The idea is not at all new to this paper: Dal Lago, Guerrieri and Heijltjes [10] take this decomposition seriously in the context of probabilistic λ-calculus, and the line of work [13, 36, 29] is about applying similar representations for efficient probabilistic programming; we discuss related work in detail in §7.

up above the node for x (the syntax assumes existing variables x and y, *cf.* §4.2). This is a flexible structure, that allows for arbitrary interleaving of effects and type constructors and makes pattern matching and projections significantly more efficient. The key insight in this paper is that this arises naturally from a semantic model of laziness (§1.4, §5.1).

General algebraic effects. We have motivated the representations using simple probabilistic branching just because lazy probabilistic programming already has a lot of practical significance. However the mathematical theory is not restricted to this particular effect. As we will see, our models support an arbitrary (finitary) signature of algebraic effects, without any commutativity or affinity requirements, although correctness with respect to the call-by-value semantics requires affinity. We show (§3) how to adapt name generation and state to this general setting.

1.4 Intermediate representations via monads for name generation and read-only state

More formally, our construction is based on two separate monads, inspired by models of name generation and read-only state.

The monad N for name generation requires some slightly advanced categorical machinery to track a dynamic set of names. We introduce this method in §3, adapting existing methods to our requirements: here each name must correspond to an algebraic operation in the effect signature, and we must also track any previously observed value.

The monad R for state access is a simple reader monad, parametrized by the current set of names. (Reading is sufficient for our purposes: no writing to the state is ever necessary beyond the generation of new names, which is handled by the first monad.) We show that the monads R and N compose using a distributive law ([3], §3.3).

With the two monads in place, we explore several possible representations for laziness. The main perspective of this paper is that each intermediate representation corresponds to a choice of evaluation strategy for the reader monad R. To preserve correctness, name generation must follow a strict call-by-value strategy, but varying the evaluation order for R has no impact on the semantics, because R is *cartesian* (both affine and relevant, see §4).

This provides some flexibility for writing a lazy interpreter. There are canonical representations corresponding to call-by-value and call-by-name strategies for R, and we further show that a third alternative exists that combines the best of both worlds. This is possible because the category of algebras for R has very rich structure (§4.3).

1.5 Summary of contributions

In summary, we provide a semantic model of laziness that can readily be turned into a realistic program interpreter. To achieve this we combine known semantic methods in a new way.

In **Section 2** we define a simple calculus with sum types, product types, and algebraic effects, and recall its call-by-value semantics.

In **Section 3** we introduce models for laziness. For a fixed signature of effects, we define the two monads N (for allocating new names) and R (for read-only state), and introduce the categorical machinery necessary for combining them.

Section 4 is specifically about the monad R. We show how its category of algebras can be used as a flexible language of intermediate representations for laziness. We prove and exploit

a general result: for 'cartesian' monads (Def. 16) call-by-name and call-by-value semantics yield comparable results. As we explain, this justifies the correctness of our representations.

In **Section 5**, we bring N into the results of §4. This provides a design for a lazy interpreter, which we describe using a realistic fragment of OCaml code. We deduce its correctness with respect to the call-by-value semantics, for affine effects.

Section 6 extends all the above results to a language with higher-order types. This is mostly straightforward, although correctness requires a technical proof via realizability and double orthogonality.

In **Section 7** we conclude by discussing related and further work.

2 Algebraic effects and eager semantics

We start by introducing a simple language equipped with a set of effectful operations defined by an algebraic signature (following [18, 41]). We then review its call-by-value semantics using monads [35]. For the moment we focus on the first-order fragment, to demonstrate our approach to laziness and the representations of sum and product types. We will consider an expressive language with higher-order types in §6.

2.1 A first-order language with effects

The starting point for algebraic effects is a signature of operations.

▶ **Definition 1.** *A* signature *consists of a set* Σ *equipped with a function* $\mathrm{ar} : \Sigma \to \mathbb{N}$*. For* $\sigma \in \Sigma$*, the integer* $\mathrm{ar}(\sigma)$ *is the* arity *of the operation* σ*.*

For discrete probabilistic programming, a possible signature is $\Sigma_{\mathrm{P}} = \{+_p \mid p \in [0,1]\}$, where every operation has arity 2.

Our programming language of study is a first-order functional language with sum and product types, parametrized by a signature Σ of operations.

$$\begin{array}{rcl} A, B & ::= & 1 \mid A \times B \mid A + B \\ M, N & ::= & \texttt{()} \mid \texttt{fst}\ M \mid \texttt{snd}\ M \mid \langle M, N \rangle \\ & \mid & \texttt{inl}\ M \mid \texttt{inr}\ M \mid \texttt{case}\ M : [x.N_1 \mid y.N_2] \\ & \mid & \texttt{let}\ x = M\ \texttt{in}\ N \mid x \\ & \mid & \sigma(M_1, \dots, M_k) \qquad (\sigma \in \Sigma, \mathrm{ar}(\sigma) = k) \end{array}$$

The type system is completely standard: contexts are lists of the form $\Gamma = x_1 : A_1, \dots, x_n : A_n$ and judgements are of the form $\Gamma \vdash M : A$. There are introduction and elimination rules for products and sums, and also a rule $\Gamma \vdash \texttt{()} : 1$. Finally we have the following rule for every algebraic operation $\sigma \in \Sigma$ with $\mathrm{ar}(\sigma) = k$:

$$\frac{\Gamma \vdash M_i : A \qquad (1 \leq i \leq k)}{\Gamma \vdash \sigma(M_1, \dots, M_k) : A}$$

We use syntactic sugar: $\mathbf{B}$ for $1 + 1$, `True` and `False` for `inl ()` and `inr ()` respectively, as well as `if` statements as shorthand for cases on $\mathbf{B}$. We also make use of n-ary sum and product types, defined from the binary ones by associating on the left, e.g. $A_1 + ... + A_n$ stands for $(\dots(A_1 + A_2)\dots) + A_n$. We write $\texttt{inj}_i$ and $\texttt{proj}_i$ for generalized versions of injections and projections. An integer $n \in \mathbb{N} \setminus 0$ can be seen as the type $1 + \cdots + 1$, n times.

For any $\sigma \in \Sigma$, we define the term $\vdash \mathrm{run}(\sigma) : \mathrm{ar}(\sigma)$ defined as $\sigma(\texttt{inj}_1\texttt{()}, \dots, \texttt{inj}_{\mathrm{ar}(\sigma)}\texttt{()})$. An example of this construction in the probabilistic setting is $\texttt{flip} := \mathrm{run}(+_{0.5}) = \texttt{True} +_{0.5} \texttt{False}$, the unbiased coin flip.

The free-algebra monad generated by a signature. Every signature Σ determines a monad T_Σ on **Set**, which computes the free Σ-algebra on a set. If X is a set, the elements of $T_\Sigma(X)$ are *computation trees* built out of the operations in Σ, with leaves labelled by elements of X. Formally, $T_\Sigma(X)$ is inductively defined by:

$$t \in T_\Sigma(X) ::= x \in X \mid \sigma(t_1, ..., t_k) \quad \sigma \in \Sigma \text{ with } \mathrm{ar}(\sigma) = k$$

The monad T_Σ gives a complete view of the algebraic operations appearing in the course of program execution. We will shortly recall (§2.2) how this is computed for a term $\Gamma \vdash M : A$.

Equations on the signature. It is common to equip the signature Σ with a set of equations, to form an algebraic theory [23]. Equations can better represent the intended behaviour of an effect: they express program equivalences that may be expected to hold in practice (for instance, in an implementation based on effect handlers [42]). As an example, in the programming language parametrized by the signature Σ_P we might expect equations like $M +_{0.5} M = M$ and $M +_p N = N +_{1-p} M$ to hold.

Formally, an *equation* for Σ is a triple (Δ, t, u) where Δ is a set and $t, u \in T_\Sigma(\Delta)$. The idea is that Δ represents a set of variables which appear in t and u, so this equation is usually written $\Delta \vdash t = u$. By asserting that the variables in Δ can be substituted for elements of an arbitrary set X, we obtain an equivalence relation on the computation trees in $T_\Sigma(X)$ generated by the equations.

An *algebraic theory* is a signature Σ together with a set $\mathcal{E}$ of equations. This induces a monad $T_{\Sigma,\mathcal{E}}(X)$, defined as the set of terms $T_\Sigma(X)$ quotiented by the equivalence relation.

In summary, an algebraic signature Σ freely determines a monad T_Σ of computation trees, and equations $\mathcal{E}$ on Σ determine a quotiented monad $T_{\Sigma,\mathcal{E}}$ in which the equations hold. This is a powerful framework: when we consider monads T that support the operations in Σ but which are not explicitly generated by a set of equations, we can still reason about which equations are satisfied in T by considering monad morphisms $T_{\Sigma,\mathcal{E}} \to T$ (see §3.3).

2.2 Eager semantics with a monad

We briefly recall (from Moggi [35]) the eager (call-by-value) semantics of our simple programming language using a strong monad.

Categorical setup. We fix a monad (T, η, μ) on a category $\mathscr{C}$. (We will often omit the structure maps η and μ.) We assume that $\mathscr{C}$ is bicartesian closed, meaning that it is has finite products and coproducts, and exponentials[3], and that T is *strong*, i.e. equipped with a natural transformation $\mathrm{str}_{A,B} : A \times T(B) \to T(A \times B)$ satisfying a number of coherence axioms [35]. In all examples of this section, $\mathscr{C} = \mathbf{Set}$ (so in particular all monads are automatically strong) but we will later move to a functor category. A final assumption is that every operation $\sigma \in \Sigma$ of arity k is *supported* in T by a morphism $\mathrm{op}_\sigma : \mathbf{1} \to T(k)$, where $\mathbf{1}$ is a terminal object and k is the k-ary coproduct $\mathbf{1} + \cdots + \mathbf{1}$. (This presentation follows [41]: op_σ is a "generic effect" for σ that determines the semantics of $\sigma(M_1, \ldots, M_k)$.)

▶ **Example 2.** We note some important instances of monads.

1. Any $\sigma \in \Sigma$ is supported by the monad T_Σ (§2.1), letting $\mathrm{op}_\sigma : \mathbf{1} \to T_\Sigma(\mathrm{ar}(\sigma)) : * \mapsto \sigma(1, \ldots, \mathrm{ar}(\sigma))$.

[3] We anticipate the higher-order type structure to come in §6.

2. The operations Σ_{P} are supported by D, the monad mapping a set X to the set of finite probabilistic distributions over X, where $+_p \in \mathrm{D}(2)$ is the distribution mapping 1 to p and 2 to $1-p$.

Call-by-value semantics of types and terms. Given all of the above structure the semantics of our language is straightforward (and well-known, since [35]).

Each type A is interpreted as an object $[\![A]\!]^v \in \mathscr{C}$ (the v stands for 'value') by induction on A, using the terminal object, products and coproducts of $\mathscr{C}$. For a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ we set $[\![\Gamma]\!]^v = [\![A_1]\!]^v \times \cdots \times [\![A_n]\!]^v$.

For the semantics of terms, recall that elements of $T([\![B]\!]^v)$ represent computations of type B, which perform effects and then return a value of type B. Accordingly, a term $\Gamma \vdash M : B$ is interpreted as a morphism $[\![M]\!]^v : [\![\Gamma]\!]^v \to T([\![B]\!]^v)$. We sometimes use an annotation $[\![-]\!]^v_T$ to emphasize that the semantics depends on the monad T. This avoids confusion when we are dealing with several monads.

We denote by $\mathbf{Kl}(T)$ the Kleisli category of T, defined to have the same objects as $\mathscr{C}$, and morphisms $A \to TB$. Composition in $\mathbf{Kl}(T)$ is defined using the monadic lifting operator: if $f : A \to TB$ then $f^\dagger := \mu \circ Tf : TA \to TB$. In a bicartesian closed category the coproducts are automatically distributive, meaning that the canonical morphism $[\texttt{inl} \times C, \texttt{inr} \times C] : A \times C + B \times C \to (A + B) \times C$ is invertible. Effectful operations $\sigma(M_1, ..., M_n)$ are interpreted using a morphism $[\![\sigma]\!] : A^n \to T(A)$ derived from $\mathrm{op}_\sigma : 1 \to T(k)$ as follows: $[\![\sigma]\!] = \mathrm{eval}^\dagger \circ \mathrm{str} \circ \langle \mathrm{id}_{A^n}, \mathrm{op}_\sigma \rangle$. The rest of the interpretation is standard [35].

Monads and program equations. We have given a general framework for the semantics of our simple language, using any strong monad that supports the operations. If the monad is of the form $T_{(\Sigma,\mathcal{E})}$, the algebraic equations $\mathcal{E}$ induce equations on programs, via the semantic interpretation. In the next section we look at an instance of this: idempotence equations as a simple model for laziness.

3 Semantics of laziness as an affine monad

In this section we develop a semantic model for laziness based on an affine monad derived from the signature Σ. We give a simple direct version in §3.1, and a much more refined version in §3.2 and §3.3 that exposes the name generation and stateful primitives. We then prove that the simple version can be recovered from the more sophisticated one (Theorem 11).

We continue to work with a fixed algebraic signature Σ.

3.1 Affine monads, idempotence, and laziness

In lazy evaluation, computations whose results are not used should not be performed. If programs can perform effects, this kind of evaluation determines an equational theory of programs which is not necessarily the same as the eager semantics.

We let $T_{\Sigma,\mathcal{E}_{\mathrm{idem}}}$ denote the monad on **Set** determined by the set of equations $\mathcal{E}_{\mathrm{idem}} = \{\{a\} \vdash \sigma(a, \ldots, a) = a \mid \sigma \in \Sigma\}$. These equations make every operation *idempotent*. For algebraic effects, idempotence justifies lazy evaluation: in a program of the form $\sigma(M, \ldots, M)$, the returned value of σ does not affect the continuation, so the operation can safely be ignored and the program is equivalent to M. Algebraic theories in which all operations are idempotent are called *affine* ([24]) and so $(\Sigma, \mathcal{E}_{\mathrm{idem}})$ is the universal affine theory over Σ.

Another perspective is based on affine monads.

▶ **Definition 3.** *A monad $T : \mathscr{C} \to \mathscr{C}$ on a cartesian category $\mathscr{C}$ is* affine *if $T1 \cong 1$.*

The connection between affine effects and laziness is known (e.g. [11]), and for algebraic effects it is equivalent to look at monads or algebraic theories. In particular the monad $T_{(\Sigma,\mathcal{E}_{\text{idem}})}$ is easily seen to be affine.

Applying the eager semantics of §2.2 under the monad $T_{(\Sigma,\mathcal{E}_{\text{idem}})}$ gives a semantics of laziness. While this is mathematically clear, it is far removed from an efficient lazy interpreter. We now show how to obtain the same semantics in a more instructive way.

3.2 Dynamic tracking of branching structure

In the rest of this section, we refine the semantic model in order to dynamically track names for occurrences of Σ-operations. We show how to encode the lazy semantics of our language using these operations, using two monads (N and R).

3.2.1 Worlds and indexed sets

So far we have only considered semantics for our language in the category **Set**. The dynamic tracking of names requires moving to a larger category in which objects are sets indexed by 'worlds'. A world contains information about the current execution: it is a record of all signature operations encountered so far and any branch that has already been explored.

The technical need for world-indexed sets will shortly become clear, as we introduce monads for refining and extending the current world.

▶ **Definition 4.** *For a signature Σ, a Σ-*world *(or just* world*) is a tuple $w = (|w|, \leq_w, \lambda_w)$ where $(|w|, \leq_w)$ is a finite totally ordered set and $\lambda_w : |w| \to \Sigma$.*

For a world w, we define a set $\mathrm{St}(w) = \prod_{x \in |w|} [\mathrm{ar}(\lambda_w(x))]$ of *states*, using the notation $[n] = \{0, \ldots, n-1\}$. In other words this is the set of possible assignments to the variables in the world. For a concrete instance, if $\Sigma = \Sigma_{\mathrm{P}}$, all signature operations are binary and therefore $\mathrm{St}(w) = \{0,1\}^{|w|}$.

A category of worlds. We now define a category of partially-explored worlds, denoted $\mathcal{W}_\Sigma$. Its objects are denoted by pairs (w, s) where w is a world and s is a partial assignment to the variables in w. Formally, this assignment is a pair of a subworld $w_0 \subseteq w$ together with an element $s \in \mathrm{St}(w_0)$, but we typically keep w_0 implicit and refer to it as $\mathrm{dom}(s)$, so $s = (s_x)_{x \in \mathrm{dom}(s)}$. A morphism $(w, s) \to (v, t)$ in $\mathcal{W}_\Sigma$ is an injective, monotone, label-preserving function $h : |w| \to |v|$ such that for any $x \in \mathrm{dom}(s)$, $h(x) \in \mathrm{dom}(t)$ and $s_x = t_{h(x)}$. We write $(w, \cdot)$ for a world with empty assignment.

We emphasize that these morphisms can extend the partial assignment. In particular we denote by $\mathrm{id}^{s \to t} : (w, s) \to (w, t)$ the morphism determined by the identity function, whenever the assignments satisfy $\mathrm{dom}(s) \subseteq \mathrm{dom}(t)$ and agree on $\mathrm{dom}(s)$.

We can extend a world w by appending new elements at the top: if v is another world, define $w \otimes v$ as the disjoint union $|w| + |v|$, labelled by the copairing $[\lambda_w, \lambda_v]$, and totally ordered following the combination of $\leq_w$ and $\leq_v$ in which elements of w are all below those of v. The operation $\otimes$ extends to a (non-symmetric) monoidal structure on $\mathcal{W}_\Sigma$, with $(w, s) \otimes (v, t)$ defined as $w \otimes v$ with the evident combination of s and t.

The operation St extends to a functor $\mathcal{W}_\Sigma^{\mathrm{op}} \to \mathbf{Set}$. On objects this is computed by collect the total assignments compatible with the current partial assignment: for $(w, s) \in \mathcal{W}_\Sigma$, let $\mathrm{St}(w, s) = \{s' \in \mathrm{St}(w) \mid \text{ for all } x \in \mathrm{dom}(s), s_x = s'_x\}$. On morphisms the action is computed

by pushing assignements across injective maps: every $h : (w, s) \to (v, t)$ induces a function $\mathrm{St}(v, t) \to \mathrm{St}(w, s)$ mapping t' to $(t'_{h(x)})_{x \in |w|}$. This action is well-defined, by the property of morphisms in $\mathcal{W}_\Sigma$.

World-indexed sets. In the rest of this section, we fix a signature Σ. We write $\mathcal{W}$ for the category $\mathcal{W}_\Sigma$. Our base category for semantics will now consist of sets indexed by worlds, formalized as functors $\mathcal{W} \to \mathbf{Set}$. We first note a convenient property.

▶ **Lemma 5.** *Let $\mathbf{Set}^{\mathcal{W}}$ denote the category of functors $\mathcal{W} \to \mathbf{Set}$ with natural transformations between them. There is an embedding $\Delta : \mathbf{Set} \hookrightarrow \mathbf{Set}^{\mathcal{W}}$ representing each set as a constant functor, and this functor has a right adjoint $\mathbf{Set}^{\mathcal{W}} \to \mathbf{Set}$ that picks out the set at index $\mathbf{0} \in \mathcal{W}$ (the unique empty world with empty assignment).*

The main motivation for using partial assignments is that for a functor $A \in \mathbf{Set}^{\mathcal{W}}$ and $(w, s) \in \mathcal{W}$, an element $a \in A(w, s)$ can be restricted along a partial assignments to variables in $w \setminus \mathrm{dom}(s)$, via the functorial action of A over the morphisms $\mathrm{id}^{s \to t} : (w, s) \to (w, t)$.

This is also a convenient setting because $\mathbf{Set}^{\mathcal{W}}$ is bicartesian closed, with products and coproducts computed pointwise and an explicit formula for exponentials [6]. Variants on this category are commonly used for tracking local information in program semantics, for instance in models of local state [38].

3.2.2 A name generation monad

We now define a monad N on $\mathbf{Set}^{\mathcal{W}}$ for dynamically adding new names to the world. This resembles existing monads for variable allocation and new name generation [39, 40, 16], but here names correspond to algebraic operations, and the formula involves a specific quotient making use of the restriction operations.

Informally, while a morphism $A \to B$ in $\mathbf{Set}^{\mathcal{W}}$ consists of functions $A(w, s) \to B(w, s)$, a Kleisli morphism $A \to NB$ can be understood as mapping each element of $A(w, s)$ to a pair containing a simple extension (without new assignments) $(w, s) \to (w', s)$ and an element of $B(w', s)$. More formally, for $A \in \mathbf{Set}^{\mathcal{W}}$, the functor $N(A) \in \mathbf{Set}^{W}$ has action

$$N(A)(w, s) = \{(v, a) \mid v \text{ is a world and } a \in A(w \otimes v, s)\}/\sim$$

where $(v, a) \sim (v', a')$ if and only if for every $t \in \mathrm{St}(w, s)$, there exist functions $f_t : (v, \cdot) \to (v''_t, \cdot)$ and $f'_t : (v', \cdot) \to (v''_t, \cdot)$ such that $A(\mathrm{id}^{s \to t} + f_t)(a) = A(\mathrm{id}^{s \to t} \otimes f'_t)(a')$ in $A(w \otimes v'', t)$.[4] Let $[v, a]_\sim$ denote the equivalence class of (v, a) under $\sim$. For $h : (w, s) \to (w', s')$, define $N(A)(h)\left([v, a]_\sim\right) = [v, A(h \otimes \mathrm{id})(a)]_\sim$.

▶ **Lemma 6.** *The operation N extends to a strong monad on $\mathbf{Set}^{\mathcal{W}}$. The monad unit $\eta^N : A \to N(A)$ and multiplication $\mu^N : N(N(A)) \to N(A)$ are defined by $\eta^N_{(w,s)}(a) = [(\mathbf{0}, a)]_\sim$ and $\mu_{(w,s)}([v, [v', a]_\sim]_\sim) = [v \otimes v', a]_\sim$.*

Note that, since the definition of $\mathcal{W}$ depends on the signature, there is a different monad N_Σ for every Σ, but we continue to omit the annotation.

One key feature of this monad is that new operations are only added to a world if they are truly needed. This is the role of the equivalence relation $\sim$. To illustrate this with an extreme example, if $A \in \mathbf{Set}^{\mathcal{W}}$ is a constant functor (say, $A = \Delta X$ for a set X), then there

---

[4] This formulation is inspired by [29]. Transitivity of the relation uses that every span can be completed to a square in the category of finite total orders and monotone injections.

is no point in ever extending the world, because all elements are already available at world $\mathbf{0}$. So the monad has no effects: one can show $N(A) \cong A$. But we will see below that N is genuinely effectful for non-constant functors.

3.2.3 A reader monad for branching structure

We now turn to an indexed kind of reader monad. At any stage a program may lookup the current state if required and branch on the value. For $A \in \mathbf{Set}^{\mathcal{W}}$ and $(w, s) \in \mathcal{W}$, define $RA(w, s) = A(w, s)^{\mathrm{St}(w,s)}$, with functorial action $R(A)(h) = A(h) \circ - \circ \mathrm{St}(h)$. This defines an endofunctor on $\mathbf{Set}^{\mathcal{W}}$ and indeed a monad.

▶ **Lemma 7.** *The operation R extends to a strong monad on $\mathbf{Set}^{\mathcal{W}}$. The monad unit $\eta^R : A \to R(A)$ and multiplication $\mu^R : R(R(A)) \to R(A)$ are standard for a reader monad, i.e. $\eta^R_{(w,s)}(a) : t \mapsto a$ and $\mu_{(w,s)}(f) : t \mapsto f(t)(t)$.*

3.3 An affine monad for the signature Σ

With the monads R and N we can formally describe the following informal interpretation for algebraic effects: upon encountering an operation, first create a named node for this occurrence, and then create a branch for each possible continuation (i.e. there are as many branches as the arity of the operation); all of this while enforcing and maintaining the invariant that no node should have all continuations the same.

Formally, we will show that the composite NR is a monad, and describe its relationship to the free monad T_Σ. The first step towards combining monads R and N is a distributive law [3]. Informally, world extensions appearing in distinct incompatible branches can be combined into a single world extension, and totally ordered according to a canonical order on the branches.

▶ **Theorem 8.** *There is a distributive law $\xi : RN \to NR$.*

Proof sketch. The component function $(\xi_A)_{(w,s)}$ of the distributive law has type

$$\left\{[v, a]_\sim \mid a \in A(w \otimes v, s)\right\}^{\mathrm{St}(w,s)} \to \{[V, f]_\sim \mid f \in A(w \otimes V, s)^{\mathrm{St}(w \otimes V, s)}\}.$$

For a family $\alpha := ([v_t, a_t]_\sim)_{t \in \mathrm{St}(w,s)}$ we define $(\xi_A)_w(\alpha)$ as $[V, f \in A(w \otimes V, s)^{\mathrm{St}(w \otimes V, s)}]_\sim$ where $V := \bigotimes_{t \in \mathrm{St}(w,s)} v_t$ with the iterated $\otimes$ following the lexicographic order on $\mathrm{St}(w, s)$, and f is the composite

$$\mathrm{St}(w \otimes V, s) \xrightarrow{\mathrm{St}((w,s) \hookrightarrow (w \otimes V, s))} \mathrm{St}(w, s) \xrightarrow{t \mapsto A(w \otimes \iota_t)(a_t)} A(w \otimes V, s). \qquad ◀$$

where $\iota_t : v_t \hookrightarrow V$ denote the injections.

It follows directly that $NR : \mathbf{Set}^{\mathcal{W}} \to \mathbf{Set}^{\mathcal{W}}$ is a monad.

▶ **Definition 9.** *For a signature Σ, let $T^{\mathrm{aff}}_\Sigma : \mathbf{Set} \to \mathbf{Set}$ be the monad induced by composing the monad NR with the adjunction of Lemma 5. Explicitly, for a set X, $T^{\mathrm{aff}}_\Sigma(X) = NR(\Delta_X)(\mathbf{0})$. This is characterized as the set of pairs (v, f), where v is a world and $f \in X^{\mathrm{St}(v)}$, quotiented by the smallest equivalence relation containing $(v, f) \sim (v', f')$ if there exists $h : (v, \cdot) \to (v', \cdot)$ in $\mathcal{W}$ such that $f' = f \circ \mathrm{St}(h)$.*

▶ **Lemma 10.** *For any signature Σ, the monad T^{aff}_Σ is affine.*

Proof. By the characterization, elements of $T_\Sigma^{\text{aff}}\mathbf{1}$ are equivalence classes of pairs $(v, f : \mathrm{St}(v) \to \mathbf{1})$. In every such pair f must be the unique map to $\mathbf{1}$ and so $(v, f) \sim (\mathbf{0}, ! : \mathrm{St}(\mathbf{0}) \to \mathbf{1})$ via the injection $\mathbf{0} \to v$. We have shown that there is a unique equivalence class and thus $T_\Sigma^{\text{aff}}\mathbf{1} \cong \mathbf{1}$. ◀

Each operation $\sigma \in \Sigma$ has a semantic representation $\mathrm{op}_\sigma \in T_\Sigma^{\text{aff}}([\mathrm{ar}(\sigma)])$ given by the equivalence class of the pair $((\{*\}, * \mapsto \sigma), \mathrm{id}_{\mathrm{ar}(\sigma)})$. The idea is to create a named occurrence of σ, whose continuation follows $\mathrm{id}_{\mathrm{ar}(\sigma)}$: for every branch of computation, i.e. for every index within the arity of σ, simply return that index.

That T_Σ^{aff} is affine implies that all operations of arity 1 are invisible in the model. This is a desired property for a model of lazy computation, because no computation can ever depend on the result of an operation of arity 1. (In lazy languages there is typically a way to enforce eager behaviour for certain operations, to enable essential unary operations like printing or writing to memory. But this breaks laziness.)

The lazy semantics. The monad T_Σ^{aff} tracks enough information to fully reconstruct a computation tree in the sense of T_Σ. To state this formally, we first define for each world w and set X a function $t_{X,w} : X^{\mathrm{St}(w)} \to T_\Sigma(X)$. The definition is by induction on the size of the world:

- If $w = \mathbf{0}$, $\mathrm{St}(w) = \mathbf{1}$. A function $\mathrm{St}(w) \to X$ can be identified with an element of X, hence of $T_\Sigma(X)$.
- If $w \neq \mathbf{0}$, then w has (up to isomorphism) a unique decomposition as $(\mathbf{1}, * \mapsto \sigma) \otimes v$ for some world v and some $\sigma \in \Sigma$. Assume that we have constructed a function $t_{X,v} : X^{\mathrm{St}(v)} \to T_\Sigma(X)$. Since $\mathrm{St}(w) \cong \sum_{i=1}^{\mathrm{ar}(\sigma)} \mathrm{St}(v)$, each function $f : \mathrm{St}(w) \to X$ can be written as a copairing $[f_i]_{i=1}^{\mathrm{ar}(\sigma)}$ for $f_i \in X^{\mathrm{St}(v)}$. If $\mathrm{ar}(\sigma) = 0$, then return $\sigma() \in T_\Sigma(X)$. Otherwise, define

$$t_{X,w}(f) = \begin{cases} t_{X,v}(f_1) & \text{if } \forall i.\ t_{X,v}(f_i) = t_{X,v}(f_1) \\ \sigma(t_{X,v}(f_1), \ldots, t_{X,v}(f_n)) & \text{otherwise.} \end{cases}$$

In summary, elements of each $X^{\mathrm{St}(w)}$ can be represented as computation trees, and the case-split definition ensures that a node is discarded if its result is not strictly needed. From this, one shows compatibility of the $t_{X,v}$ with the equivalence relation $\sim$, so that elements of T_Σ^{aff} induce 'idempotent' computation trees, as per the following theorem. We emphasize that T_Σ^{aff} contains strictly more information than $T_{\Sigma,\mathcal{E}_{\text{idem}}}$, namely, the total order on operations.

▶ **Theorem 11.** *For any algebraic signature Σ, there is a monad morphism $T_\Sigma^{\text{aff}} \to T_{\Sigma,\mathcal{E}_{\text{idem}}}$, where $\mathcal{E}_{\text{idem}}$ is the set of idempotence equations (§3.1), that commutes with the canonical maps from the free monad as per the following diagram.*

$$\begin{array}{ccc} T_\Sigma & & \\ \downarrow & \searrow & \\ T_\Sigma^{\text{aff}} & \longrightarrow & T_{\Sigma,\mathcal{E}_{\text{idem}}} \end{array}$$

Using that $T_{(\Sigma,\mathcal{E}_{\text{idem}})}$ is the initial affine monad supporting Σ, we immediately deduce:

▶ **Corollary 12.** *Let T be an arbitrary monad on* **Set** *supporting the operations of an algebraic signature Σ. If T is affine, then there is a canonical monad morphism $T_\Sigma^{\text{aff}} \to T$ preserving the semantics of Σ-operations.*

Summary. We have constructed a monad $T^{\mathrm{aff}}_{\Sigma}$ that is closely related to the free affine monad $T_{(\Sigma,\mathcal{E}_{\mathrm{idem}})}$. The key point is that $T_{(\Sigma,\mathcal{E}_{\mathrm{idem}})}$ is a simple canonical model for laziness, that only imposes idempotence equations, whereas $T^{\mathrm{aff}}_{\Sigma}$ has a much more complex presentation obtained via a factorization over a functor category. The power of this approach is to expose all name generation and state access, necessary in for more efficient intermediate representations. Theorem 11 and Corollary 12 ensure correctness.

4 Reader algebras as intermediate representations

In this section we temporarily focus on the reader monad, to show how an efficient tree-like intermediate representation for lazy evaluation (cf. §1.3) arises naturally from an interpretation in its category of algebras.

4.1 Algebras for a monad

We give the basic definitions. To motivate, recall that any algebraic theory $(\Sigma, \mathcal{E})$ induces a monad $T_{\Sigma,\mathcal{E}}$. Recall also (e.g. [32]) that a *model* of $(\Sigma, \mathcal{E})$ is a set X equipped with a function $X_\sigma : X^{\mathrm{ar}(\sigma)} \to X$ for every $\sigma \in \Sigma$, satisfying the equations in $\mathcal{E}$. It is an easy observation that the structure of a model can equivalently be presented as a function $T_{\Sigma,\mathcal{E}}(X) \to X$ satisfying some compatibility axioms. Under this presentation, X is known as a $T_{\Sigma,\mathcal{E}}$-algebra. This is an instance of the general definition that follows.

▶ **Definition 13.** *Let T be a monad on a category $\mathscr{C}$ with structure maps η and μ. An* algebra for T *(or T-*algebra*) consists of an object $A \in \mathscr{C}$ and a morphism $h : TA \to A$, subject to the following axioms (1) $h \circ \eta_A = \mathrm{id}_A$ and (2) $h \circ Th = h \circ \mu_A$.*

If (A, h) and (B, k) are T-algebras, a morphism $f : A \to B$ is a homomorphism of T-algebras *if $f \circ h = k \circ Tf$. We write $\mathbf{Alg}(T)$ for the category of T-algebras and homomorphisms.*

We write $U : \mathbf{Alg}(T) \to \mathscr{C}$ for the forgetful functor $(A, h) \mapsto A$, and $F : \mathscr{C} \to \mathbf{Alg}(T)$ for the free algebra *functor that maps A to $(TA, \mu_A : TTA \to TA)$. Note that $T = U \circ F$.*

The following examples are important for this paper.

▶ **Example 14.** An algebra for the monad $T_\Sigma : \mathbf{Set} \to \mathbf{Set}$, for a signature Σ, is a set X equipped with an operation $\mathrm{op}_\sigma : X^{\mathrm{ar}(\sigma)} \to X$ for every $\sigma \in \Sigma$. There are no axioms. Homomorphisms of T_Σ-algebras are functions that preserve the op_σ.

An algebra for the monad $\mathrm{D} : \mathbf{Set} \to \mathbf{Set}$ of finite distributions is a pair $(X, h : \mathrm{D}X \to X)$ that can be more directly described as a convex space: a set X equipped with an operation $+_p : X^2 \to X$ for any $p \in [0, 1]$, an abstract notion of convex sum. (This is subject to equational laws, e.g. [15]).

Algebras in semantics. Computation types in our language naturally admit an algebra structure for T_Σ, because given computations $M_1, \ldots, M_{\mathrm{ar}(\sigma)}$ one can always construct $\sigma(M_1, \ldots, M_{\mathrm{ar}(\sigma)})$. Although algebras are not used explicitly in the call-by-value semantics of §2.2, the structure is there nonetheless: it is well-known that the Kleisli category $\mathbf{Kl}(T)$ is equivalent to the full subcategory of $\mathbf{Alg}(T)$ consisting of free algebras [32]. Algebras play a more explicit role in monadic models of call-by-name, see e.g. [27].

4.2 Cartesian monads

Reader algebras. We now look more specifically at algebras for the reader monad $R : \mathbf{Set}^{\mathcal{W}} \to \mathbf{Set}^{\mathcal{W}}$. Since the discussion is largely independent of world extensions and name

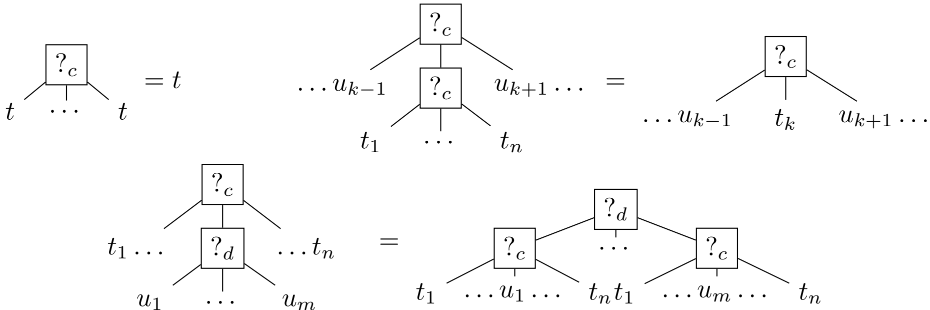


Figure 2 Equations for algebras over $R_{(w,s)}$. Here $c, d \in |w| \setminus \mathrm{dom}(s)$ with respective arities n, m.

generation, in some cases it will suffice to consider 'local' reader monads $R_{(w,s)} : \mathbf{Set} \to \mathbf{Set}$ given as $X \mapsto X^{\mathrm{St}(w,s)}$, for a fixed world $(w, s) \in \mathcal{W}$.

▶ **Lemma 15.** *For any $(w, s) \in \mathcal{W}$, the structure of an $R_{(w,s)}$-algebra on a set A admits an equivalent presentation in terms of operations $?_c : A^{\mathrm{ar}(\lambda_w(c))} \to A$ for every $c \in |w| \setminus \mathrm{dom}(s)$, satisfying the axioms of Figure 2.*

Notation. In a reader algebra, if c is a name for a binary operation, we write $t +_c u$ for $?_c(t, u)$ for readability. We demonstrate several examples below in this special case.

The equations of reader algebras can be expressed as program equations showing that, as a computational effect, read-only state is very tame, and difficult for a program context to 'observe'. In the call-by-value language, under the signature $\Sigma_w = (|w|, \mathrm{ar} \circ \lambda_v)$ induced by any world w, one has the semantic equalities

$$\begin{aligned} \mathtt{let}\, x = M\, \mathtt{in}\, N &= N \\ \mathtt{let}\, x = M\, \mathtt{in}\, \langle x, x\rangle &= \mathtt{let}\, x = M\, \mathtt{in}\, (\mathtt{let}\, y = M\, \mathtt{in}\, \langle x, y\rangle) \\ \mathtt{let}\, x = M\, \mathtt{in}\, (\mathtt{let}\, y = M'\, \mathtt{in}\, N) &= \mathtt{let}\, y = M'\, \mathtt{in}\, (\mathtt{let}\, x = M\, \mathtt{in}\, N) \end{aligned}$$

which respectively express that the effect is affine (§3.1), *relevant*, and *commutative*. Taken in combination, the three properties amount to the following:

▶ **Definition 16** ([14]). *A strong monad $T : \mathscr{C} \to \mathscr{C}$ is* cartesian[5] *when the underlying functor is finite-product-preserving, i.e. the morphism $\langle T\pi_1, T\pi_2\rangle : T(A \times B) \to T(A) \times T(B)$ is invertible.*

The following is almost immediate:

▶ **Lemma 17.** *The monad $R : \mathbf{Set}^{\mathcal{W}} \to \mathbf{Set}^{\mathcal{W}}$ is cartesian, as are the monads $R_{(w,s)} : \mathbf{Set} \to \mathbf{Set}$ for every $(w, s) \in \mathcal{W}$.*

4.3 Algebras of a cartesian monad

We now look at the structure of $\mathbf{Alg}(T)$ when T is a cartesian monad. Assuming enough structure on the base category, categories of algebras have products and coproducts, but for cartesian monads we have the following stronger result:

▶ **Theorem 18** ([20], Thm 2.6). *If $T : \mathscr{C} \to \mathscr{C}$ is a cartesian monad and $\mathscr{C}$ is cartesian closed and complete, then $\mathbf{Alg}(T)$ is cartesian closed.*

[5] The terminology 'cartesian' is due to Jacobs [14] and unfortunately clashes with a distinct notion of cartesian monad [25]. Kock [20] calls the monads of Def. 16 'cartesian closed'.

Details of categorical structure. We walk through the categorical structure of $\mathbf{Alg}(T)$: products, coproducts, and exponentials.

For an arbitrary monad $T : \mathscr{C} \to \mathscr{C}$ on cartesian $\mathscr{C}$ and for T-algebras (A, h_A) and (B, h_B), there is an algebra structure on $A \times B$ given by $(h_A \times h_B) \circ \langle T\pi_1, T\pi_2\rangle : T(A \times B) \to A \times B$. In other words, the algebra structure on products is defined componentwise. As an example, for a reader monad $R_{(w,s)}$ with only binary operations $c \in w$, if A and B are $R_{(w,s)}$-algebras then so is $A \times B$, with $(a, b) +_c (a', b') := (a +_c a', b +_c b')$.

Coproducts of algebras are more subtle: when (A, h_A) and (B, h_B) are algebras, there is in general no algebra structure on the coproduct $A+B$ in $\mathscr{C}$. The solution [31] is a quotient of $T(A+B)$, namely the coequalizer $T(TA+TB) \overset{T(h_A+h_B)}{\underset{\mu\circ T[T\mathrm{inl}_{\mathscr{C}}, T\mathrm{inr}_{\mathscr{C}}]}{\rightrightarrows}} T(A+B) \overset{q}{\dashrightarrow} A \oplus B$ when it exists in $\mathscr{C}$. We briefly illustrate this for $R_{(w,s)}$-algebras (A, h_A) and (B, h_B). What should $\mathrm{inl}\ a +_c \mathrm{inr}\ b$ be, for $a \in A$ and $b \in B$? One could freely add these operations, i.e. take $(A + B)^{\mathrm{St}(w)}$, but then the obvious map $i : A \to (A + B)^{\mathrm{St}(w)}$ is not a homomorphism: $i(\mathrm{inl}_R(a+_c a'))$ gives $s \mapsto \mathrm{inl}_{\mathbf{Set}}(a+_c a')$, while $(i(a)+_c i(a'))(s)$ gives $\mathrm{inl}_{\mathscr{C}}(a)$ when $s(c) = \texttt{True}$ and $\mathrm{inl}_{\mathscr{C}}(a')$ otherwise. The coequalizer makes these elements equal.

For exponentials, given two algebras A and B, one can construct an internal-hom $A \multimap B$ representing the algebra of homomorphisms, with the algebra structure intuitively defined pointwise. We omit the abstract construction of $A \multimap B$ as an equalizer, which we do use explicitly. See [20] for details.

Algebras as a semantic model: $[\![-]\!]^{\mathfrak{a}}$. We now assume that $\mathscr{C}$ is bicartesian closed, complete, and cocomplete, and that T is cartesian. As a bicartesian closed category, $\mathbf{Alg}(T)$ supports an interpretation of the fragment of our language without any calls to Σ-effects. We explain how this interpretation relates to the eager semantics. This is only a limited form of correctness (because there are no effects); later (in §5.3) we extend it to the full language by reinstating the monad N.

Let $[\![-]\!]^{\mathfrak{a}}$ denote the semantics of types and terms in $\mathbf{Alg}(T)$ (i.e. $[\![-]\!]^{\mathfrak{a}}$ is the semantics of §2.2 under the identity monad on $\mathbf{Alg}(T)$). We write $[\![-]\!]^{\mathfrak{a}}_T$ to specify the monad, if necessary. We first observe:

▶ **Theorem 19.** *Under the above assumptions on $\mathscr{C}$, the canonical functor $\mathbf{Kl}(T) \to \mathbf{Alg}(T)$ (given by $X \mapsto (TX, \mu_X)$ and $(f : X \to TY) \mapsto (f^\dagger : TX \to TY)$) preserves finite products and coproducts.*

Proof. By the cartesian property for T there is an isomorphism $T(A\times B) \cong T(A)\times T(B)$. As a left adjoint the free-algebra functor $F : \mathscr{C} \to \mathbf{Alg}(T)$ preserves coproducts, i.e. $T(A+B) \cong T(A) \oplus T(B)$ in $\mathbf{Alg}(T)$. ◀

From there we easily construct an isomorphism $\varphi_A : [\![A]\!]^{\mathfrak{a}} \cong T([\![A]\!]^v)$ and verify that for any term $\Gamma \vdash M : B$ of the pure fragment, the diagram

$$\begin{array}{ccc} [\![\Gamma]\!]^{\mathfrak{a}} & \xrightarrow{[\![M]\!]^{\mathfrak{a}}} & [\![B]\!]^{\mathfrak{a}} \\ \downarrow{\scriptstyle \varphi_\Gamma} & & \downarrow{\scriptstyle \varphi_B} \\ T([\![\Gamma]\!]^v) & \xrightarrow{[\![M]\!]^{v\,\dagger}} & T([\![B]\!]^v) \end{array} \tag{1}$$

commutes. This follows from an easy induction, because for now the language is first-order.

4.4 The intermediate representation

We continue to consider a fixed world w, and show that the semantics $[\![A]\!]^{\mathfrak{a}}_{R_{(w,s)}}$ (here just $[\![A]\!]^{\mathfrak{a}}$) can be described in terms of arborescent structures, like those of the introduction. We make these structures precise, using labelled, multi-valued decision nodes (generalizing those of binary decision diagrams [5]) interleaved with type constructors, considered up to an equational theory. The equations also make it possible to reason about possible implementations, where certain kinds of nodes can be permuted.

A set $\mathrm{Tree}(w,s)$ is defined inductively as

$$t, u \in \mathrm{Tree}(w,s) := \texttt{()} \mid ?_c(t_1, ..., t_n) \mid \texttt{inl}\ t \mid \texttt{inr}\ t \mid \langle t, u\rangle$$

where $c \in |w| \setminus \mathrm{dom}(s)$ and $\mathrm{ar}(\lambda_w(c)) = n$. We call the top-level constructor in $?_c(t_1, \ldots, t_n)$ a *choice node*; all other constructors are *value nodes*. A tree $t \in \mathrm{Tree}(w,s)$ is a *value* if it only consists of value nodes.

For each type A, we furthermore carve out a subset $\mathrm{Tree}(A)(w,s) \subseteq \mathrm{Tree}(w,s)$ of *well-typed* trees (of type A), via the following rules:

1. $\texttt{()} \in \mathrm{Tree}(1)(w,s)$.
2. If $t_1, \ldots, t_n \in \mathrm{Tree}(A)(w,s)$ and $c \in |w| \setminus \mathrm{dom}(s)$ then $?_c(t_1, \ldots, t_n) \in \mathrm{Tree}(A)(w,s)$.
3. If $t \in \mathrm{Tree}(A)(w,s)$ then $\texttt{inl}\ t \in \mathrm{Tree}(A+B)(w,s)$ and $\texttt{inr}\ t \in \mathrm{Tree}(B+A)(w,s)$.
4. If $t \in \mathrm{Tree}(A)(w,s)$ and $u \in \mathrm{Tree}(B)(w,s)$ then $\langle t, u\rangle \in \mathrm{Tree}(A \times B)(w,s)$.

We recover $[\![D]\!]^{\mathfrak{a}}$ via an equational theory. Let $\equiv$ be the equivalence relation on trees generated by all equations of reader algebras (Lemma 15), together with the following equations, asserting the compatibility of products and coproducts with the algebra structure.

$$\begin{aligned} \texttt{inl}\ ?_c(t_1, \ldots, t_n) &= ?_c(\texttt{inl}\ t_1, \ldots, \texttt{inl}\ t_n) \qquad (\text{resp. } \texttt{inr}) \\ \langle ?_c(t_1, \ldots, t_n), ?_c(u_1, \ldots, u_n)\rangle &= ?_c(\langle t_1, u_1\rangle, \ldots, \langle t_n, u_n\rangle) \end{aligned}$$

It is easy to verify that if $t \equiv u$ for $t \in \mathrm{Tree}(A)(w,s)$ then $u \in \mathrm{Tree}(A)(w,s)$, therefore $\equiv$ restricts to an equivalence relation on each of the $\mathrm{Tree}(A)(w,s)$.

▶ **Theorem 20.** *The constructions* Tree *and* $\mathrm{Tree}(A)$ *determine functors* $\mathcal{W} \to \mathbf{Set}$*, and for every type* A*, there is an isomorphism* $[\![A]\!]^{\mathfrak{a}}_R(w,s) \cong \mathrm{Tree}(A)(w,s)/\equiv$*, natural in* $(w,s) \in \mathcal{W}$*.*

This theorem states that $\mathrm{Tree}(A)$ is a faithful representation of $[\![A]\!]^{\mathfrak{a}}$ up to the equations. Concretely, this means that one can implement $[\![A]\!]^{\mathfrak{a}}$ by $\mathrm{Tree}(A)$ (a simple algebraic data type) and use the equations to optimize the structure. We will do in §5.

4.5 Comparison with call-by-name semantics

We briefly compare the semantics $[\![-]\!]^{\mathfrak{a}}_T$, for a cartesian monad T, with the traditional semantics of call-by-name types as T-algebras [26], in which the interpretation of types is as follows:

$$[\![1]\!]^n = F\mathbf{1} \quad [\![A \times B]\!]^n = [\![A]\!]^n \times [\![B]\!]^n \quad [\![A + B]\!]^n = F(U[\![A]\!]^n + U[\![B]\!]^n)$$

The key difference between $[\![-]\!]^{\mathfrak{a}}$ and $[\![-]\!]^n$ is the treatment of sum types. The call-by-name equational theory does not include the η-rule for sums, and therefore the semantics is not based on a categorical coproduct. (To illustrate more concretely, this means that in a probabilistic call-by-name language terms $\texttt{inl}\ (M +_p N)$ and $\texttt{inl}\ M +_p \texttt{inl}\ N$ are not identified: the context $\texttt{case} \bullet : [\texttt{inl}\ x.\ \texttt{()} \mid \texttt{inr}\ x.\ \texttt{()}]$ will trigger the effect in the latter but not the former.)

We can relate the two semantics formally. Since the coproduct of algebras $A \oplus B$ is defined as a quotient of $T(A+B)$, there is a map $q : T(A+B) \to A \oplus B$. This induces a morphism $\chi_A : [\![A]\!]^n \to [\![A]\!]^{\mathfrak{a}}$, for any type A, by induction. The following holds in our first-order language:

▶ **Lemma 21.** *For any term $\Gamma \vdash M : B$, the diagram*

$$\begin{array}{ccc} [\![\Gamma]\!]^n & \xrightarrow{[\![M]\!]^n} & [\![B]\!]^n \\ {\scriptstyle \chi_\Gamma}\downarrow & & \downarrow{\scriptstyle \chi_B} \\ [\![\Gamma]\!]^{\mathfrak{a}} & \xrightarrow{[\![M]\!]^{\mathfrak{a}}} & [\![B]\!]^{\mathfrak{a}} \end{array}$$

commutes, where χ_Γ is the evident extension to contexts.

Technical point. Another difference between the two semantics is that the call-by-name interpretation $[\![M]\!]^n$ is not in general a morphism of algebras, only a morphism between the carrier objects. The interpretation of sum types as a free algebra requires an application of the strength and the return function of the monad (to define the injections), which are not morphisms of algebras.

5 The lazy interpreter

We are now a position to present a design for a lazy interpreter for the simple language with Σ-effects. This relies on the decomposition of T_Σ^{aff} in terms of monads N and R on the functor category $\mathbf{Set}^{\mathcal{W}}$. We first extend the results of §4 so that the algebra semantics for R can be appropriately combined with N, then we give concrete code for the interpreter.

The two roles played by monads. Monads can play two separate roles in programming languages: on the one hand, they describe the semantics of functions in an impure language, and on the other, they are a programming method for user-defined effects. Our proposal is that monads N and R should respectively play these separate roles within the lazy interpreter: we regard $\mathbf{Kl}(N)$ as a model for a language where name generation is implemented natively, whereas we let the programmer explicitly define and manipulate the algebra semantics for R.

To make our presentation concrete, we use OCaml as a host language for the interpreter. It is easy to represent name generation in OCaml using a counter (see §5.2), although other methods would be possible. Our interpreter then consists of an implementation of the algebra semantics for the monad R, using the intermediate representation of §4.4. (As an alternative to OCaml we could perhaps have used a formal metalanguage for $\mathbf{Kl}(N)$, e.g. [39]. Our presentation emphasizes the practical aspects.)

5.1 An "efficient" lazy semantics

Our optimized representations exploit the cartesian structure of $R : \mathbf{Set}^{\mathcal{W}} \to \mathbf{Set}^{\mathcal{W}}$, but the design of our interpreter is based on a 'lifted' version $R_N : \mathbf{Kl}(N) \to \mathbf{Kl}(N)$, giving access to name generation. This makes the presentation a bit more technical, e.g. because $\mathbf{Kl}(N)$ does not have cartesian products. In what follows we clarify the situation and fix notation.

Distributive laws and algebras. From the distributive law between R and N (Theorem 8) we in fact obtain two lifted monads [3].

(1) A lifted version of R to the Kleisli category of N, denoted by $R_N : \mathbf{Kl}(N) \to \mathbf{Kl}(N)$.

(2) A lifted version of N to the category of R-algebras, denoted by $N^R : \mathbf{Alg}(R) \to \mathbf{Alg}(R)$.

Our interest is primarily in (1) and the induced category of algebras $\mathbf{Alg}(R_N)$, following the methodology of §4. But $\mathbf{Kl}(N)$ is not bicartesian closed (formally, it is a closed Freyd category with sums [28]) which makes the definitions more tedious. We can bypass this issue using (2) because $\mathbf{Kl}(N^R)$ corresponds to a well-behaved subcategory of $\mathbf{Alg}(R_N)$, as shown in the next lemma. As a rough summary, the idea will be to perform all constructions in the bicartesian closed category $\mathbf{Alg}(R)$ equipped with the monad N^R, and embed the resulting model in $\mathbf{Alg}(R_N)$. By 'embedding' we mean a full and faithful, injective-on-objects functor.

▶ **Lemma 22.** *A distributive law as in Thm 8 induces an embedding $J : \mathbf{Kl}(N^R) \to \mathbf{Alg}(R_N)$.*

Proof sketch. Objects of $\mathbf{Kl}(N^R)$ are R-algebras, i.e. pairs $(A \in \mathbf{Set}^{\mathcal{W}}, a : RA \to A)$ while objects of $\mathbf{Alg}(R_N)$ are pairs $(A \in \mathbf{Set}^{\mathcal{W}}, a : RA \to NA)$. Define J on objects as $(A, a) \mapsto (A, \eta \circ a)$. The rest of the construction is straightforward. ◀

One way to understand this embedding is that the objects of $\mathbf{Kl}(N^R)$ are the 'pure' R_N-algebras: those whose algebra structure does not create any new names.

Finally, we relate the category $\mathbf{Kl}(NR)$ (for the composite monad NR on $\mathbf{Set}^{\mathcal{W}}$) with $\mathbf{Alg}(R_N)$: there is a lifting $(-)^\star : \mathbf{Kl}(NR) \to \mathbf{Alg}(R_N)$ mapping an object X to $R(X)$ and a morphism $f : X \to NR(Y) \in \mathbf{Kl}(NR)[X, Y]$ to a $f^\star : R(X) \to NR(Y) := f^\dagger \circ \eta^N_{RX} \in \mathbf{Alg}(R_N)[RX, RY]$.

A new lazy semantics to structure the interpreter. We can now introduce a semantic model including both ingredients of the interpreter: "efficient" representations of types as algebras, and a mechanism for name generation. This is entirely in the spirit of §4, but now in a more general setting that includes N. We will prove the semantics correct with respect to T^{aff}_Σ in §5.3.

The category $\mathbf{Alg}(R)$ is bicartesian closed (because R is a cartesian monad) and supports an interpretation of the pure language as described in §4: indeed in $\mathbf{Alg}(R)$, terms can interact with computations but cannot perform effects as they cannot allocate new names. The monad N^R supports the algebraic operations in Σ as follows: we define $\mathrm{op}_\sigma : \mathbf{1} \to [\![k]\!]^v_{N^R} \cong NR(k)$ as the element $\big((\mathbf{1}, * \mapsto \sigma), s \mapsto s_*\big)$.

▶ **Definition 23** ('Interpreter' semantics)**.** *For a type A and a term $\Gamma \vdash M : A$, let $[\![A]\!]^{\mathrm{i}}$ denote $J[\![A]\!]^v_{N^R}$, an object of $\mathbf{Alg}(R_N)$, and similarly let $[\![M]\!]^{\mathrm{i}}$ denote $J[\![M]\!]^v_{N^R}$, a morphism of $\mathbf{Alg}(R_N)$.*

To spell this out a bit more explicitly, denote "pure" R_N-algebras, i.e. essentially R-algebras, and $\Gamma \vdash M : A$ denotes an R-algebra homomorphism $[\![\Gamma]\!]^{\mathrm{i}} \to N([\![B]\!]^{\mathrm{i}})$. As usual the interpretation of types follows the categorical structure: $[\![1]\!]^{\mathrm{i}} = \mathbf{1}$, $[\![A + B]\!]^{\mathrm{i}} = [\![A]\!]^{\mathrm{i}} \oplus_R [\![B]\!]^{\mathrm{i}}$, and $[\![A \times B]\!]^{\mathrm{i}} = [\![A]\!]^{\mathrm{i}} \times [\![B]\!]^{\mathrm{i}}$.

Summary. We give a summary of the overall situation in Figure 3. The figure also outlines the proof goal: to find functors that preserve categorical structure and the interpretation of effects. To prove the correspondance between the interpreter (R-algebras over the Kleisli of N) and the reference semantics (Kleisli of T), there are several steps. The first step is to use that R is product-preserving so that in $\mathbf{Kl}(N)$, the algebra model and the Kleisli model for R coincide (Theorem 19), and the Kleisli model of R on $\mathbf{Kl}(N)$ is exactly $\mathbf{Kl}(NR)$. The second step is to go from $\mathbf{Kl}(NR)$ to $\mathbf{Kl}(T^{\mathrm{aff}}_\Sigma)$ where the difficulty is the change of category: NR is defined on $\mathbf{Set}^{\mathcal{W}}$ while T^{aff}_Σ is defined on $\mathbf{Set}$. However since T^{aff}_Σ is defined using NR this is straightforward. The last arrow from $\mathbf{Kl}(T^{\mathrm{aff}}_\Sigma)$ to $\mathbf{Kl}(T)$ is given by Theorem 12.

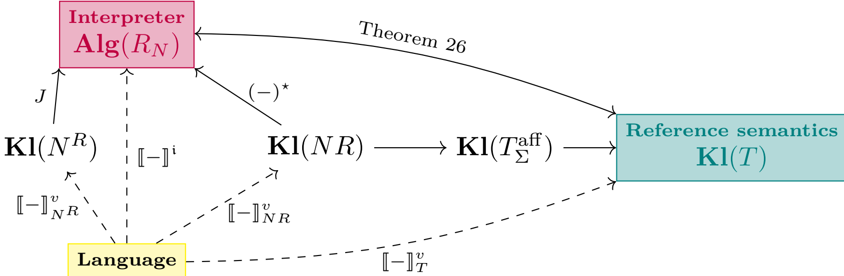


Figure 3 Landscape of models and correctness results for an affine monad T. Dashed arrows are interpretations and solid arrows are (first-order) interpretation-preserving functors.

5.2 Practical design for a lazy interpreter

We now demonstrate that the semantic model $[\![-]\!]^{\mathrm{i}}$ informs the implementation of a lazy interpreter. We stick to a very simple setting: a signature with a single binary operation `Plus`. The interface is completely standard for a definitional interpreter [43].

```
type op = Plus        (* signature *)
type term = Op of op * term * term
 | Case of term * (var * term) * (var * term) |...
type tree             (* defined below *)
type env = (var * tree) list
val interp : env -> term -> tree
```

Name generation is implemented directly via OCaml mutable references:

```
type name = int * op
let eq (x, _) (y, _) = x = y
let allocate : op -> name =
   let r = ref 0 in (fun op -> incr r; (!r, op))
```

The intermediate representation (the functor Tree $\in \mathbf{Set}^{\mathcal{W}}$) corresponds to the OCaml type

```
type tree =
  | Choice of name * tree * tree | Inl of tree
  | Inr of tree | Unit | Pair of tree * tree
```

equipped with the following ‘functorial action’; recall that morphisms in $\mathcal{W}$ may assign values to variables outside the current assignment.

```
(* Remove all branches on [name], picking branch [b] *)
let rec set name b tree =
  match tree with
  | Choice (name', t, u) when eq name name' ->
    if b then t else u
  | Choice (name', t, u) ->
    Choice (name', set name b t, set name b u)
  | Inl t -> Inl (set name b tree) | Inr t -> Inr (set name b tree)
  | Unit -> Unit | Pair (t, u) ->
    Pair (set name b t, set name b u)
```

The algebra structure on the type `tree` is reflected in the function `choose` below, corresponding to the operation $+_c$ in the signature.[6]

```
let rec choose name t1 t2 =
  match (set name true t1, set name false t2) with
  | Inl t, Inl t' -> Inl (Choice (name, t, t'))
  | Inr t, Inr t' -> Inr (Choice (name, t, t'))
  | t, t' -> Choice (name, t, t')
```

Once this infrastructure is in place, it remains to follow the inductive structure of the semantics $[\![-]\!]^{\mathfrak{i}}$. Most cases (pairs, products) are completely standard. The key point is that laziness is dealt with automatically, via the intermediate representation. We illustrate with two interesting cases: the algebraic operation, and the case construct for pattern matching.

```
(* Implements the copairing [f, g] *)
let rec copair tree (f : tree -> tree) (g : tree -> tree) =
  match tree with
  | Inl x -> f x
  | Inr y -> g y
  | Choice (name, t1, t2) ->
    choose name (copair t1 f g) (copair t2 f g)
  | _ -> assert false

let rec interp (env : env) (term : term) : tree =
  match term with
  | Op {operation; l; r} ->
      let name = allocate operation in
      Choice (name, interp env l, interp env r)
  | Case (term, (x1, body1), (x2, body2)) ->
      copair (interp env term)
        (fun t -> interp ((x1, t) :: env) body1)
        (fun u -> interp ((x2, u) :: env) body2)
```

5.3 Correctness

We now discuss the correctness of the "efficient" semantics of §5.3, which our interpreter follows. The correctness argument is carried out in two steps. First, the method of §4 (extended to work on $\mathbf{Kl}(N)$, instead of $\mathbf{Set}^{\mathcal{W}}$) provides correctness with respect to $[\![-]\!]^v_{NR}$. Then, via the results of §3.3 we connect to the simple set models for laziness, and indeed to any other affine monad that supports the operations. (We state this formally as Theorem 26.)

We extend the proof of Section 4.3 to work in $\mathbf{Kl}(N^R)$ instead of $\mathbf{Alg}(R)$. Recall that there is an isomorphism of R-algebras $\varphi_A : [\![A]\!]^{\mathfrak{a}} \cong R([\![A]\!]^v_R)$ for any type A.

▶ **Lemma 24.** *For any type A, the functor $\mathbf{Alg}(R) \to \mathbf{Kl}(N^R) \xrightarrow{J} \mathbf{Alg}(R_N)$ turns φ_A into an isomorphism of R_N-algebras $\overline{\varphi_A} : [\![A]\!]^{\mathfrak{i}} \xrightarrow{\cong} R_N([\![A]\!]^v_N)$ (n.b. the codomain is $R([\![A]\!]^v_N)$).*

One can show the next two theorems by induction, but we avoid this because they follow from §6. We state them here for completeness of the first-order presentation. First we relate the interpreter semantics with the eager semantics with NR.

[6] One could directly use the `Choice` constructor, but our implementation additionally preserves the invariant that there are no redundant reads on the same name; this is correct up to $\equiv$. (This is similar to what happens with BDDs, but here the tree structure is more complex.)

▶ **Theorem 25.** *For $\Gamma \vdash M : B$, the following commutes in $\mathbf{Set}^{\mathcal{W}}$:*

$$\begin{array}{ccc} [\![\Gamma]\!]^{\mathsf{i}} & \xrightarrow{[\![M]\!]^{\mathsf{i}}} & N([\![B]\!]^{\mathsf{i}}) \\ \downarrow{\scriptstyle \overline{\varphi_\Gamma}} & & \downarrow{\scriptstyle N(\overline{\varphi_B})} \\ R([\![\Gamma]\!]^v) & \xrightarrow{[\![M]\!]^{v\star}} & NR([\![B]\!]^v) \end{array}$$

Next, consider an arbitrary affine monad T that supports Σ. By Corollary 12 there is a monad morphism $T^{\mathrm{aff}}_{\Sigma} \to T$ which induces a function $\psi_A : N([\![A]\!]^{\mathsf{i}})_{\mathbf{0}} \to T([\![A]\!]_{\mathbf{Set}})$ for every A (via $N\overline{\varphi_D}$).

▶ **Theorem 26.** *For $\Gamma \vdash M : B$, in the setting of the paragraph above, the diagram below commutes in* **Set***:*

$$\begin{array}{ccc} N([\![\Gamma]\!]^{\mathsf{i}})_{\mathbf{0}} & \xrightarrow{{[\![M]\!]^{\mathsf{i}}}^{\dagger}_{N}} & N([\![B]\!]^{\mathsf{i}})_{\mathbf{0}} \\ \downarrow{\scriptstyle \psi_\Gamma} & & \downarrow{\scriptstyle \psi_A} \\ T([\![\Gamma]\!]^v_{\mathbf{Set}}) & \xrightarrow{{[\![M]\!]^{v}}_{T}} & T([\![B]\!]^v_{\mathbf{Set}}) \end{array}$$

To give some intuition with the probabilistic signature, if $T = \mathrm{D}$: the codomain represents the set of decision diagrams (over any variables), and ψ_A collapses them to a distribution.

6 Extension to higher-order types

We extend the developments of this paper to a language with higher-order types. All of our semantic models are already equipped to support this, and implementing the interpreter is completely straightforward (§6.3). The challenge is to extend the correctness proof, which we do via a realizability argument (§6.1, §6.2).

Extended language and semantics We extend our language with higher-order, adding function types $A \to B$ and terms $\lambda x.\, M$ and $M\, N$. Say a type is a *data type* when it does not contain any arrows, and say a term $\Gamma \vdash M : B$ has a *first-order interface* when B and Γ consist only of data types (but M is otherwise unconstrained).

Given a strong monad T on a bicartesian closed category $\mathscr{C}$, the eager interpretation (of §2.2) is extended: for types, via $[\![A \to B]\!]^v = (T[\![B]\!]^v)^{[\![A]\!]^v}$; and for terms, via the currying isomorphism and evaluation map in $\mathscr{C}$.

This definition covers the two semantics that concern us here: the $[\![-]\!]^v_{NR}$ (taking $T = N \circ R$) and (with T the lifted monad $N^R : \mathbf{Alg}(R) \to \mathbf{Alg}(R)$). Unfolding the definitions (§5.1), we get: $[\![A \to B]\!]^v_{NR} = [\![A]\!]^v \to NR([\![B]\!]^v)$ $\qquad [\![A \to B]\!]^{\mathsf{i}} = [\![A]\!]^{\mathsf{i}} \multimap N([\![B]\!]^{\mathsf{i}})$.

6.1 Realizability model

The challenge of extending correctness. The correctness theorem for the first-order language (Theorem 26) does not hold in that form at higher-order types, because of possible effects nested in λ-abstractions, which are represented differently in $\mathbf{Kl}(NR)$ and in $\mathbf{Alg}(R_N)$. Our goal here is to show that Theorem 26 holds for any term M of the extended language *with a first-order interface*, even if M uses higher-order constructors.

Our proof is based on a realizability model that provides the right inductive invariant for the higher-order structure. The development is technical because our language has effects, and general sum types, both of which are a known source of complications with these

techniques [12, 17]. Our proof uses $\top\top$-*lifting* [30] (a.k.a *double orthogonality*) and we follow a presentation inspired by [45].

For readers unfamiliar with double-orthogonality we briefly motivate: traditional logical relations or realizability techniques are based on a predicate (or relation) on terms, defined by induction on types, in a uniform style. Extending this method to handle sum types and effects is a known challenge.[7] The method of $\top\top$-lifting involves simultaneously defining a predicate on terms, and a predicate on *contexts*. For negative types, contexts are seen as primitive, while for positive types, terms are primitive.

Pole and contexts. Realizability begins by specifying a *pole*: a set of terms that determines the proof goal. Our pole will be the set of closed terms of data type, whose interpretations $[\![-]\!]^{\mathfrak{a}}$ and $[\![-]\!]^{v}$ agree modulo the mediating isomorphisms of $\overline{\varphi}$ of Lemma 24:

$$\text{Pole} = \{\vdash M : D \mid [\![M]\!]^{v} = N\overline{\varphi}_D \circ [\![M]\!]^{\mathfrak{i}}\}.$$

The set of contexts is defined by the following grammar:

$$C \quad ::= \quad \bullet \mid C\,M \mid \texttt{fst}\ C \mid \texttt{snd}\ C \mid \texttt{case}\,C : [x.N_1 \mid y.N_2]$$

Contexts can be typed: we write $\vdash C : A \Rightarrow B$ when $x : A \vdash C[x] : B$. Let $\text{Term}(A)$ denote the set of closed terms of type A, and $\text{Ctx}(A)$ the set of contexts such that $\vdash C : A \Rightarrow D$ for some data type D. Since contexts are merely terms with a free variable, they can be interpreted: $\vdash C : A \Rightarrow D$ gives $[\![C]\!]^{\mathfrak{i}} : [\![A]\!]^{\mathfrak{a}} \multimap N([\![D]\!]^{\mathfrak{i}})$ and $[\![C]\!]^{v} : [\![A]\!]^{v} \to NR([\![D]\!]^{v})$.

Interaction and orthogonality Informally, a context $C \in \text{Ctx}(A)$ interacts well with $M \in \text{Term}(A)$ when $C[M] \in \text{Pole}$. This induces a notion of orthogonality:

$$\begin{array}{rrcl} \text{for } X \subseteq \text{Term}(A). & X^{\perp} & = & \{C \in \text{Ctx}(A) \mid \forall M \in X, C[M] \in \text{Pole}\} \\ \text{for } X \subseteq \text{Ctx}(A). & X^{\perp} & = & \{M \in \text{Term}(A) \mid \forall C \in X, C[M] \in \text{Pole}\} \end{array}$$

We now define, for any type, a pair of sets $(|A|, \|A\|)$, respectively containing types and contexts, such that $|A|^{\perp} = \|A\|$ and $\|A\|^{\perp} = |A|$. For negative type we set $|A| = \|A\|_V^{\perp}$ and $\|A\| = |A|^{\perp}$ where $\|1\|_V = \emptyset$ and

$$\begin{array}{rcl} \|A \times B\|_V & = & \{C[\texttt{fst}\ \bullet] \mid C \in \|A\|\} \cup \{C[\texttt{snd}\ \bullet] \mid C \in \|B\|\} \\ \|A \to B\|_V & = & \{C[\bullet N] \mid N \in |A| \wedge C \in \|B\|\}, \end{array}$$

and for positive types we set $\|A\| = |A|_V^{\perp}$ and $|A| = \|A\|^{\perp}$, where

$$|A + B|_V \quad = \quad \{\texttt{inl}\ M \mid M \in |A|\} \cup \{\texttt{inr}\ M \mid M \in |B|\}.$$

6.2 Fundamental lemma and correctness proof

We show the *fundamental lemma* for our realizability model, which implies the desired correctness result (Theorem 32).

In the proof of the fundamental lemma, the key argument is for the case of effects. Recall (§2.1) that the construct $\sigma(M_1, ..., M_n)$, for $\sigma \in \Sigma$ of arity n, is equivalent to $\text{case}\,\text{run}(\sigma)\,[i.M_i]_{i\in n}$, with $\text{run}(\sigma) : n$. Note that $[\![n]\!]^{\mathfrak{i}} = R(n) = R([\![n]\!]^{v})$, and it is easy to see that $\overline{\varphi_n} : [\![n]\!]^{\mathfrak{i}} \to R([\![n]\!]^{v})$ is the identity. Moreover $[\![\text{run}(\sigma)]\!]^{\mathfrak{i}} = [\![\text{run}(\sigma)]\!]^{v} : 1 \to NR(n)$.

---

[7] One view is that uniformity breaks because arrows and products are 'negative' (characterized by destructors) while sums types are 'positive' (defined by constructors).

▶ **Lemma 27.** *For $n \in \mathbb{N}$ and C a context in $\|n\|$, the diagram below commutes in* $\mathbf{Set}^{\mathcal{W}}$.

$$\begin{array}{ccc} [\![n]\!]^{\mathsf{i}} & \xrightarrow{[\![C]\!]^{\mathsf{i}}} & N([\![D]\!]^{\mathsf{i}}) \\ \| & & \downarrow{\scriptstyle N(\overline{\varphi_D})} \\ R([\![n]\!]^v) & \xrightarrow{[\![C]\!]^{v}_{NR}{}^{\star}} & NR([\![D]\!]^v) \end{array}$$

▶ **Lemma 28.** *For any $\sigma \in \Sigma$ of arity n, we have* $\mathrm{run}(\sigma) \in |k|$.

We can now state the fundamental lemma.

▶ **Theorem 29** (Fundamental Lemma)**.** *For a term-in-context $\Gamma \vdash M : B$, where $\Gamma = x_1 : A_1, .., x_n : A_n$, if for all $i \leq n$, $N_i \in |A_i|$, then $(\mathtt{let}\, \vec{x}_i = \vec{N}_i \,\mathtt{in}\, M) \in |B|$.*

Proof. For all 'pure' constructs, the proof follows the standard pattern (e.g. [45]). For effects, we use that $\sigma(M_1, ..., M_n) = \mathtt{case}\, \mathrm{run}(\sigma) : [\mathrm{in}_k() \,.\, M_k]$ and conclude from Lemma 28. ◀

Correctness for first-order interfaces. With the fundamental lemma, we deduce our correctness theorems from the next lemma, proved by a straightforward induction on D.

▶ **Lemma 30.** *For any data type D, the identity context $\bullet$ is in $\|D\|$.*

▶ **Theorem 31.** *For $\vdash M : D$, then $N\overline{\varphi_D}([\![M]\!]^{\mathsf{i}}) = [\![M]\!]^v$.*

▶ **Theorem 32.** *Let $\Gamma \vdash M : D$ be a term with a first-order interface, but otherwise arbitrary. The diagram below commutes in* $\mathbf{Set}$.

$$\begin{array}{ccc} [\![\Gamma]\!]^{\mathsf{i}}(\mathbf{0}) & \xrightarrow{[\![M]\!]^{\mathsf{i}}_{\mathbf{0}}} & N([\![D]\!]^{\mathsf{i}}(\mathbf{0})) \\ \downarrow{\scriptstyle (\overline{\varphi_\Gamma})_{\mathbf{0}}} & & \downarrow{\scriptstyle N(\overline{\varphi_D})_{\mathbf{0}}} \\ R([\![\Gamma]\!]^v_{NR})(\mathbf{0}) & \xrightarrow{([\![M]\!]^v_{NR}{}^{\star})_{\mathbf{0}}} & NR([\![B]\!]^v_{NR})(\mathbf{0}) \end{array}$$

We conclude with the appropriate generalization of Theorem 26:

▶ **Theorem 33.** *For any affine monad T that supports Σ, if a term $\Gamma \vdash M : B$ has a first-order interface, then the diagram below commutes.*

$$\begin{array}{ccc} N([\![\Gamma]\!]^{\mathsf{i}})(\mathbf{0}) & \xrightarrow{([\![M]\!]^{\mathsf{i}}{}^{\dagger}_{N})_{\mathbf{0}}} & N([\![D]\!]^{\mathsf{i}})(\mathbf{0}) \\ \downarrow{\scriptstyle \psi_\Gamma} & & \downarrow{\scriptstyle \psi_B} \\ T([\![\Gamma]\!]^v) & \xrightarrow{[\![M]\!]^{v}_{T}{}^{\dagger}} & T([\![B]\!]^v) \end{array}$$

More informally: given inputs encoded in the intermediate representation, the lazy interpreter agrees with the eager semantics.

6.3 Extending the lazy interpreter

Extending the interpreter to the higher-order language is completely straightforward, using the higher-order types in OCaml and continuing to follow the semantics $[\![-]\!]^{\mathsf{i}}$. Functions are simply represented as another constructor on tree,

```
type tree = ... | Function of (tree -> tree)
```

and the interpretation of terms uses environments and closures.

We note that further optimizations are likely possible at that level: e.g. the model satisfies $(f +_c g)(x) = f(x) +_c g(x)$ and so, when evaluating a λ-abstraction $\lambda x. M$, one could examinate the tree produced by M and push any top-level operations before returning.

7 Discussion

Inspired by a practical challenge within the Botascopia project ([7], §1.1), we have made significant progress in the theoretical understanding of laziness. Most importantly, this work exposes the semantic structures involved in a lazy interpreter. Our contributions include a new semantic model for laziness, and formal correctness of the interpreter w.r.t. call-by-value for affine effects. We have focussed on the basic semantic building blocks, but we emphasize that there is some flexibility in practice, e.g. it could be useful for the interpreter to keep track explicitly of the partial assignment.

Related work around laziness and effects. The authors of [10] present a syntactic decomposition of a probabilistic choice operator in λ-calculus, and study encodings of various evaluation strategies. This was an inspiration for the present work, although laziness is not explicitly considered there. McDermott and Mycroft [33] propose an extended Call-By-Push-Value (ECBPV) calculus to study laziness and effects. ECBPV does not yet have a denotational semantics and it would be a natural step to develop our approach in that direction (indeed there is a model of CBPV based on our monad R_N). We also point out [34], which studies a proof technique for relating call-by-value and call-by-name at any type, based on a method similar to our mediating maps $[\![A]\!]^n \to [\![A]\!]^a \cong T([\![A]\!]^v)$. Although their method applies to higher-order types without an explicit logical relation, the authors mention specific difficulties in applying it to reader monads and supporting arbitrary sum types. Finally, lazy evaluation has been studied in different forms for the λ-calculus ([1, 2, 22, 37]), often without effects. An interesting direction will be to relate our three semantic categories to the three classes of terms in lazy λ-calculus.

Related work in efficient probabilistic programming. The representations of this paper are very reminiscent of Dice [13], a popular probabilistic language which performs a compilation to binary decision diagrams. In the same line of work, Roulette [36] extends Dice to a symbolic interpreter for discrete probability. Symbolic execution and laziness are a priori different but there are clear connection between the present work and the categorical semantics of Roulette [9, 29]. The latter goes further in understanding the logical aspects, but does not consider general algebraic effects.

LazyPPL [11] and Pluck [4] are lazy probabilistic languages justified by a denotational semantics based on infinite 'rose trees'. One difference is that an interpreter can only use infinite rose trees when the host language is already lazy (e.g. Haskell), and so this semantics hides the internal mechanism of laziness. Older but significant uses of laziness in probabilistic programming include [19, 21].

Perspectives and future work. Overall, this work is a step towards a full theory of effectful call-by-need, with an equational theory and categorical axiomatization.

On the practical side, we can already port these ideas to Botascopia, for significant gains. Another opportunity for improvement is to allow for sophisticated pattern matching à la OCaml, which would interact differently with an intermediate representation. Another key challenge is to support recursion in terms and types. We will study and implement canonical versions of our tree-like representations, in the spirit of *reduced ordered* BDDs. Our semantic models already track the variable order, and indeed our methods could even support a more refined programming style where users can access the order, as well as programmatic ways to impose eager evaluation for an operation.

References

1. Samson Abramsky and C-H Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, 1993. doi:10.1006/inco.1993.1044.
2. Zena M Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246, 1995. doi:10.1145/199448.199507.
3. Jon Beck. Distributive laws. In B. Eckmann, editor, *Seminar on Triples and Categorical Homology Theory*, pages 119–140, Berlin, Heidelberg, 1969. Springer Berlin Heidelberg.
4. Maddy Bowers, Alexander K. Lew, Joshua B. Tenenbaum, Armando Solar-Lezama, and Vikash K. Mansinghka. Stochastic lazy knowledge compilation for inference in discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1863–1887, 2025. URL: http://dx.doi.org/10.1145/3729325, doi:10.1145/3729325.
5. Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992. doi:10.1145/136035.136043.
6. Marta Cavallo Bunge. *Categories of set valued functors*. University of Pennsylvania, 1966.
7. Simon Castellan, Aurore Alcolei, Anne Atlan, Xavier Aubriot, Mathurin Bellot, Gurvan Cabot, Lysa Dahmani, Jos Käfer, Sophie Nadot, Sara Oort, Agnès Schermann-Legionnet, and Eric Tannier. Botascopia: a digital setup for generating low tech plant field guides. working paper or preprint, 2025. URL: https://hal.science/hal-05419216.
8. Simon Castellan, Jos Käfer, and Eric Tannier. Back to the trees: Identifying plants with Human Intelligence. In *LIMITS 2023 - Ninth Workshop on Computing within Limits*, pages 1–11, Virtuel - Online, France, jul 2023. LIMITS. URL: https://hal.science/hal-04121511, doi:10.21428/bf6fb269.265c52ce.
9. Jack Czenszak, John M Li, and Steven Holtzen. Towards symbolic execution for probability and non-determinism. 2025.
10. Ugo Dal Lago, Giulio Guerrieri, and Willem Heijltjes. Decomposing Probabilistic Lambda-Calculi. In *FOSSACS 2020 - Foundations of Software Science and Computation Structures - 23rd International Conference*, pages 136–156, Dublin, Ireland, apr 2020. URL: https://inria.hal.science/hal-03120783, doi:10.1007/978-3-030-45231-5_8.
11. Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. Affine monads and lazy structures for bayesian programming. oct 2022. URL: https://zenodo.org/record/7150943, doi:10.5281/ZENODO.7150943.
12. Marcelo P. Fiore and Alex K. Simpson. Lambda definability with sums via grothendieck logical relations. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 1999. doi:10.1007/3-540-48959-2_12.
13. Steven Holtzen, Guy Van Den Broeck, and Todd Millstein. Artifact for scaling exact inference for discrete probabilistic programs. *Zenodo*, 2020. URL: https://zenodo.org/record/4060132, doi:10.5281/ZENODO.4060132.
14. Bart Jacobs. Semantics of weakening and contraction. *Ann. Pure Appl. Log.*, 69(1):73–106, 1994. doi:10.1016/0168-0072(94)90020-5.
15. Bart Jacobs. Convexity, duality and effects. In *IFIP International Conference on Theoretical Computer Science*, pages 1–19. Springer, 2010. doi:10.1007/978-3-642-15240-5_1.
16. Younesse Kaddar and Sam Staton. A model of stochastic memoization and name generation in probabilistic programming: categorical semantics via monads on presheaf categories. *Electronic Notes in Theoretical Informatics and Computer Science*, 3, 2023. doi:10.46298/entics.12291.
17. Shin-ya Katsumata. A characterisation of lambda definability with sums via tt-closure operators. In Michael Kaminski and Simone Martini, editors, *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, volume 5213 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 2008. doi:10.1007/978-3-540-87531-4_21.

18 G. A. Kavvos. Adequacy for algebraic effects revisited. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):927–955, apr 2025. URL: http://dx.doi.org/10.1145/3720457, doi:10.1145/3720457.

19 Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages*, pages 360–384. Springer, 2009. doi:10.1007/978-3-642-03034-5_17.

20 Anders Kock. Bilinearity and cartesian-closed monads. *Mathematica Scandinavica*, 29(2):161–174, 1971. URL: http://www.jstor.org/stable/24491025.

21 Daphne Koller, David McAllester, and Avi Pfeffer. Effective bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997. URL: http://www.aaai.org/Library/AAAI/1997/aaai97-115.php.

22 John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154, 1993. doi:10.1145/158511.158618.

23 F William Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences*, 50(5):869–872, 1963.

24 F William Lawvere. Some algebraic problems in the context of functorial semantics of algebraic theories. In *Reports of the Midwest Category Seminar II*, pages 41–61, 1968.

25 Tom Leinster. *Higher operads, higher categories*. Number 298. Cambridge University Press, 2004.

26 Paul Blain Levy. *Call-By-Push-Value*. Springer Netherlands, Dordrecht, 2003. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233, doi:10.1007/978-94-007-0954-6.

27 Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, dec 2006. URL: http://dx.doi.org/10.1007/s10990-006-0480-6, doi:10.1007/s10990-006-0480-6.

28 PaulBlain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and computation*, 185(2):182–210, 2003. doi:10.1016/S0890-5401(03)00088-9.

29 John Li, Jack Czenszak, and Steven Holtzen. Categorical semantics of probabilistic symbolic execution. In *Proceedings of the 47th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '26)*, Boulder, CO, USA, 2026. ACM. doi:10.1145/3808343.

30 Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, pages 262–277, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. doi:10.1007/11417170_20.

31 Fred EJ Linton. Coequalizers in categories of algebras. In *Seminar on Triples and Categorical Homology Theory: ETH 1966/67*, pages 75–90. Springer, 2006.

32 Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1998.

33 Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In *ESOP*, pages 235–262, 2019. doi:10.1007/978-3-030-17184-1_9.

34 Dylan McDermott and Alan Mycroft. Galois connecting call-by-value and call-by-name. *Logical Methods in Computer Science*, 20, 2024. doi:10.46298/lmcs-20(1:13)2024.

35 Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

36 Cameron Moy, Jack Czenszak, John M. Li, Brianna Marshall, and Steven Holtzen. Roulette: A language for expressive, exact, and efficient discrete probabilistic programming. *Proc. ACM Program. Lang.*, 9(PLDI), jun 2025. doi:10.1145/3729334.

37 Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–81, 1994.

38 Peter W O'Hearn and Robert D Tennent. Semantics of local variables. *Applications of categories in computer science*, 177:217–238, 1992.

39 Andrew M Pitts and Ian DB Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *International Symposium on Mathematical Foundations of Computer Science*, pages 122–141. Springer, 1993. doi:10.1007/3-540-57182-5_8.

40 Gordon Plotkin and John Power. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002. doi:10.1007/3-540-45931-6_24.

41 Gordon Plotkin and John Power. Computational effects and operations: An overview. *Electronic Notes in Theoretical Computer Science*, 73:149–163, oct 2004. URL: http://dx.doi.org/10.1016/j.entcs.2004.08.008, doi:10.1016/j.entcs.2004.08.008.

42 Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, dec 2013. URL: http://dx.doi.org/10.2168/lmcs-9(4:23)2013, doi:10.2168/lmcs-9(4:23)2013.

43 John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740, 1972. doi:10.1145/800194.805852.

44 Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT press, 1993.

45 Pierre Évariste Dagand, Lionel Rieg, and Gabriel Scherer. Dependent pearl: Normalization by realizability, 2020. URL: https://arxiv.org/abs/1908.09123, arXiv:1908.09123, doi:10.48550/arXiv.1908.09123.