

Smart Choices and the Selection Monad

MARTÍN ABADI, Google Research, United States
GORDON D. PLOTKIN, Google Research, United States

Describing systems in terms of choices and of the resulting costs and rewards offers the promise of freeing algorithm designers and programmers from specifying how those choices should be made; in implementations, the choices can be realized by optimization techniques and, increasingly, by machine learning methods. We study this approach from a programming-language perspective. We define two small languages that support decision-making abstractions: one with choices and rewards, and the other additionally with probabilities. We give both operational and denotational semantics. The operational semantics combine the usual semantics of standard constructs with optimization over a space of possible executions. The denotational semantics, which are compositional and can also be viewed as an implementation by translation to a simpler language, rely on the selection monad. We establish that the two semantics coincide in both cases.

CONTENTS

Abstract	1
Contents	1
1 Introduction	2
2 The selection monad and algebraic operations	4
3 A general language with algebraic operations	7
3.1 Syntax	7
3.2 Operational semantics	8
3.3 Denotational semantics	10
3.4 Adequacy	11
3.5 Program equivalences	11
4 A language of choices and rewards	12
4.1 Syntax	12
4.2 Rewards and additional effects	12
4.3 Operational semantics	13
4.4 Denotational semantics	15
4.5 Adequacy	15
4.6 Program equivalences	16
5 Adding probabilities	18
5.1 Syntax	18
5.2 Rewards and additional effects	19
5.3 Operational semantics	19
5.4 Denotational semantics	22
5.5 Adequacy	22
5.6 Program equivalences	22
6 Conclusion	23
Acknowledgements	24
References	24

Authors' addresses: Martín Abadi, Google Research, 1600 Amphitheatre Parkway, Mountain View, California, 94043, United States, abadi@google.com; Gordon D. Plotkin, Google Research, 1600 Amphitheatre Parkway, Mountain View, California, 94043, United States, plotkin@google.com.

1 INTRODUCTION

Models and techniques for decision-making, such as Markov Decision Processes (MDPs) and Reinforcement Learning (RL), enable the description of systems in terms of choices and of the resulting costs and rewards. For example, an agent that plays a board game may be defined by its choices in moving pieces and by how many points these yield in the game. An implementation of such a system may aim to make the choices following a strategy that results in attractive costs and rewards, perhaps the best ones. For this purpose it may rely on classic optimization techniques or, increasingly, in forms of machine learning (ML). Deep RL has been particularly prominent in the last decade, but contextual bandits and ordinary supervised learning can also be useful.

In a programming context, several languages and libraries support choices, rewards, costs, and related notions in a general way (not specific to any application, such as a particular board game). McCarthy’s *amb* operator [McCarthy 1963] may be seen as an early example of a construct for making choices. More recent work includes many libraries for RL (e.g., [Budden et al. 2020]), languages for planning such as DTGolog [Boutilier et al. 2000] and some descendants (e.g., [Sanner 2011]) of the Planning Domain Definition Language [McDermott et al. 1998], a “credit-assignment” compiler for learning to search built on the Vowpal-Rabbit learning library [Chang et al. 2016], and Dyna [Vieira et al. 2017], a programming language for machine-learning applications based on MDPs. It also includes SmartChoices [Carbune et al. 2018], an “approach to making machine learning (ML) a first class citizen in programming languages”, one of the main inspirations for our work. SmartChoices and several other recent industry projects in this space (such as Spiral [Bychkovsky 2018]) extend mainstream programming languages and systems with the ability to make data-driven decisions by coding in terms of choices (or predictions) and feedback (in other words, perceived costs or rewards), and thus aim to have widespread impact on programming practice.

The use of decision-making abstractions has the potential to free algorithm designers and programmers from taking care of many details. For example, in an ordinary programming system, a programmer that implements quicksort should consider how to pick pivot elements and when to fall back to a simpler sorting algorithm for short inputs. Heuristic solutions to such questions abound, but they are not always optimal, and they require coding and sometimes maintenance when the characteristics of the input data or the implementation platform change. In contrast, SmartChoices enables the programmer to code in terms of choices and costs—or, equivalently, rewards, which we define as the opposite of costs—and to let the implementation of decision-making take care of the details [Carbune et al. 2018, Section 4.3]. As another example, consider the program in Figure 1 that does binary search in a sorted array. This pseudocode is a simplified

```
let binsearch(x : Int, a : Array[Int], l : Int, r : Int) =
  if l > r then None // the special value None represents failure
  else let m : [l,r] = choice in \ choose an integer in [l, r]
    if a[m] = x then m
    else cost(1); \ pay to recurse
      if a[m] < x then binsearch(x, a, m+1, r)
      else binsearch(x, a, l, m-1)
```

Fig. 1. Smart binary search

version of the one in [Carbune et al. 2018, Section 4.2], which also includes a way of recording observations of the context of choices (in this example, x , $a[l]$, and $a[r]$) that facilitate machine learning. Here, a choice determines the index m where the array is split. Behind the scenes, a clever implementation can take into account the distribution of the data in order to decide exactly how

to select m . For example, if x is half way between $a[1]$ and $a[r]$ but the distribution of the values in the array favors smaller values, then the selected m may be closer to r than to 1 . In order to inform the implementation, the programmer calls `cost`: each call to `cost` adds to the total cost of an execution, for the notion of cost that the programmer would wish to minimize. In this example, the total cost is the number of recursive calls. In other examples, the total cost could correspond, for instance, to memory requirements or to some application-specific metric such as the number of points in a game.

In this paper, we study decision-making abstractions from a programming-language perspective. We define two small languages that support such abstractions, one with choices and rewards, and the other one additionally with probabilities. In the spirit of SmartChoices (and in contrast with DTGolog and Dyna, for instance), the languages are mostly mainstream: only the decision-making abstractions are special. We give them both operational and denotational semantics. Their operational semantics combine the usual semantics of standard constructs with optimization. Despite the global character of optimization, our results include a tractable, more local formulation of their operational semantics (Theorems 2 and 5). Their denotational semantics are based on the selection monad [Bolt et al. 2018; Escardó 2015; Escardó and Oliva 2011; Escardó and Oliva 2017, 2010, 2012; Escardó et al. 2011; Hedges 2015], which we explain below. We establish that the two semantics coincide, proving adequacy results for both languages (Theorems 3 and 6). We also investigate program equivalences, which can justify program transformations, and develop proof systems for them. For example, one of our axioms concerns the commutation of choices and rewards. We consider (in particular, in Theorem 4) the soundness and completeness of the proof systems with respect to concepts of observational equivalence and semantic equivalence.

A brief, informal discussion of the semantics of `binsearch` may provide some intuition on the two semantics and on the role of the selection monad.

- If we are given the sequence of values picked by the choice construct in an execution of `binsearch`, a standard operational semantics straightforwardly allows us to construct the rest of the execution. We call this semantics the *ordinary operational semantics*. For each such sequence of values, the ordinary operational semantics implies a resulting total cost, and thus a resulting total reward. We define the *selection operational semantics* by requiring that the sequence of values be the one that maximizes this total reward.

Although they are rather elementary, these operational semantics are not always a convenient basis for reasoning, because (as usual for operational semantics) they are not compositional, and in addition the selection operational semantics is defined in terms of sequences of choices and accumulated rewards in multiple executions. On the other hand, the chosen values are simply plain integers.

- In contrast, in the denotational semantics, we look at each choice of `binsearch` as being made locally, without implicit reference to the rest of the execution or other executions, by a higher-order function of type $(\text{Int} \rightarrow R) \rightarrow \text{Int}$ (where Int is a finite set of machine integers), whose expected argument is a reward function f that maps each possible value of the choice to the corresponding reward of type R of the rest of the program. We may view f as a reward continuation. One possible such higher-order function is the function `argmax` that picks a value for the argument x for f that yields the largest reward $f(x)$. (There are different versions of `argmax`, in particular with different ways of breaking ties, but informally one often identifies them all.)

The type $(\text{Int} \rightarrow R) \rightarrow \text{Int}$ of this example is a simple instance of the selection monad, and `argmax` is an example of a selection function. More generally, we use types of the form $(X \rightarrow R) \rightarrow T(X)$, where X is any type, and T is another monad, in particular one that allows us to

represent rewards and probabilities. The monadic approach leads to a denotational semantics that is entirely compositional, and therefore facilitates proofs of program equivalences of the kind mentioned above. The denotational semantics may be viewed as an implementation by translation to a language in which there are no primitives for decision-making, and instead one may program with selection functions.

We develop our languages and their corresponding theory in stages. In Section 2, we review the selection monad and algebraic operations, and discuss algebraic operations for the selection monad. In Section 3, we present a general language with algebraic operations, give a basic adequacy theorem, and briefly discuss a calculus for program equivalences. This section is an adaptation of prior work, useful for our project but not specific to it. In Section 4, we define and study our first language with decision-making abstractions; it is a simply typed, higher-order λ -calculus, extended with a binary choice operation – or – and a construct for adding rewards. Finally, in Section 5, we proceed to our second language, which adds a probabilistic choice operator to the first. Probabilistic choices are not subject to optimization, but in combination with – or –, they enable us to imitate the choice capabilities of MDPs. Unlike MDPs, the language does not support infinite computations. We conjecture they can be treated via a metric approach to semantics; at any rate, there is no difficulty in adding a primitive recursion operator to the language, permitting MDP runs of arbitrary prescribed lengths.

In sum, we regard the main contributions of this paper as being (1): the connection between programming languages with decision-making abstractions and the selection monad, and (2): the definition and study of operational and denotational semantics for those languages, and the establishment of adequacy theorems for them. The adequacy theorems show that global operationally-defined optimizations can be characterized compositionally using a semantics based on the selection monad.

As described above, the selection operational semantics and the denotational semantics with the argmax selection function both rely on maximizing rewards. In many cases, optimal solutions are expensive. Even in the case of binsearch , an optimal solution that immediately picks m such that $a[m]$ equals x (without ever recursing) seems unrealistic. For efficiency, the optimization may be approximate and data-driven. In particular, as an alternative to the use of maximization in the selection operational semantics, we may sometimes be able to make the choices with contextual-bandit techniques, as in [Carbune et al. 2018, Section 4.2]. In the denotational semantics, assuming that R is the type of real numbers, we may use other selection functions instead of argmax . (The use of argmax is convenient, but our approach does not require it.) For example, instead of computing $\text{argmax}(f)$, we may approximate f by a differentiable function over the real numbers, represented by a neural network with learned parameters, and then find a local maximum of this approximation by gradient ascent. We have explored such approximations only informally so far; Section 6 briefly mentions aspects of this and other subjects for further work.

2 THE SELECTION MONAD AND ALGEBRAIC OPERATIONS

In this section we present material on the selection monad and on generic effects and algebraic operations, including a discussion of algebraic operations for the selection monad.

The selection monad

$$S(X) = (X \rightarrow R) \rightarrow X$$

introduced in [Escardó and Oliva 2010], is a strong monad available in any cartesian closed category, for simplicity discussed here only in the category of sets. One can think of $F \in S(X)$ as *selection functions* which choose an element $x \in X$, given a *reward function* $\gamma : X \rightarrow R$, viewing R

as a type of *rewards*. In a typical example, the choice x optimizes, maximizing the reward $\gamma(x)$ in some sense.

Computationally, we may understand $F \in S(X)$ as producing x given a *reward continuation* γ , i.e., a function giving the reward of the remainder of the computation. Indeed there is a close connection to the continuation monad $K(X) = (X \rightarrow R) \rightarrow R$: there is a monad morphism $S \rightarrow K$ sending $F \in S(X)$ to $\lambda\gamma : (X \rightarrow R). \gamma(F(\gamma))$. The selection monad has strong connections to logic. For example, as explained in [Escardó and Oliva 2012], whereas logic translations using K , taking R to be \perp , verify the double-negation law $\neg\neg P \supset P$, translations using S verify the instance $((P \supset R) \supset P) \supset P$ of Peirce's law. Again, with R the truth values, elements of $K(X)$ correspond to quantifiers, and elements of $S(X)$ correspond to selection operators, such as Hilbert's ε -operator.

The selection monad has unit $\eta_X : X \rightarrow S(X)$, where $\eta_S(x) = \lambda\gamma : X \rightarrow R. x$. The Kleisli extension $f^{\dagger s} : S(X) \rightarrow S(Y)$ of a function $f : X \rightarrow S(Y)$ is a little involved, so we explain it in stages. We need a function

$$S(X) \xrightarrow{f^{\dagger s}} S(Y)$$

Equivalently, we need to pick an element of Y , given a reward continuation $\gamma : Y \rightarrow R$ and a computation $F \in S(X)$, and do so as follows:

- Given γ , f yields a final result in Y as a function of X , viz. $res_x = f(x)(\gamma)$. We think of this as the optimal choice of $y \in Y$ starting from x (optimal given the reward function γ , that is).
- As we know the reward for such a final result, we know the reward for this particular application of f , viz. $rew_x = \gamma(res_x)$.
- This reward function is in turn the reward continuation of F , and we can choose the optimal element of X for this reward function, viz. $opt = F(rew)$.
- Now that we know the best choice of x , we use it to get the desired element of Y , viz. res_{opt} .

Intuitively, F chooses the $x \in X$ which will give the optimal $y \in Y$, and then f uses that x .

Writing all this out, we find:

$$\begin{aligned} f^{\dagger s} F \gamma &= res_{opt} \\ &= res_{F(rew)} \\ &= res_{F(\lambda x : X. \gamma(res_x))} \\ &= res_{F(\lambda x : X. \gamma(f(x)(\gamma)))} \\ &= f F(\lambda x : X. \gamma(f(x)(\gamma))) \gamma \end{aligned}$$

The selection monad has strength $(st_S)_{X,Y} : X \times S(Y) \rightarrow S(X \times Y)$ where:

$$(st_S)_{X,Y}(x, F) = \lambda\gamma : X \times Y \rightarrow R. \langle x, F(\lambda y : Y. \gamma(x, y)) \rangle$$

There is a generalization of the selection monad, incorporating a given strong monad T . This generalization will prove useful when we wish to combine various additional effects with selection. Suppose that R is a T -algebra with algebra map $\alpha : T(R) \rightarrow R$. Then, as essentially proved in [Escardó and Oliva 2017] for any cartesian closed category, we can define a strong monad S_T (which may just be written S , below) by setting:

$$S(X) = (X \rightarrow R) \rightarrow T(X)$$

It has unit $(\eta_S)_X : X \rightarrow S(X)$ where $(\eta_S)_X(x) = \lambda\gamma : X \rightarrow R. (\eta_T)_X(x)$. The Kleisli extension $f^{\dagger s} : S(X) \rightarrow S(Y)$ of a function $f : X \rightarrow S(Y)$ is given, analogously to the above, by:

$$f^{\dagger s} F \gamma = (\lambda x : X. f(x)(\gamma))^{\dagger T} (F(\lambda x : X. (\alpha \circ T(\gamma))(f(x)(\gamma))))$$

The selection monad has strength $(st_S)_{X,Y} : X \times S(Y) \rightarrow S(X \times Y)$ where:

$$(st_S)_{X,Y}(x, F) = \lambda\gamma : X \times Y \rightarrow R. (st_T)_{X,Y}(x, F(\lambda y : Y. \gamma(x, y)))$$

and there is a monad morphism $\theta : T \rightarrow S$ sending $x \in T(X)$ to $\lambda y : (X \rightarrow R). x$.

In order to be able to give semantics to effectual operations such as probabilistic choice, we use the apparatus of generic effects and algebraic operations in the category of sets discussed (in a much more general setting) in [Plotkin and Power 2003]. Suppose that M is a (necessarily strong) monad on the category of sets. A *generic effect* g of M of type (I, O) is just a Kleisli map:

$$g : I \rightarrow M(O)$$

and an algebraic operation is a family

$$\text{op}_X : I \times M(X)^O \rightarrow M(X)$$

natural with respect to Kleisli maps in the sense that the following diagram commutes for all $e : X \rightarrow M(Y)$:

$$\begin{array}{ccc} I \times M(X)^O & \xrightarrow{\text{op}_X} & M(X) \\ \downarrow I \times (e^\dagger_M)^O & & \downarrow e^\dagger_M \\ I \times M(Y)^O & \xrightarrow{\text{op}_Y} & M(Y) \end{array}$$

There is a 1-1 correspondence between generic effects and algebraic operations. Given g one may define:

$$\text{op}_X(i, a) = a^\dagger_M(g(i))$$

and conversely one sets:

$$g(i) = \text{op}_O(i, (\eta_M)_O)$$

Naturality implies a weaker but useful property, that the above diagram commutes for maps $T(f)$, for any $f : X \rightarrow Y$. In other words, such maps are homomorphisms $T(f) : T(X) \rightarrow T(Y)$, if we regard $T(X)$ and $T(Y)$ as algebras equipped with (any) corresponding algebraic operation instances.

We will generally obtain the algebraic operations we need via their generic effects. When I is a product $I_1 \times \dots \times I_m$ we obtain functions

$$\text{op}_X : (I_1 \times \dots \times I_m) \times M(X)^O \rightarrow M(X)$$

from which one obtains, in a standard way, semantically useful functions

$$\widetilde{\text{op}}_X : (M(I_1) \times \dots \times M(I_m)) \times M(X)^O \rightarrow M(X)$$

using Kleisli extension and the monoidal structure $(m_T)_{X,Y} : M(X) \times M(Y) \rightarrow M(X \times Y)$ induced by the monadic strength (see [Kock 1972]).

Returning to the selection monad S_T , we assume from now on that R is linearly ordered. There is then a natural generic effect g_{or} of S_T of type $(\mathbb{1}, \mathbb{B})$ for binary choice, where $\mathbb{B} = \{0, 1\}$. First, for any $\gamma : X \rightarrow R$ define a binary function \max_γ (written infix) on X by:

$$x \max_\gamma y = \begin{cases} x & \text{if } \gamma(x) \geq \gamma(y) \\ y & \text{otherwise} \end{cases}$$

Note that, for any γ , the function \max_γ is associative and idempotent (but not commutative).

Ignoring its trivial argument we then take $g_{\text{or}} \in S_T(\mathbb{B})$ to be:

$$g_{\text{or}}(\gamma) = 0 \max_\gamma 1$$

Thus g_{or} picks a boolean that maximizes the resulting reward, solving ties in favour of 0.

The corresponding algebraic operation family of “binary choice” functions is given by:

LEMMA 1. *We have:*

$$\text{or}_X(G, H)(\gamma) = G(\gamma) \max_{\alpha \in T(\gamma)} H(\gamma)$$

Note that the choice functions or_X are associative and idempotent as \max_γ always is. We could as well have used generics picking from any finite set, as in the example in Figure 1, with resulting choice functions of corresponding arity; however, binary choice is sufficiently illustrative.

We can also obtain algebraic operations for S_T from T ones. Given a T -generic effect $g : I \rightarrow T(O)$, $h = \theta_O \circ g$ is one for S_T , and the corresponding algebraic operations are naturally related:

LEMMA 2. *Let $(\text{op}_T)_X : I \times T(X)^O \rightarrow T(X)$ be the algebraic operation family corresponding to g . Then that corresponding to h is:*

$$(\text{op}_S)_X(i, F) = \lambda \gamma : X \rightarrow R. (\text{op}_T)_X(i, \lambda o : O. F o \gamma)$$

3 A GENERAL LANGUAGE WITH ALGEBRAIC OPERATIONS

The goal of this section is to give some definitions and results, in particular an adequacy theorem, for a general language with algebraic operations. We treat our two languages of later sections as instances of this language via such algebraic operations.

3.1 Syntax

We make use of a standard call-by-value λ -calculus equipped with algebraic operations. Our language is a convenient variant of the one in [Plotkin and Power 2001] (itself a variant of Moggi's computational λ -calculus [Moggi 1989]). The somewhat minor differences are that we allow a variety of basic types, our algebraic operations may have parameters, and we make use of general big-step transition relations as well as small-step ones.

Specifically, the types of the language are given by:

$$\sigma ::= b \mid \text{Unit} \mid \sigma \times \sigma \mid \sigma \rightarrow \sigma$$

where b ranges over a given set of *basic types*, $B\text{Types}$, including Bool , and the terms are given by:

$$\begin{aligned} M ::= & x \mid c \mid f(M_1, \dots, M_n) \mid \text{if } M \text{ then } M' \text{ else } M'' \mid \\ & \text{op}(M'_1, \dots, M'_n; M_1, \dots, M_n) \mid \\ & * \mid \langle M, M' \rangle \mid \text{fst}(M) \mid \text{snd}(M) \mid (\lambda x : \sigma. M) \mid MM' \end{aligned}$$

Here c ranges over a set of constants of given basic types, written $c : b$; f ranges over a given set of (first-order) function symbols each given first-order types, written $f : w \rightarrow b$, with $w \in B\text{Types}^*$ and $b \in B\text{Types}$; and op ranges over a given set of algebraic operation symbols each given parameter basic types and an arity, written $\text{op} : w; n$, with $w \in B\text{Types}^*$. The constants include $\text{tt} : \text{Bool}$ and $\text{ff} : \text{Bool}$; the function symbols include equality symbols $=_b : b \rightarrow \text{Bool}$ for all basic types.

We work up to α -equivalence, as usual, and free variables and substitution are also defined as usual. The typing rules are standard, and omitted, except for that for the algebraic operations, which, aside from their parameters, are polymorphic:

$$\frac{\Gamma \vdash M'_1 : b_1, \dots, \Gamma \vdash M'_n : b_n \quad \Gamma \vdash M_1 : \sigma, \dots, \Gamma \vdash M_n : \sigma}{\Gamma \vdash \text{op}(M'_1, \dots, M'_n; M_1, \dots, M_n) : \sigma} \quad (\text{op} : b_1 \dots b_n; n)$$

where $\Gamma = x_1 : \sigma_1, \dots, x_n : \sigma_n$ is an environment. We write $M : \sigma$ for $\vdash M : \sigma$ and say then that the (closed) term M is *well-typed*; such terms are the *programs* of our language.

3.2 Operational semantics

The operational semantics of programs is given in three parts: a small-step semantics, a big-step semantics, and an evaluation function. Following [Felleisen and Friedman 1987], we make use of evaluation contexts. The set of *values* is given by:

$$V ::= c \mid * \mid \langle V, V' \rangle \mid \lambda x : \sigma. M$$

where we restrict $\lambda x : \sigma. M$ to be closed. The *evaluation contexts* are given by:

$$\begin{aligned} \mathcal{E} ::= & [] \mid f(V_1, \dots, V_{k-1}, \mathcal{E}, M_{k+1}, \dots, M_n) \mid \text{if } \mathcal{E} \text{ then } M \text{ else } M' \mid \\ & \text{op}(V_1, \dots, V_{k-1}, \mathcal{E}, M'_{k+1}, \dots, M'_n; M_1, \dots, M_n) \\ & \langle \mathcal{E}, M' \rangle \mid \langle V, \mathcal{E} \rangle \mid \text{fst}(M) \mid \text{snd}(M) \mid \mathcal{E}M' \mid (\lambda x : \sigma. M)\mathcal{E} \end{aligned}$$

and are restricted to be closed. The *redexes* are defined by:

$$\begin{aligned} R ::= & f(c_1, \dots, c_n) \mid \text{if } \text{tt} \text{ then } M \text{ else } M' \mid \text{if } \text{ff} \text{ then } M \text{ else } M' \mid \\ & \text{op}(c_1, \dots, c_{n'}, M_1, \dots, M_n) \\ & \text{fst}(\langle M, M' \rangle) \mid \text{snd}(\langle M, M' \rangle) \mid (\lambda x : \sigma. M)V \end{aligned}$$

and are restricted to be closed. Any program is of one of two mutually exclusive forms: it is either a value V or else has the form $\mathcal{E}[R]$ for a unique evaluation context \mathcal{E} and a unique redex R .

We define two small-step transition relations on redexes, *ordinary* transitions and algebraic operation transitions:

$$R \rightarrow M' \quad \text{and} \quad R \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M' \quad (\text{op} : b_1 \dots b_{n'}; n \text{ and } i = 1, n)$$

The idea of the algebraic operation transitions is to indicate which argument of the operation is being followed, with which parameters. The definition of the first kind of transition is standard; we just mention that for each $f : b_1 \dots b_n \rightarrow b$ and constants $c_1 : b_1, \dots, c_n : b_n$, we assume given a constant $\text{val}_f(c_1, \dots, c_n) : b$, where, in the case of equality, we have:

$$\text{val}_{=b}(c_1, c_2) = \begin{cases} \text{tt} & \text{if } c_1 = c_2 \\ \text{ff} & \text{otherwise} \end{cases}$$

We then have the ordinary transitions:

$$f(c_1, \dots, c_n) \rightarrow c \quad (\text{val}_f(c_1, \dots, c_n) = c)$$

The algebraic operation transition relations are given by the following rule:

$$\text{op}(c_1, \dots, c_{n'}; M_1, \dots, M_n) \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M_i$$

We can then extend these transition relations to corresponding ordinary and algebraic operation transition relations on programs

$$M \rightarrow M' \quad \text{and} \quad M \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M'$$

To do so, we use evaluation contexts in a standard way in the following rules:

$$\frac{R \rightarrow M'}{\mathcal{E}[R] \rightarrow \mathcal{E}[M']} \quad \frac{R \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M'}{\mathcal{E}[R] \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} \mathcal{E}[M']}$$

These transition relations are all deterministic.

For any program M which is not a value, exactly one of two mutually exclusive possibilities holds:

- $M \rightarrow N$ for some N ; in this case N is determined and of the same type as M ;
- $M \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M_i$ for some $\text{op} : (w; n)$ and uniquely determined $c_1, \dots, c_{n'}$ and M_i ($i = 1, n$) of the same type as M .

We say a program M is *terminating* if there is no infinite chain of (small-step) transitions from M .

LEMMA 3. *Every program is terminating.*

PROOF. This is a standard computability argument; for some detail, see the proof of Theorem 1 in [Plotkin and Power 2001]. \square

Using the small-step relations one defines big-step ordinary and algebraic operation transition relations by:

$$\frac{M \rightarrow^* V}{M \Rightarrow V} \quad \frac{M \rightarrow^* M' \quad M \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M'}{M \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M'}$$

For any program M which is not a value, exactly one of two mutually exclusive possibilities holds:

- $M \Rightarrow V$ for some value V ; in this case V is determined and of the same type as M .
- $M \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M_i$ for some $\text{op} : (w; n)$ and uniquely determined $c_1, \dots, c_{n'}$ and M_i ($i = 1, n$) of the same type as M .

The big-step transition relations from a given program M form a finite tree with values at the leafs, all transitions, except for those leading to values being algebraic operation transitions, and with algebraic operation transitions of type $(w; n)$ branching n -fold. We write $\|M\|$ for the height of this tree.

Rather than use trees, we follow [Plotkin and Power 2001] and use *effect values* E . These give the same information and, conveniently, form a subset of our programs. They are defined as follows:

$$E ::= V \mid \text{op}(c_1, \dots, c_{n'}; E_1, \dots, E_n)$$

(Our effect values are a finitary version of the interaction trees of [Xia et al. 2020]). Every program M can be given an effect value $\text{Op}(M)$ defined as follows using the big-step transition relations:

$$\text{Op}(M) = \begin{cases} V & \text{if } M \Rightarrow V \\ \text{op}(c_1, \dots, c_{n'}; \text{Op}(M_1), \dots, \text{Op}(M_n)) & \text{if } M \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M_i \text{ for } i = 1, n \end{cases}$$

This definition can be justified by induction on $\|M\|$. Note that $\text{Op}(E) = E$, for any effect value $E : \sigma$. Note too that the transitions of programs and their evaluations closely parallel each other, indeed we have:

$$M \Rightarrow V \iff \text{Op}(M) = V$$

and

$$M \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} M_i \iff \text{Op}(M) \xrightarrow[\text{op}_i]{c_1, \dots, c_{n'}} \text{Op}(M_i)$$

We next give a proof-theoretic account of this evaluation function to help us prove the basic adequacy of our operational semantics. There is a natural equational theory, with evident rules, which establishes judgments of the form $\Gamma \vdash M = N : \sigma$, where it is assumed that $\Gamma \vdash M : \sigma$ and $\Gamma \vdash N : \sigma$. The axioms are the small-step reductions for the redexes together with a commutation schema that algebraic operations commute with evaluation contexts; they are given in Figure 2.

PROPOSITION 1. *For any program $M : \sigma$ we have:*

$$\vdash M = \text{Op}(M) : \sigma$$

$$\begin{aligned}
& f(c_1, \dots, c_n) = c \quad (\text{val}_f(c_1, \dots, c_n) = c) \\
& \text{if } \text{tt} \text{ then } M \text{ else } M' = M \quad \text{if } \text{ff} \text{ then } M \text{ else } M' = M' \\
& \text{fst}(\langle M, M' \rangle) = M \quad \text{snd}(\langle M, M' \rangle) = M' \\
& (\lambda x : \sigma. M)V = M[V/x] \\
& \mathcal{E}[\text{op}(c_1, \dots, c_{n'}; M_1, \dots, M_n)] = \text{op}(c_1, \dots, c_{n'}; \mathcal{E}[M_1], \dots, \mathcal{E}[M_n])
\end{aligned}$$

Fig. 2. Axioms

3.3 Denotational semantics

The semantics of our language makes use of a given strong monad, following that of Moggi's computational λ -calculus [Moggi 1989]. In order to be able to give semantics to effectual operations we use the apparatus of generic effects and algebraic operations as discussed above. For the sake of simplicity we work in the category of sets, although the results go through much more generally, for example in a cartesian closed category with binary sums.

To give the semantics of our language a number of ingredients are needed. We assume given:

- a (necessarily) strong monad M on the category of sets,
- sets $\llbracket b \rrbracket$ for the basic types b , with $\llbracket \text{Bool} \rrbracket = \mathbb{B} =_{\text{def}} \{0, 1\}$,
- elements $\llbracket c \rrbracket$ of $\llbracket b \rrbracket$ for constants $c : b$, with $\llbracket \text{tt} \rrbracket = 1$ and $\llbracket \text{ff} \rrbracket = 0$,
- functions $\llbracket f \rrbracket : \llbracket b_1 \rrbracket \times \dots \times \llbracket b_n \rrbracket \rightarrow \llbracket b \rrbracket$ for function symbols $f : b_1 \dots b_n \rightarrow b$, and
- generic effects

$$g_{\text{op}} : \llbracket b_1 \rrbracket \times \dots \times \llbracket b_{n'} \rrbracket \rightarrow M(\llbracket n \rrbracket)$$

for algebraic operation symbols $\text{op} : (b_1 \dots b_{n'}; n)$.

We further assume that different constants receive different denotations (so we can think of them as just names for their denotations, similarly to how one thinks of numerals), and that the given denotations of function symbols are consistent with their operational semantics in that:

$$\text{val}_f(c_1, \dots, c_n) = c \implies \llbracket f \rrbracket(\llbracket c_1 \rrbracket, \dots, \llbracket c_n \rrbracket) = \llbracket c \rrbracket$$

With these ingredients, we can give our language its semantics. Types are interpreted by putting:

$$\begin{aligned}
\mathcal{M}[\llbracket b \rrbracket] &= \llbracket b \rrbracket \\
\mathcal{M}[\llbracket \sigma \times \tau \rrbracket] &= \mathcal{M}[\llbracket \sigma \rrbracket] \times \mathcal{M}[\llbracket \tau \rrbracket] \\
\mathcal{M}[\llbracket \sigma \rightarrow \tau \rrbracket] &= \mathcal{M}[\llbracket \sigma \rrbracket] \rightarrow M(\mathcal{M}[\llbracket \tau \rrbracket])
\end{aligned}$$

To every term

$$\Gamma \vdash N : \sigma$$

we associate a function

$$\mathcal{M}[\Gamma \vdash N : \sigma] : \mathcal{M}[\Gamma] \rightarrow M(\mathcal{M}[\llbracket \sigma \rrbracket])$$

where $\mathcal{M}[\llbracket x_1 : \sigma_1, \dots, x_n : \sigma_n \rrbracket] =_{\text{def}} \mathcal{M}[\llbracket \sigma_1 \rrbracket] \times \dots \times \mathcal{M}[\llbracket \sigma_n \rrbracket]$. When the typing $\Gamma \vdash N : \sigma$ is understood, we generally write $\mathcal{M}[\llbracket N \rrbracket]$ rather than $\mathcal{M}[\Gamma \vdash N : \sigma]$.

The semantic clauses for conditionals and the product and function space expressions are standard, and we omit them. For constants $c : b$ we put:

$$\mathcal{M}[\llbracket c \rrbracket](\rho) = (\eta_M)_{\llbracket b \rrbracket}(\llbracket c \rrbracket)$$

For function symbol applications $f(M_1, \dots, M_n)$, where $f : b_1 \dots b_n \rightarrow b$, we put:

$$\mathcal{M}[\llbracket f(M_1, \dots, M_n) \rrbracket](\rho) = \llbracket f \rrbracket^{\sim}(\llbracket \mathcal{M} \rrbracket(M_1)(\rho), \dots, \llbracket \mathcal{M} \rrbracket(M_n)(\rho))$$

where

$$\llbracket f \rrbracket^{\sim} : M(\llbracket b_1 \rrbracket) \times \dots \times M(\llbracket b_n \rrbracket) \rightarrow M(\llbracket b \rrbracket)$$

is obtained from $\llbracket f \rrbracket$ in a standard way via Kleisli extension and the monadic monoidal structure. For algebraic operation terms $\Gamma \vdash \text{op}(M'_1, \dots, M'_{n'}; M_1, \dots, M_n) : \sigma$, where $\text{op} : (b_1 \dots b_{n'}; n)$, we make use of the algebraic operation family

$$\text{op}_X : (\llbracket b_1 \rrbracket \times \dots \times \llbracket b_{n'} \rrbracket) \times M(X)^{[n]} \rightarrow M(X)$$

corresponding to the generic effects g_{op} and put:

$$\mathcal{M}[\llbracket \text{op}(M'_1, \dots, M'_{n'}; M_1, \dots, M_n) \rrbracket](\rho) = \text{op}_{\llbracket \sigma \rrbracket}(\langle \mathcal{M}[\llbracket M'_1 \rrbracket](\rho), \dots, \mathcal{M}[\llbracket M'_{n'} \rrbracket](\rho) \rangle \langle \mathcal{M}[\llbracket M_1 \rrbracket](\rho), \dots, \mathcal{M}[\llbracket M_n \rrbracket](\rho) \rangle)$$

We further give values $V : \sigma$ an *effect-free* semantics $\llbracket V \rrbracket \in \mathcal{M}[\llbracket \sigma \rrbracket]$, extending that for constants:

$$\begin{aligned} \llbracket \langle V, V' \rangle \rrbracket &= \langle \llbracket V \rrbracket, \llbracket V' \rrbracket \rangle \\ \llbracket \lambda x : \tau. N \rrbracket &= \mathcal{M}[\llbracket x : \tau \vdash N : \tau' \rrbracket] \end{aligned}$$

(where in the second line τ' is determined as $\lambda x : \tau. N$ is well-typed). This effect-free semantics of values $V : \sigma$ determines their denotational semantics:

$$\mathcal{M}[\llbracket V \rrbracket](\rho) = (\eta_M)_{\mathcal{M}[\llbracket \sigma \rrbracket]}(\llbracket V \rrbracket)$$

Below, we may regard the effect-free semantics as providing functions:

$$\llbracket - \rrbracket_\sigma : \text{Val}_\sigma \rightarrow \mathcal{M}[\llbracket \sigma \rrbracket]$$

where Val_σ is the set of values of type σ .

3.4 Adequacy

Our proof system is consistent relative to our denotational semantics:

LEMMA 4. *If $\Gamma \vdash M = N : \sigma$ then $\mathcal{M}[\llbracket M \rrbracket] = \mathcal{M}[\llbracket N \rrbracket]$.*

The naturality condition with respect to Kleisli morphisms for algebraic operations is used here to establish the soundness of the commutation schema.

Our basic adequacy theorem now follows immediately from Proposition 1 and Lemma 4:

THEOREM 1. *For any program N we have: $\mathcal{M}[\llbracket N \rrbracket] = \mathcal{M}[\llbracket \text{Op}(N) \rrbracket]$.*

This adequacy theorem differs from the usual ones where the denotational semantics determines termination and the denotation of any final result; further, for basic types it generally determines the value produced by the operational semantics. In our case the first part is not relevant as terms always terminate. We do have that the denotational semantics determines the denotation of any final result. For basic types (as at any type) it determines the effect values produced up to their denotation, though the extent of that determination will depend on the choice of the generic effects.

3.5 Program equivalences

The equational system described above, in Section 3.2, helps prove adequacy, but is too weak to provide a means of reasoning about our calculus. A suitable consistent and complete system for Moggi's computational λ -calculus was given in [Moggi 1989]. As well as equations $M = N : \sigma$ (whose type we may omit) it has predicates $M \downarrow_\sigma$, read as “ M is pure (i.e., effect-free).” One can substitute a term M for a variable only if one can prove $M \downarrow_\sigma$. It is straightforward to adapt this system to our calculus (details omitted). For the algebraic operations one adds two equations, one:

$$g(\text{op}(x_1, \dots, x_{n'}; M_1, \dots, M_n)) = \text{op}(x_1, \dots, x_{n'}; g(M_1), \dots, g(M_n)) \quad (\text{no } x_i \in \text{any FV}(M_j))$$

expressing their naturality (and generalizing the commutation schema of Figure 2), and the other:

$$\text{op}(M'_1, \dots, M'_{n'}; M_1, \dots, M_n) = \text{let } x_1 : b_1, \dots, x_{n'} : b_{n'} \text{ be } M'_1, \dots, M'_{n'} \text{ in } \text{op}(x_1, \dots, x_{n'}; M_1, \dots, M_n)$$

(where $\text{op} : (b_1 \dots b_n; n)$) expressing the order of evaluation of the parameter arguments of op . The resulting system is consistent relative to our semantics, and the question of completeness relative to a general categorical semantics would be off interest. In later sections, we add equational axioms, such as an associativity axiom, for the particular algebraic operations investigated there. We write

$$\Gamma \vdash_{\text{Ax}} M = N : \sigma \quad \text{and} \quad \Gamma \vdash_{\text{Ax}} M \downarrow_\sigma$$

to mean that $M = N$ (resp. $M \downarrow_\sigma$) is provable from a given set of axioms Ax (where $\Gamma \vdash M : \sigma$ and $\Gamma \vdash N : \sigma$). The truth $\Gamma \models M = N : \sigma$ and $\Gamma \models M \downarrow_\sigma$ of assertions $\Gamma \vdash M = N : \sigma$ and $\Gamma \vdash M \downarrow_\sigma$ is defined as $\mathcal{M}[\llbracket M \rrbracket] = \mathcal{M}[\llbracket N \rrbracket]$ and $\exists a \in \mathcal{M}[\llbracket \sigma \rrbracket]. \mathcal{M}[\llbracket M \rrbracket] = \eta_M(a)$. Then we have *equational consistency*, meaning that, if the axioms are true, then:

$$\Gamma \vdash_{\text{Ax}} M = N : \sigma \implies \Gamma \models M = N : \sigma$$

and the analogous *purity consistency* also holds.

An alternate approach, well worth pursuing, would be to use instead the (purely equational) fine-grained variant of the computational λ -calculus: see [Levy et al. 2003].

4 A LANGUAGE OF CHOICES AND REWARDS

Building on the framework of Section 3, in this section we define and study a language with constructs for choices and rewards.

4.1 Syntax

For our language we assume available: a basic type Rew with $\llbracket \text{Rew} \rrbracket = \mathbb{R}$; a constant $0 : \text{Rew}$; function symbols $+$: $\text{Rew} \text{Rew} \rightarrow \text{Rew}$ and \leq : $\text{Rew} \text{Rew} \rightarrow \text{Bool}$; and two algebraic operation symbols: a *choice operation* $\text{or} : (\varepsilon; 2)$ to make choices, and a *reward operation* $\text{reward} : (\text{Rew}; 1)$, to prescribe rewards. We leave any other basic type symbols, constants, or function symbols unspecified.

We may use infix for $+$ and \leq . Similarly, we may use infix notations $M_0 \text{or} M_1$ or $N \cdot M$ for the algebraic operation expressions $\text{or}(M_0, M_1)$ and $\text{reward}(N; M)$. The signature $\text{or} : (\varepsilon; 2)$ means that $M_0 \text{or} M_1$ has the same type as M_0 and M_1 , which must be the same. The signature $\text{reward} : (\text{Rew}; 1)$ means $N \cdot M$ has the same type as M and that N must be of type Rew . For example, assuming that 5 and 6 are two constants of type Rew , we may write the tiny program:

$$(5 \cdot \text{tt}) \text{or} (6 \cdot \text{ff})$$

Intuitively, this program could potentially return either tt or ff , with respective rewards 5 and 6. In the intended semantics that maximizes rewards, then, the program returns ff with reward 6.

4.2 Rewards and additional effects

We assume that the linearly ordered set of rewards \mathbb{R} is also has a commutative monoid structure, written additively, and that this addition preserves and reflects the order, in that:

$$r \leq s \iff r + t \leq s + t \quad (r, s, t \in \mathbb{R})$$

For example, \mathbb{R} could be the reals with addition, or the positive reals with multiplication, in either case with the usual order.

We use the rewards, so equipped, for the monad employed to handle additional effects, as indicated in Section 2. (This monad will be useful for the theory of the operational semantics of our language, so is discussed here.) We take T to be the writer monad $T(X) = \mathbb{R} \times X$, using the assumed monoid structure on \mathbb{R} ; the monoid addition provides a natural T -algebra structure $\alpha : T(\mathbb{R}) \rightarrow \mathbb{R}$.

The generic effect $g_{\text{reward}} : R \rightarrow T(\mathbb{I})$, where $g_{\text{reward}}(r) =_{\text{def}} \langle r, * \rangle$, induces the algebraic operation family $\text{reward}_X : R \times T(X) \rightarrow T(X)$ where:

$$(\text{reward}_T)_X(r, \langle s, x \rangle) = \langle r + s, x \rangle$$

We generally write applications of $(\text{reward}_T)_X$ using an infix operator, \cdot_X .

4.3 Operational semantics

While the operational semantics of Section 3 is ordinary and does not address optimization, the *selection operational semantics* selects an optimal choice strategy, as suggested in the Introduction. Below we prove an adequacy result relative to a denotational semantics using the selection monad S_T . We thereby give a compositional account of a global quantity: the optimal reward of a program.

We employ a version of argmax . Given a finite linearly-ordered set S and a reward function $f : S \rightarrow R$, we set $\text{argmax}(f)$ to be the least $s \in S$ maximizing $f(s)$. So, linearly ordering S by:

$$s \preceq_f s' \iff (f(s) > f(s') \vee (f(s) = f(s') \wedge s \leq s'))$$

the selection is of the least element in this linear order. It will be convenient to use the notation $\text{argmax } s : S. e$ for $\text{argmax}(f)$ where $f(s) =_{\text{def}} e$, and $e \in R$ when $s \in S$.

We next define strategies for effect values. The idea is to view an effect value $E : \sigma$ as a one-player game for Player. The subterms of E are the positions of the game. In particular:

- if E is a value, then E is a final position and the reward is 0;
- if $E = \text{or}(E_0, E_1)$ then Player can choose whether to move to the position E_0 or E_1 ; and
- if $E = c \cdot E' : \sigma$ then Player moves to E' and c is added to the final reward.

The finite set $\text{Str}(E)$ of strategies of E are defined by the following rules, writing $s : E$ for $s \in \text{Str}(E)$:

$$* : V \quad \frac{s : E_0}{0s : \text{or}(E_0, E_1)} \quad \frac{s : E_1}{1s : \text{or}(E_0, E_1)} \quad \frac{s : E}{s : c \cdot E}$$

For any effect value $E : \sigma$, the outcome $\text{Out}(s, E) \in R \times \text{Val}_\sigma$ of a strategy $s : E$ is defined by:

$$\begin{aligned} \text{Out}(*, V) &= \langle 0, V \rangle \\ \text{Out}(0s, E_0 \text{ or } E_1) &= \text{Out}(s, E_0) \\ \text{Out}(1s, E_0 \text{ or } E_1) &= \text{Out}(s, E_1) \\ \text{Out}(s, c \cdot E) &= \llbracket c \rrbracket \cdot \text{Out}(s, E) \end{aligned}$$

We can then define the reward of such a strategy by:

$$\text{Rew}(s, E) = \pi_1(\text{Out}(s, E))$$

Note that $\pi_1 : R \times X \rightarrow R$ can be written as $\alpha \circ T(0)$, setting 0 to be the constantly 0 reward function.

As there can be more than one strategy maximizing the reward on a game, we need a way of choosing between them. We therefore define a linear order \leq_E on the strategies of a game $E : \sigma$:

- Game is V :

$$* \leq_V *$$

- Game is $E_0 + E_1$:

$$(i, s) \leq_{E_0 + E_1} (j, s') \iff \begin{aligned} &i < j \quad \vee \\ &i = j = 0 \wedge s \leq_{E_0} s' \quad \vee \\ &i = j = 1 \wedge s \leq_{E_1} s' \end{aligned}$$

- Game is $c \cdot E$:

$$s \leq_{c \cdot E} s' \iff s \leq_E s'$$

We can now give our selection operational semantics: for $M : \sigma$ we define $\text{Op}_s(M) \in \mathbb{R} \times \text{Val}_\sigma$ by:

$$\text{Op}_s(M) = \text{Out}(\text{argmax } s : \text{Op}(M). \text{Rew}(s, \text{Op}(M)), \text{Op}(M))$$

meaning that we take the $\text{Op}(M)$ -strategy maximizing the reward, and if there is more than one such strategy, we take the least, according to the $\text{Op}(M)$ -strategy linear order $\leq_{\text{Op}(M)}$.

While the operational semantics has been defined by a global optimization over all strategies, it can be equivalently given locally without reference to any strategies. We need two lemmas:

LEMMA 5. *Given functions $X \xrightarrow{g} Y \xrightarrow{\gamma} \mathbb{R}$, for all $u, v \in X$ we have $g(u \max_{Y \circ g} v) = g(u) \max_Y g(v)$.*

LEMMA 6. *(First argmax lemma) Let $S_0 \cup S_1$ split a finite linear order S into two with $S_0 < S_1$ (the latter in the sense that $s_0 < s_1$ for all $s_0 \in S_0$ and $s_1 \in S_1$). Then, for all $f : X \rightarrow \mathbb{R}$ we have:*

$$\text{argmax } s : S_0 \cup S_1. f(s) = (\text{argmax } s : S_0. f(s)) \max_f (\text{argmax } s : S_1. f(s))$$

We now have our local characterization of the operational semantics:

THEOREM 2. *For well-typed effect values we have:*

- (1) $\text{Op}_s(V) = \langle 0, V \rangle (= \eta_T(V))$
- (2) $\text{Op}_s(E_0 \text{ or } E_1) = \text{Op}_s(E_0) \max_{\pi_1} \text{Op}_s(E_1)$
- (3) $\text{Op}_s(c \cdot E) = \llbracket c \rrbracket \cdot_T \text{Op}_s(E)$

PROOF. We just prove (2) and (3). For (2), we calculate:

$$\begin{aligned} \text{Op}_s(E_0 \text{ or } E_1) &= \text{Out}(\text{argmax } s : E_0 \text{ or } E_1. \text{Rew}(s, E_0 \text{ or } E_1), E_0 \text{ or } E_1) \\ &= \text{Out}\left(\left(\begin{array}{c} \text{argmax } 0s : E_0 \text{ or } E_1. \text{Rew}(0s, E_0 \text{ or } E_1) \\ \max_{\text{Rew}(-, E_0 \text{ or } E_1)} \\ \text{argmax } 1s : E_0 \text{ or } E_1. \text{Rew}(1s, E_0 \text{ or } E_1) \end{array}\right), E_0 \text{ or } E_1\right) \\ &\quad \text{(by the first argmax lemma (Lemma 6))} \\ &= \text{Out}\left(\left(\begin{array}{c} \text{argmax } 0s : E_0 \text{ or } E_1. \text{Rew}(s, E_0) \\ \max_{\pi_1 \circ \text{Out}(-, E_0 \text{ or } E_1)} \\ \text{argmax } 1s : E_0 \text{ or } E_1. \text{Rew}(s, E_1) \end{array}\right), E_0 \text{ or } E_1\right) \\ &= \begin{array}{c} \text{Out}(\text{argmax } 0s : E_0 \text{ or } E_1. \text{Rew}(s, E_0), E_0 \text{ or } E_1) \\ \max_{\pi_1} \\ \text{Out}(\text{argmax } 1s : E_0 \text{ or } E_1. \text{Rew}(s, E_1), E_0 \text{ or } E_1) \end{array} \\ &\quad \text{(by Lemma 5)} \\ &= \text{Out}(\text{argmax } s : E_0. \text{Rew}(s, E_0), E_0) \max_{\pi_1} \text{Out}(\text{argmax } s : E_1. \text{Rew}(s, E_1), E_1) \\ &= \text{Op}_s(E_0) \max_{\pi_1} \text{Op}_s(E_1) \end{aligned}$$

And for (3) we calculate:

$$\begin{aligned} \text{Op}_s(c \cdot E) &= \text{Out}(\text{argmax } s : c \cdot E. \text{Rew}(s, c \cdot E), c \cdot E) \\ &= \llbracket c \rrbracket \cdot \text{Out}(\text{argmax } s : c \cdot E. \llbracket c \rrbracket + \text{Rew}(s, E), E) \\ &= \llbracket c \rrbracket \cdot \text{Out}(\text{argmax } s : c \cdot E. \text{Rew}(s, E), E) \\ &= \llbracket c \rrbracket \cdot \text{Out}(\text{argmax } s : E. \text{Rew}(s, E), E) \\ &= \llbracket c \rrbracket \cdot \text{Op}_s(E) \end{aligned}$$

where the fourth equality holds as the monoid preserves and reflects the ordering of \mathbb{R} . \square

4.4 Denotational semantics

In order to define a corresponding denotational semantics, we take T to be the (strong) writer monad $R \times -$, as discussed above, with T -algebra $+ : R \times R \rightarrow R$, so we have a strong monad

$$S(X) = (X \rightarrow R) \rightarrow R \times X$$

as described in Section 2. We use this monad to give the denotational semantics

$$\mathcal{S}[\![M]\!] : \llbracket \Gamma \rrbracket \rightarrow \llbracket \sigma \rrbracket \quad (\text{for } \Gamma \vdash M : \sigma)$$

of our language, following the pattern explained in the previous section. For the semantics of basic types, constants, and function symbols, we assume: $\llbracket \text{Rew} \rrbracket = R$; $\llbracket c \rrbracket$ is as before, for $c : \text{Rew}$; and $+$ and \leq are the monoid operation and ordering on R . Turning to the algebraic operation symbols, for or we use the algebraic operation family or_X given in Section 2, and for reward we take the algebraic operation family $(\text{reward}_S)_X$ induced by the $(\text{reward}_T)_X$, so:

$$(\text{reward}_S)_X(r, G)(\gamma) = (\text{reward}_T)_X(r, G\gamma) = \langle r + \pi_1(G\gamma), \pi_2(G\gamma) \rangle$$

4.5 Adequacy

We now aim to prove that the selection operational semantics coincides with its denotational semantics. This will be our *selection adequacy theorem*.

We need some notation to help connect the operational semantics of programs with their denotations. We write $\llbracket \langle r, V \rangle \rrbracket$ for $\langle r, \llbracket V \rrbracket \rangle$ (for $r \in R, V : \sigma$). Note that $\llbracket \langle r, V \rangle \rrbracket = T(\llbracket - \rrbracket_\sigma)(\langle r, V \rangle)$.

LEMMA 7. *For any effect value $E : \sigma$ we have:*

$$\mathcal{S}[\![E]\!](0) = \llbracket \text{Op}_s(E) \rrbracket$$

PROOF. We proceed by structural induction on E , omitting a case. Suppose $E = \text{or}(E_0, E_1)$. Then:

$$\begin{aligned} \mathcal{S}[\![\text{or}(E_0, E_1)]\!](0) &= \mathcal{S}[\![E_0]\!](0) \max_{\alpha \circ T(0)} \mathcal{S}[\![E_1]\!](0) && (\text{by Lemma 1}) \\ &= \llbracket \text{Op}_s(E_0) \rrbracket \max_{\alpha \circ T(0)} \llbracket \text{Op}_s(E_1) \rrbracket && (\text{by induction hypothesis}) \\ &= T(\llbracket - \rrbracket_\sigma)(\text{Op}_s(E_0) \max_{\alpha \circ T(0) \circ T(\llbracket - \rrbracket_\sigma)} \text{Op}_s(E_1)) && (\text{by Lemma 5}) \\ &= T(\llbracket - \rrbracket_\sigma)(\text{Op}_s(E_0) \max_{\alpha \circ T(0)} \text{Op}_s(E_1)) \\ &= T(\llbracket - \rrbracket_\sigma)(\text{Op}_s(E_0 \text{ or } E_1)) && (\text{by Theorem 2, part (2)}) \\ &= \llbracket \text{Op}_s(E_0 \text{ or } E_1) \rrbracket \end{aligned}$$

Suppose instead that $E = c \cdot E'$. Then:

$$\begin{aligned} \llbracket \text{Op}_s(c \cdot E') \rrbracket &= T(\llbracket - \rrbracket_\sigma)(\llbracket c \rrbracket \cdot_T \text{Op}_s(E')) && (\text{by Theorem 2, part (3)}) \\ &= \llbracket c \rrbracket \cdot_T T(\llbracket - \rrbracket_\sigma)(\text{Op}_s(E')) && (\text{as } T(\llbracket - \rrbracket_\sigma) \text{ is a homomorphism}) \\ &= \llbracket c \rrbracket \cdot_T \mathcal{S}[\![E']\!](0) && (\text{by induction hypothesis}) \\ &= (\llbracket c \rrbracket \cdot_S \mathcal{S}[\![E']\!](0)) \\ &= \mathcal{S}[\![c \cdot E']\!](0) \end{aligned}$$

□

THEOREM 3 (SELECTION ADEQUACY). *For any program $M : \sigma$ we have:*

$$\mathcal{S}[\![M]\!](0) = \llbracket \text{Op}_s(M) \rrbracket$$

PROOF. We have:

$$\begin{aligned} \mathcal{S}[\![M]\!](0) &= \mathcal{S}[\![\text{Op}(M)]\!](0) && (\text{by Theorem 1}) \\ &= \llbracket \text{Op}_s(\text{Op}(M)) \rrbracket && (\text{by Lemma 7}) \\ &= \llbracket \text{Op}_s(M) \rrbracket && (\text{by the definition of } \text{Op}_s, \text{ and as } \text{Op}(\text{Op}(M)) = \text{Op}(M)) \end{aligned}$$

□

The theorem relates the compositional denotational semantics with the globally optimizing operational semantics. Whereas the former optimizes only locally, as witnessed by the semantics of `or`, the latter optimizes over all possible Player strategies. The theorem states that the rewards according to both semantics agree, and that the denotation of the value returned by the operational semantics is determined by the denotational semantics. (In the case of basic types, the denotations of values determine the values themselves—see Section 3—so the values returned by the operational semantics are determined by the denotational semantics.)

The theorem compares the operational semantics of a program with its denotational semantics relative to the zero-reward continuation. This is reasonable as the operational semantics of a program does not consider any continuation, so, in particular, no reward continuation. As rewards mount up additively, the zero-reward continuation is appropriate at the top level.

4.6 Program equivalences

A general notion of semantic equivalence was given in Section 3.5. Given a notion of observation of a program derived from its operational semantics, one can also define a notion of *observational equivalence* $M \approx_\sigma N$ between programs of the same type σ , yielding a criterion for when two programs should be considered equal. Observational equivalence is generally robust against variations in the notion of observation. Operational adequacy generally yields the implication:

$$\models M = N : \sigma \implies M \approx_\sigma N$$

and the converse is, of course, *full abstraction*.

In the case of our language of choice and rewards, as observations we take boolean returns:

$$\text{Ob}(M) = \pi_2(\mathcal{S}[\![M]\!](0)) (= \pi_2(\mathcal{S}[\![\text{Op}_s(M)]\!])) \quad (M : \text{Bool})$$

where the operational definition in brackets is equivalent because of adequacy. Thus, our observations ignore rewards. We then define observational equivalence relative to boolean contexts:

$$M \approx_\sigma N \iff \forall C[] : \sigma \rightarrow \text{Bool}. \text{Ob}(C[M]) = \text{Ob}(C[N])$$

where M and N are closed terms of the same type σ , and $C[M] : \text{Bool}$ and $C[N] : \text{Bool}$.

Because the denotational semantics is compositional, it facilitates proofs of program equivalences, including ones that justify program transformations, and more broadly can be convenient for certain arguments about programs. For this purpose, we rely on the equivalence relation $\Gamma \vdash_{\text{Ax}} M = N : \sigma$ described in Section 3.5. As remarked there, our general semantics is equationally consistent (assuming the axioms are satisfied). We interest ourselves in a limited converse, where σ is a basic type and M and N are programs; we call this *program completeness for basic types*.

Turning again to our language of choices and rewards, our system of axioms, Ax , is given in Figure 3. As is straightforwardly verified, the algebraic operations obey some natural equations: the choice operation is associative and idempotent (but it is not commutative); further, the reward operation respects the monoid operations and commutes with the choice operation. This justifies the first five of our axioms. One can also show the following, perhaps less expected, equalities hold for $r, s \in \mathbb{R}$ and $F, G \in \mathcal{S}(X)$, for any set X :

$$r \cdot F \text{ or } s \cdot F = t \cdot F \quad (t = \max(r, s)) \tag{1}$$

$$(r \cdot F \text{ or } G) \text{ or } s \cdot F = r \cdot F \text{ or } G \quad (r \geq s) \tag{2}$$

$$(r \cdot F \text{ or } G) \text{ or } s \cdot F = G \text{ or } s \cdot F \quad (r < s) \tag{3}$$

and this justifies our last two axioms.

Some useful consequences of these equations, mirroring the equalities 1–3, are:

$$c \cdot M \text{ or } c' \cdot M = c'' \cdot M \quad (\text{where } \mathcal{S}[\![c'']\!] = \max(\mathcal{S}[\![c]\!], \mathcal{S}[\![c']\!])) \tag{R_1}$$

$$\begin{aligned}
(L \text{ or } M) \text{ or } N &= L \text{ or } (M \text{ or } N) & M \text{ or } M &= M \\
0 \cdot N &= N & x \cdot (y \cdot N) &= (x + y) \cdot N \\
x \cdot (M \text{ or } N) &= (x \cdot M) \text{ or } (x \cdot N) \\
\text{if } x \geq y \text{ then } x \cdot M \text{ else } y \cdot M &= x \cdot M \text{ or } y \cdot M \\
\text{if } x \geq z \text{ then } (x \cdot M \text{ or } y \cdot N) \text{ else } (y \cdot N \text{ or } z \cdot M) &= (x \cdot M \text{ or } y \cdot N) \text{ or } z \cdot M
\end{aligned}$$

Fig. 3. Equations for choices and rewards

$$(c \cdot L \text{ or } c' \cdot M) \text{ or } c'' \cdot N = (c \cdot L \text{ or } c' \cdot M) \quad (\text{if } \llbracket c \rrbracket \geq \llbracket c'' \rrbracket) \quad (\text{R}_2)$$

$$(c \cdot L \text{ or } c' \cdot M) \text{ or } c'' \cdot N = (c' \cdot M \text{ or } c'' \cdot N) \quad (\text{if } \llbracket c \rrbracket < \llbracket c'' \rrbracket) \quad (\text{R}_3)$$

Our equational system allows us to put programs of basic type into a normal form. We say that such a term is in *normal form* if (ignoring bracketing of or) it is an effect value of the form

$$(c_1 \cdot d_1) \text{ or } \dots \text{ or } (c_n \cdot d_n)$$

with $n > 0$ and no d_i occurring twice.

LEMMA 8. *Every program M of basic type is provably equal to a normal form $\text{NF}(M)$.*

PROOF. By the ordinary adequacy theorem (Theorem 1), M can be proved equal to an effect value E . Using the associativity equations and the fact that reward and or commute, E can be proved equal to a term of the form $c_1 \cdot d_1 \text{ or } \dots \text{ or } c_n \cdot d_n$, possibly with some d 's occurring more than once. Such duplications can be removed using equations R_1 , R_2 , and R_3 . \square

The next theorem shows that, under reasonable assumptions, four equivalence relations coincide, and thereby simultaneously establishes: a normal form for provable equality; completeness of our proof system for equations between programs; and full abstraction for programs of basic type.

We say that R is *expressively unbounded* (w.r.t. the constants of type Rew) if, for every $r \in \text{R}$, there is a $c : \text{Rew}$ such that $r < \llbracket c \rrbracket$; and we say that R is *expressively dense* (w.r.t. the constants of type Rew) if, for all $r, s \in \text{R}$ with $r < s$, there is a $c : \text{Rew}$ such that $r < \llbracket c \rrbracket < s$.

THEOREM 4. *Suppose that R is expressively unbounded and dense. Then, for any basic type b , and any two programs $M : b$ and $N : b$, the following equivalences hold:*

$$\text{NF}(M) = \text{NF}(N) \iff \vdash_{\text{Ax}} M = N : b \iff \models M = N : b \iff M \approx_b N$$

PROOF. We already know the implications from left-to-right hold (using Lemma 8). So it suffices to show that:

$$\text{NF}(M) \neq \text{NF}(N) \implies M \not\approx_b N$$

Suppose that $\text{NF}(M)$ and $\text{NF}(N)$ are distinct normal forms, say

$$(c_1 \cdot d_1) \text{ or } \dots \text{ or } (c_n \cdot d_n) \quad \text{and} \quad (c'_1 \cdot d'_1) \text{ or } \dots \text{ or } (c'_{n'} \cdot d'_{n'})$$

As R is expressively unbounded we can choose c with $\llbracket c \rrbracket > \text{any } \llbracket c_i \rrbracket \text{ or } \llbracket c'_j \rrbracket$.

Suppose, first, that some d_i is no d'_j . Consider the context

$$C_1[-] =_{\text{def}} \text{if } [-] = d_i \text{ then } \llbracket c \rrbracket \cdot \text{tt} \text{ else } \text{ff}$$

As no d'_j is d_i , $\text{Ob}(C_1[N]) = \text{ff}$. As $\llbracket c_i \rrbracket + \llbracket c \rrbracket > \text{any } \llbracket c_j \rrbracket$ with $j \neq i$, we have $\text{Ob}(C_1[M]) = \text{tt}$; hence, in this case, $M \not\approx N$, as required.

So we may assume that every d_i is some d'_j (and vice versa). Suppose next that some corresponding c_i and c'_j differ. As R is linearly ordered we may assume, w.l.o.g. that $\llbracket c_i \rrbracket > \llbracket c'_j \rrbracket$. As R is expressively dense, we can choose \underline{c} with $\llbracket c_i \rrbracket > \llbracket \underline{c} \rrbracket > \llbracket c_j \rrbracket$. Then the context

$$C_2[-] =_{\text{def}} (c + \underline{c}) \cdot \text{ff} \text{ or } C_1[-]$$

distinguishes M and N , with $\text{Ob}(M) = \text{tt}$ and $\text{Ob}(N) = \text{ff}$.

So we now have that $n = n'$ and that $(c_1 \cdot d_1) \dots (c_n \cdot d_n)$ and $(c'_1 \cdot d'_1) \dots (c'_n \cdot d'_n)$ are distinct and are permutations of each other. They therefore have the forms

$$(c_1 \cdot d_1) \text{ or } \dots \text{ or } (c_{i-1} \cdot d_{i-1}) \text{ or } (c_i \cdot d_i) \text{ or } (c_{i+1} \cdot d_{i+1}) \text{ or } \dots \text{ or } (c_n \cdot d_n)$$

and

$$(c_1 \cdot d_1) \text{ or } \dots \text{ or } (c_{i-1} \cdot d_{i-1}) \text{ or } (c'_i \cdot d'_i) \text{ or } (c'_{i+1} \cdot d'_{i+1}) \text{ or } \dots \text{ or } (c'_n \cdot d'_{n-1})$$

respectively where d_i and d'_i are different. Then the context:

$$\begin{aligned} & \text{let } x : b \text{ be } [-] \text{ in} \\ C_3[-] =_{\text{def}} & \begin{cases} \text{if } x = d_i \text{ then } c \cdot \text{tt} \text{ else} \\ \text{if } x = d'_i \text{ then } c \cdot \text{ff} \text{ else ff} \end{cases} \end{aligned}$$

distinguishes M and N . Thus, in all cases we have $M \not\approx_b N$, concluding the proof. \square

Full abstraction then follows for all first-order terms, as usual. It would be interesting to know if it fails at second order.

5 ADDING PROBABILITIES

We next extend the language of choices and rewards by probabilistic nondeterminism. Thus, we have the three main ingredients of MDPs, though in the setting of a higher-order λ -calculus rather than the more usual state machines. We proceed as in the previous section, often reusing notation.

5.1 Syntax

We add function symbols $\text{con}_p : \text{RewRew} \rightarrow \text{Rew}$ for $p \in [0, 1]$, and algebraic operation symbols $+_p : (\varepsilon, 2)$, and use infix notation for both. The former is intended to represent a convex combination of rewards; the latter is for binary probabilistic choice.

For example (continuing an example from Section 4.1), we may write the tiny program:

$$(5 \cdot \text{tt}) \text{ or } ((5 \cdot \text{tt}) +_{.5} (6 \cdot \text{ff}))$$

Intuitively, like the program of Section 4.1, this program could return either tt or ff , with respective rewards 5 and 6. Both outcomes are possible on the right branch of its choice, each with probability .5. The intended semantics aims to maximize expected rewards, so that branch is selected.

This example illustrates how the language can express MDP-like transitions. In MDPs, at each time step, the decision-maker chooses an action, and the process randomly moves to a new state and yields rewards; the distribution over the new states depends on the current state and the action. In our language, all decisions are binary, but bigger decisions can be programmed from them. Moreover, the decisions are separate from the probabilistic choices and the rewards, but as in this example it is a simple matter of programming to combine them. A more complete encoding of MDPs can be done by adding primitive recursion to the language, as suggested in the Introduction.

5.2 Rewards and additional effects

We further assume that R is equipped with a *convex algebra* (a.k.a. a *barycentric algebra*) structure compatible with the order and monoid structure. That is: there are binary probabilistic choice functions $+_p : R^2 \rightarrow R$ ($p \in [0, 1]$) such that the following four equations hold:

$$\begin{aligned} x +_1 y &= x \\ x +_p x &= x \\ x +_p y &= y +_{1-p} x \\ (x +_p y) +_q z &= x +_{pq} (y +_{\frac{r-pq}{1-pq}} z) \quad (p, q < 1) \end{aligned}$$

and probabilistic choice respects and preserves the order in its first argument (and so, too, in its second) in the sense that

$$r \leq r' \iff r +_p s \leq r' +_p s \quad (p \neq 0)$$

and probabilistic choice is homogeneous, in the sense that

$$r + (s +_p s') = (r + s) +_p (r + s')$$

Continuing the examples of Section 4.2, probabilistic choice can be defined using the usual convex combination: $r +_p s = pr + (1-p)s$ of real numbers. Convex algebras provide a suitable algebraic structure for probability. They seem to have been first introduced in [Stone 1949], and have an extensive history, briefly surveyed in [Keimel and Plotkin 2017]. They are equivalent to *convex spaces* which are algebras equipped with operations $\sum_{i=1}^n p_i x_i$ (where $p_i \in [0, 1]$ and $\sum_{i=1}^n p_i = 1$), subject to natural axioms [Pumplün and Röhrh 1995]; we will use the two notations interchangeably.

We take T to be the combination $\mathcal{P}_f(R \times -)$ of the finite probability distribution monad with the writer monad. This monad has unit $(\eta_T)_X(x) = \langle 0, x \rangle$, and the Kleisli extension of a map $f : X \rightarrow T(X)$ is $f^{\dagger T}(\sum p_i \langle r_i, x_i \rangle) = \sum p_i (r_i \cdot f(x_i))$. The algebra map $\alpha : T(R) \rightarrow R$ is given using the convex algebra and monoid structure on R : $\alpha(\sum p_i \langle r_i, s_i \rangle) = \sum p_i (r_i + s_i)$. For rewards and probabilistic choices, we define the generic effects $g_{\text{reward}} : R \rightarrow T(\mathbb{1})$ and $g_{+_p} \in T(\mathbb{B})$ to be $\lambda r : R. \langle r, * \rangle$ and $p \langle 0, 0 \rangle + (1-p) \langle 0, 1 \rangle$. The corresponding algebraic operations on T are:

$$(\text{reward}_T)_X(r, \sum p_i(r_i, x_i)) = \sum p_i(r + r_i, x_i) \quad (+_{pT})_X(\mu, \nu) = p\mu + (1-p)\nu$$

We generally write $(\text{reward}_T)_X$ using an infix operator \cdot_X , as in Section 4.2.

5.3 Operational semantics

We again take a game-theoretic point of view, with Player now playing a game against Nature, assumed to make probabilistic choices. Player therefore seeks to optimize their expected rewards. Effect values $E : \sigma$ are regarded as games but with one additional clause:

- if $E = E_0 +_p E_1$, it is Nature's turn to move. Nature picks E_0 with probability p , and E_1 with probability $1-p$.

To account for probabilistic choice we add a rule to the definition of strategies:

$$\frac{s_0 : E_0 \quad s_1 : E_1}{(s_0, s_1) : E_0 +_p E_1}$$

and a case to the definition of the linear orders on strategies:

- Game is $E_0 +_p E_1$:

$$(s_0, s_1) \leq_{E_0 +_p E_1} (s'_0, s'_1) \iff s_0 <_{E_0} s'_0 \quad \vee \quad (s_0 = s'_0 \wedge s_1 \leq_{E_1} s'_1)$$

For any effect value $E : \sigma$, the outcome $\text{Out}(s, E)$ of a strategy $s : E$ is a finite probability distribution over $R \times \text{Val}_\sigma$, i.e., an element of $\mathcal{P}_f(R \times \text{Val}_\sigma)$. It is defined by:

$$\begin{aligned} \text{Out}(*, V) &= \delta_{\langle 0, V \rangle} \\ \text{Out}(0s, E_0 \text{ or } E_1) &= \text{Out}(s, E_0) \\ \text{Out}(1s, E_0 \text{ or } E_1) &= \text{Out}(s, E_1) \\ \text{Out}(s, c \cdot E) &= \llbracket c \rrbracket \cdot_{\text{Val}_\sigma} \text{Out}(s, E) \\ \text{Out}((s_0, s_1), E_0 +_p E_1) &= p \text{Out}(s_0, E_0) + (1 - p) \text{Out}(s_1, E_1) \end{aligned}$$

We used the Dirac distribution δ_x here; below, as is common, we just write x .

The expected value of a finite probability distribution on $R \times X$, for a set X , is:

$$E(\sum p_i(r_i, m_i)) = \sum p_i r_i$$

(Note that $E : \mathcal{P}_f(R \times X) \rightarrow R$ can be written as $\alpha \circ T(0)$, just as $\pi_1 : R \times X \rightarrow R$ in the previous section could be.) The expected reward of a strategy is then:

$$\text{Rew}(s, E) = E(\text{Out}(s, E))$$

Our selection operational semantics, $\text{Op}_s(M) \in R \times \text{Val}_\sigma$ for $M : \sigma$, is defined as before by:

$$\text{Op}_s(M) = \text{Out}(\text{argmax } s : \text{Op}(M), \text{Rew}(s, \text{Op}(M)), \text{Op}(M))$$

where we are now, as anticipated, focusing on expected rewards.

Much as in Section 4, we develop a local characterization of the globally optimizing selection operational semantics. We give this characterization in Theorem 5, below; it is analogous to Theorem 2 in Section 4. Some auxiliary lemmas are required. The first of them is another argmax lemma that deals with strategies for probabilistic choice:

LEMMA 9. (Second argmax lemma) Let P and Q be finite linear orders, suppose $f : P \times Q \rightarrow R$, and let $P \times Q$ be given the lexicographic ordering. Define $g : P \rightarrow R$, $\underline{u} \in P$ and $\underline{v} \in Q$ by:

$$\begin{aligned} g(u) &= \text{argmax } v : Q. f(u, v) \\ \underline{u} &= \text{argmax } u : P. f(u, g(u)) \\ \underline{v} &= g(\underline{u}) \end{aligned}$$

Then:

$$(\underline{u}, \underline{v}) = \text{argmax } (u, v) : P \times Q. f(u, v)$$

PROOF. Consider any pair (u_0, v_0) . By the definition of g we have $g(u_0) \leq u_1$ in the sense that:

$$(f(u_0, g(u_0)) > f(u_0, v_0) \vee (f(u_0, g(u_0)) = f(u_0, v_0) \wedge g(u_0) \leq v_0)$$

and it follows that $(u_0, g(u_0)) \leq_f (u_0, v_0)$.

Next, by the definition of \underline{u} we have $\underline{u} \leq u_0$ in the sense that:

$$(f(\underline{u}, g(\underline{u})) > f(u_0, g(u_0)) \vee (f(\underline{u}, g(\underline{u})) = f(u_0, g(u_0)) \wedge \underline{u} \leq u_0)$$

and it follows that $(\underline{u}, g(\underline{u})) \leq_f (u_0, g(u_0))$. (The only non-obvious point may be that in the case where $f(\underline{u}, g(\underline{u})) = f(u_0, g(u_0))$, we have $\underline{u} \leq u_0$, so either $\underline{u} < u_0$, when $(\underline{u}, g(\underline{u})) <_f (u_0, g(u_0))$ or else $\underline{u} = u_0$, when $(\underline{u}, g(\underline{u})) = (u_0, g(u_0))$.)

So, as $\underline{v} = g(\underline{u})$, we have

$$(\underline{u}, \underline{v}) = (\underline{u}, g(\underline{u})) \leq_f (u_0, g(u_0)) \leq_f (u_0, v_0)$$

thereby establishing the required minimality of $(\underline{u}, \underline{v})$. \square

The next two lemmas concern expectations for distributions constructed by the \cdot operation and by convex combinations.

LEMMA 10.

$$\mathbf{E}(r \cdot \mu) = r + \mathbf{E}(\mu) \quad \text{Rew}(s, c \cdot E) = \llbracket c \rrbracket + \text{Rew}(s, E)$$

LEMMA 11.

$$\begin{aligned} \mathbf{E}(p\mu + (1-p)v) &= p\mathbf{E}(\mu) + (1-p)\mathbf{E}(v) \\ \text{Rew}((s_0, s_1), E_0 +_p E_1) &= p\text{Rew}(s_0, E_0) + (1-p)\text{Rew}(s_1, E_1) \end{aligned}$$

THEOREM 5. *The following hold for well-typed effect values:*

- (1) $\text{Op}_s(V) = \langle 0, V \rangle (= \eta_T(V))$
- (2) $\text{Op}_s(E_0 \text{ or } E_1) = \text{Op}_s(E_0) \max_E \text{Op}_s(E_1)$
- (3) $\text{Op}_s(c \cdot E) = \llbracket c \rrbracket \cdot \text{Op}_s(E)$
- (4) $\text{Op}_s(E_0 +_p E_1) = p\text{Op}_s(E_0) + (1-p)\text{Op}_s(E_1)$

PROOF. We just consider the fourth case. We have:

$$\text{Op}_s(E_0 +_p E_1) = \text{Out}(\text{argmax}(s_0, s_1) : E_0 +_p E_1. \text{Rew}((s_0, s_1), E_0 +_p E_1), E_0 +_p E_1)$$

So, following the second argmax lemma (Lemma 9), we first consider the function

$$f(s_0, s_1) \stackrel{\text{def}}{=} \text{Rew}((s_0, s_1), E_0 +_p E_1) = p\text{Rew}(s_0, E_0) + (1-p)\text{Rew}(s_1, E_1)$$

where the second equality holds by Lemma 11. We next consider the function:

$$\begin{aligned} g(s_0) &\stackrel{\text{def}}{=} \text{argmax } s_1 : E_1. f(s_0, s_1) \\ &= \text{argmax } s_1 : E_1. p\text{Rew}(s_0, E_0) + (1-p)\text{Rew}(s_1, E_1) \\ &= \text{argmax } s_1 : E_1. \text{Rew}(s_1, E_1) \end{aligned}$$

where the second equality holds as convex combinations are order-preserving and reflecting in their second argument. Finally we consider

$$\begin{aligned} \underline{s}_0 &\stackrel{\text{def}}{=} \text{argmax } s_0 : E_0. f(s_0, g(s_0)) \\ &= \text{argmax } s_0 : E_0. p\text{Rew}(s_0, E_0) + (1-p)\text{Rew}(g(s_0), E_1) \\ &= \text{argmax } s_0 : E_0. \text{Rew}(s_0, E_0) \end{aligned}$$

where the second equality holds as convex combinations are order-preserving and reflecting in their first argument, and as $g(s_0)$ is independent of s_0 .

So setting

$$\underline{s}_1 = g(\underline{s}_0) = \text{argmax } s_1 : E_1. \text{Rew}(s_1, E_1)$$

by the second argmax lemma (Lemma 9) we have:

$$(\underline{s}_0, \underline{s}_1) = \text{argmax}(s_0, s_1) : E_0 +_p E_1. \text{Rew}((s_0, s_1), E_0 +_p E_1)$$

so we finally have:

$$\begin{aligned} \text{Op}_s(E_0 +_p E_1) &= \text{Out}((\underline{s}_0, \underline{s}_1), E_0 +_p E_1) \\ &= p\text{Out}(\underline{s}_0, E_0) + (1-p)\text{Out}(\underline{s}_1, E_1) \\ &= p\text{Out}(\text{argmax } s_0 : E_0. \text{Rew}(s_0, E_0), E_0) \\ &\quad + (1-p)\text{Out}(\text{argmax } s_1 : E_1. \text{Rew}(s_1, E_1), E_1) \\ &= p\text{Op}_s(E_0) + (1-p)\text{Op}_s(E_1) \end{aligned}$$

as required. \square

5.4 Denotational semantics

For the corresponding denotational semantics, we take T to be the combination $\mathcal{P}_f(R \times -)$ of the finite probability distribution monad with the writer monad, as discussed above, and we have an algebra map $\alpha : T(R) \rightarrow R$. So we have the anticipated strong monad

$$S(X) = (X \rightarrow R) \rightarrow \mathcal{P}_f(R \times X)$$

Extending the semantics of the language of the previous section, the function symbols con_p denote the convex combination operations on R . As regards the algebraic operation symbols, for or we use the algebraic operation family or_X given in Section 2, and for reward and $+$ we take the algebraic operation families $(\text{reward}_S)_X$ and $(+_p)_X$ induced by the $(\text{reward}_T)_X$ and $(+_p)_X$, so:

$$(\text{reward}_S)_X(r, G)(\gamma) = (\text{reward}_T)_X(r, G\gamma) \quad \text{and} \quad (+_p)_X(F, G)(\gamma) = (+_p)_X(F(\gamma), G(\gamma))$$

5.5 Adequacy

As in Section 4.5, we aim to prove a selection adequacy theorem connecting the globally defined selection operational semantics with the denotational semantics. We again need some notation: for $\mu = \sum p_i \langle r_i, V_i \rangle \in \mathcal{P}_f(R \times \text{Val}_\sigma)$ we write $\llbracket \mu \rrbracket$ for $\sum p_i \langle r_i, \llbracket V_i \rrbracket \rangle \in \mathcal{P}_f(R \times \llbracket \sigma \rrbracket)$. Note that, as before, we have $\llbracket \mu \rrbracket = T(\llbracket - \rrbracket_\sigma)(\mu)$ and that $E : \mathcal{P}_f(R \times X) \rightarrow R$ can be written as $\alpha \circ T(0)$.

LEMMA 12. *For any effect value $E : \sigma$ we have:*

$$S\llbracket E \rrbracket(0) = \llbracket \text{Op}_s(E) \rrbracket$$

PROOF. We use structural induction, considering only the case where $E = E_0 +_p E_1$. We have:

$$\begin{aligned} \llbracket \text{Op}_s(E_0 +_p E_1) \rrbracket &= T(\llbracket - \rrbracket_\sigma)(\text{Op}_s(E_0) (+_p)_T \text{Op}_s(E_1)) \quad (\text{by Theorem 5, part (4)}) \\ &= \llbracket \text{Op}_s(E_0) \rrbracket (+_p)_T \llbracket \text{Op}_s(E_1) \rrbracket \quad (\text{as } T(\llbracket - \rrbracket_\sigma) \text{ is a homomorphism}) \\ &= S\llbracket E_0 \rrbracket(0) (+_p)_T S\llbracket E_1 \rrbracket(0) \quad (\text{by induction hypothesis}) \\ &= (S\llbracket E_0 \rrbracket (+_p)_S S\llbracket E_1 \rrbracket(0)) \\ &= S\llbracket E_0 +_p E_1 \rrbracket(0) \end{aligned}$$

□

Selection adequacy for our language with probabilities then follows:

THEOREM 6 (SELECTION ADEQUACY). *For any program $M : \sigma$ we have:*

$$S\llbracket M \rrbracket(0) = \llbracket \text{Op}_s(M) \rrbracket$$

5.6 Program equivalences

We can again define a notion of observational equivalence. Define $\text{ob} : T(X) \rightarrow \mathcal{P}_f(X)$ by

$$\text{ob}(\sum_i p_i \langle r_i, x_i \rangle) = \sum_i p_i x_i$$

and as observations we take probabilistic boolean returns:

$$\text{Ob}(M) = \text{ob}(S\llbracket M \rrbracket(0)) (= \text{ob}(\llbracket \text{Op}_s(M) \rrbracket)) \quad (M : \text{Bool})$$

yielding a notion of observational equivalence $M \approx_\sigma N$, as before.

Turning to equations, our axioms for the language of choices and rewards (see Figure 3) remain true; the axioms for convex algebras yield corresponding equations, for example:

$$M +_p N = N +_{1-p} M$$

and some others are given in Figure 4.

As always, semantic equality implies observational equivalence. However, we conjecture that our semantics is not fully abstract. For example, the programs $1 \cdot \text{tt} +_{1/2} 3 \cdot \text{tt}$ and $2 \cdot \text{tt}$ are

$$\begin{aligned}
x \cdot (M +_p N) &= x \cdot M +_p x \cdot N \\
L +_p (M \text{ or } N) &= (L +_p M) \text{ or } (L +_p N) \\
(L \text{ or } M) +_p N &= (L +_p N) \text{ or } (M +_p N)
\end{aligned}$$

Fig. 4. Some equations for choices, rewards, and probabilistic choices

semantically distinct but, we believe, observationally equivalent. Indeed we conjecture that the following observational equivalence holds for programs $M : \sigma$:

$$x \cdot M +_p y \cdot M \approx_\sigma (x \text{ con}_p y) \cdot M \quad (\text{C})$$

Thus it seems we cannot observationally distinguish between different distributions over the same value, but which have the same expected reward for that value.

Our notion of observational equivalence does at least determine one reward statistic, the expected value of a program:

FACT 1. *Suppose R is expressively dense. Then, for any programs $M, N : \sigma$ we have:*

$$M \approx_\sigma N \implies \mathbf{E}(\text{Op}_s(M)) = \mathbf{E}(\text{Op}_s(N))$$

PROOF. Since $M \approx_\sigma N$, they return the same probability distribution $\sum_{i=1}^n p_i V_i$ on values. Suppose, for the sake of contradiction, that, for example, $\mathbf{E}(\text{Op}_s(M)) < \mathbf{E}(\text{Op}_s(N))$. Choose c so that:

$$\mathbf{E}(\text{Op}_s(M)) < \llbracket c \rrbracket < \mathbf{E}(\text{Op}_s(N))$$

Define a boolean context $C[-]$ by:

$$C[-] = \text{let } x : \sigma \text{ be } [-] \text{ in } (x = V_1 \vee \dots \vee x = V_n)$$

Then $\mathbf{E}(\text{Op}_s(L)) = \mathbf{E}(\text{Op}_s(C[L]))$, for any $L : \sigma$, and the following context distinguishes M and N :

$$(c \cdot \text{ff}) \text{ or } C[-]$$

□

As our notion of observational equivalence seems not unreasonable, it would be interesting to find a denotational semantics that is fully abstract at basic types for it. Perhaps, too, a complete axiom system for such semantics incorporating the equivalence (C) could be defined for equalities between programs of basic type.

6 CONCLUSION

This paper studies decision-making abstractions in the context of simple higher-order programming languages, focusing on their semantics, treating them operationally and denotationally. The denotational semantics are compositional. They are based on the selection monad, which has rich connections with logic and game theory. Unlike other programming-language research on games and semantics (for example, see [Abramsky et al. 2000; Hyland and Ong 2000]), the treatment of games in this paper is extensional, focusing on choices but ignoring other aspects of computation, such as function calls and returns. Moreover, the games are one-player games. Going further, we have started to explore extensions of our languages with multiple players, where each choice and each reward is associated with one player. For example, writing A and E for the players, we can

program a version of the classic prisoners’s dilemma:

```
let silentA, silentE : Bool be (tt orA ff), (tt orE ff) in
if silentA and silentE then - 1 ·A - 1 ·E *
else if silentA then - 3 ·A *
else if silentE then - 3 ·E *
else - 2 ·A - 2 ·E *
```

Here, silent_A and silent_E indicate whether the players remain silent, and the rewards, which are negative, correspond to years of prison. Many of our techniques carry over to languages with multiple players, which give rise to interesting semantic questions (e.g., should we favor some players over others? require Nash equilibria?) and may also be useful in practice.

In describing Software 2.0, Karpathy suggested specifying some goal on the behavior of a desirable program, writing a “rough skeleton” of the code, and using the computational resources at our disposal to search for a program that works [Karpathy 2017]. While this vision may be attractive, realizing it requires developing not only search techniques but also the linguistic constructs to express goals and code skeletons. In the variant of this vision embodied in SmartChoices, the skeleton is actually a complete program, albeit in an extended language with decision-making abstractions. Thus, in the brave new world of Software 2.0 and its relatives, programming languages still have an important role to play, and their study should be part of their development. Our paper aims to contribute to one aspect of this project; much work remains.

In comparison with recent theoretical work on languages with differentiation (for example, see [Abadi and Plotkin 2020; Barthe et al. 2020; Brunel et al. 2020; Cruttwell et al. 2019; Fong et al. 2019; Huot et al. 2020]), our languages are higher-level: they focus on how optimization or machine learning may be made available to a programmer rather than on how they would be implemented. However, a convergence of these research lines is possible, and perhaps desirable. One thought is to extend our languages with differentiation primitives to construct selection functions that use gradient descent. These would be alternatives to argmax as discussed in the Introduction. Monadic reflection and reification, in the sense of Filinski [Filinski 1994], could support the use of such alternatives, and more generally enhance programming flexibility. Similarly, it would be attractive to deepen the connections between our languages and probabilistic ones (for example, see [Goodman et al. 2012]). It may also be interesting to connect our semantics with particular techniques from the literature on MDPs and RL, and further to explore whether monadic ideas can contribute to implementations that include such techniques. Finally, at the type level, the monadic approach distinguishes “selected” values and “ordinary” ones; the “selected” values are reminiscent of the “uncertain” values of $\text{Uncertain} < T >$ [Bornholt et al. 2014], and the distinction may be useful as in that setting.

ACKNOWLEDGEMENTS

We are grateful to Craig Boutilier, Eugene Brevdo, Daniel Golovin, Michael Isard, Eugene Kirpichov, Ohad Kammar, Matt Johnson, Dougal Maclaurin, Martin Mladenov, Adam Paszke, Dimitrios Vytiniotis, and Jay Yagnik for discussions of this and related work.

REFERENCES

- Martín Abadi and Gordon D. Plotkin. 2020. A simple differentiable programming language. *Proc. ACM Program. Lang.* 4, POPL (2020), 38:1–38:28. <https://doi.org/10.1145/3371106>
- Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for PCF. *Inf. Comput.* 163, 2 (2000), 409–470. <https://doi.org/10.1006/inco.2000.2930>
- Gilles Barthe, Raphaëlle Crubillé, Ugo Dal Lago, and Francesco Gavazzo. 2020. On the Versatility of Open Logical Relations - Continuity, Automatic Differentiation, and a Containment Theorem. In *Programming Languages and Systems - 29th*

- European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Peter Müller (Ed.), Vol. 12075. Springer, 56–83. https://doi.org/10.1007/978-3-030-44914-8_3
- Joe Bolt, Jules Hedges, and Philipp Zahn. 2018. Sequential games and nondeterministic selection functions. *CoRR* abs/1811.06810 (2018). arXiv:1811.06810 <http://arxiv.org/abs/1811.06810>
- James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. 2014. Uncertain<T>: a first-order type for uncertain data. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14*, Rajeev Balasubramanian, Al Davis, and Sarita V. Adve (Eds.). ACM, 51–66. <https://doi.org/10.1145/2541940.2541958>
- C. Bouillier, R. Reiter, M. Soutchanski, and S. Thrun. 2000. Decision-Theoretic, High-level Robot Programming in the Situation Calculus. In *Proceedings of the AAAI National Conference on Artificial Intelligence*. AAAI.
- Alois Brunel, Damiano Mazza, and Michele Pagani. 2020. Backpropagation in the simply typed lambda-calculus with linear negation. *Proc. ACM Program. Lang.* 4, POPL (2020), 64:1–64:27. <https://doi.org/10.1145/3371132>
- David Budden, Matteo Hessel, John Quan, and Steven Kapturowski. 2020. RLax: Reinforcement Learning in JAX. <http://github.com/deepmind/rlax>
- Vladimir Bychkovsky. 2018. Spiral: Self-tuning services via real-time machine learning. Blog post at <https://engineering.fb.com/data-infrastructure/spiral-self-tuning-services-via-real-time-machine-learning/>.
- Victor Carbune, Thierry Coppey, Alexander N. Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. 2018. Predicted Variables in Programming. *CoRR* abs/1810.00619 (2018). arXiv:1810.00619 <http://arxiv.org/abs/1810.00619>
- Kai-Wei Chang, He He, Stéphane Ross, Hal Daumé III, and John Langford. 2016. A Credit Assignment Compiler for Joint Prediction. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 1705–1713. <http://papers.nips.cc/paper/6256-a-credit-assignment-compiler-for-joint-prediction>
- Geoff Cruttwell, Jonathan Gallagher, and Ben MacAdam. 2019. Towards formalizing and extending differential programming using tangent categories. *Proc. ACT* (2019).
- Martín Escardó. 2015. Constructive decidability of classical continuity. *Math. Struct. Comput. Sci.* 25, 7 (2015), 1578–1589. <https://doi.org/10.1017/S096012951300042X>
- Martín Escardó and Paulo Oliva. 2011. Sequential games and optimal strategies. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 467, 2130 (2011), 1519–1545.
- Martín Escardó and Paulo Oliva. 2017. The Herbrand Functional Interpretation of the double Negation Shift. *J. Symb. Log.* 82, 2 (2017), 590–607. <https://doi.org/10.1017/jsl.2017.8>
- Martín Hötzel Escardó and Paulo Oliva. 2010. Selection functions, bar recursion and backward induction. *Math. Struct. Comput. Sci.* 20, 2 (2010), 127–168. <https://doi.org/10.1017/S0960129509990351>
- Martín Hötzel Escardó and Paulo Oliva. 2012. The Peirce translation. *Ann. Pure Appl. Log.* 163, 6 (2012), 681–692. <https://doi.org/10.1016/j.apal.2011.11.002>
- Martín Hötzel Escardó, Paulo Oliva, and Thomas Powell. 2011. System T and the Product of Selection Functions. In *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings (LIPIcs)*, Marc Bezem (Ed.), Vol. 12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 233–247. <https://doi.org/10.4230/LIPIcs.CSL.2011.233>
- Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ -calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 193–222.
- Andrzej Filinski. 1994. Representing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. Association for Computing Machinery, 446–457. <https://doi.org/10.1145/174675.178047>
- Brendan Fong, David I. Spivak, and Rémy Tuyéras. 2019. Backprop as Functor: A compositional perspective on supervised learning. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 1–13. <https://doi.org/10.1109/LICS.2019.8785665>
- Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2012. Church: a language for generative models. *CoRR* abs/1206.3255 (2012). arXiv:1206.3255 <http://arxiv.org/abs/1206.3255>
- Jules Hedges. 2015. The selection monad as a CPS transformation. *CoRR* abs/1503.06061 (2015). arXiv:1503.06061 <http://arxiv.org/abs/1503.06061>
- Mathieu Huot, Sam Staton, and Matthijs Vákár. 2020. Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Jean Goubault-Larrecq and Barbara König (Eds.), Vol. 12077. Springer, 319–338. https://doi.org/10.1007/978-3-030-45231-5_17

- J. M. E. Hyland and C.-H. Luke Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408. <https://doi.org/10.1006/inco.2000.2917>
- Andrej Karpathy. 2017. Software 2.0. <https://medium.com/@karpathy/software-2-0-a64152b37c35>.
- Klaus Keimel and Gordon D. Plotkin. 2017. Mixed powerdomains for probability and nondeterminism. *Log. Methods Comput. Sci.* 13, 1 (2017). [https://doi.org/10.23638/LMCS-13\(1:2\)2017](https://doi.org/10.23638/LMCS-13(1:2)2017)
- Anders Kock. 1972. Strong functors and monoidal monads. *Archiv der Mathematik* 23, 1 (1972), 113–120.
- Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210. [https://doi.org/10.1016/S0890-5401\(03\)00088-9](https://doi.org/10.1016/S0890-5401(03)00088-9)
- John McCarthy. 1963. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 35. Elsevier, 33 – 70. [https://doi.org/10.1016/S0049-237X\(08\)72018-4](https://doi.org/10.1016/S0049-237X(08)72018-4)
- D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. 1998. *PDDL - The Planning Domain Definition Language*. Technical Report TR-98-003. Yale Center for Computational Vision and Control,.
- Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89)*, Pacific Grove, California, USA, June 5-8, 1989. IEEE Computer Society, 14–23. <https://doi.org/10.1109/LICS.1989.39155>
- Gordon Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structures*, Furio Honsell and Marino Miculan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–24.
- Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. <https://doi.org/10.1023/A:1023064908962>
- Dieter Pumplün and Helmut Röhr. 1995. Convexity theories IV. Klein-Hilbert parts in convex modules. *Applied Categorical Structures* 3, 2 (1995), 173–200.
- Scott Sanner. 2011. Relational dynamic influence diagram language (rddl): Language description. Official language of the uncertainty track of the Seventh International Planning Competition.
- Marshall Harvey Stone. 1949. Postulates for the barycentric calculus. *Annali di Matematica Pura ed Applicata* 29, 1 (1949), 25–30.
- Tim Vieira, Matthew Francis-Landau, Nathaniel Wesley Filardo, Farzad Khorasani, and Jason Eisner. 2017. Dyna: Toward a Self-Optimizing Declarative Language for Machine Learning Applications. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. Association for Computing Machinery, 8–17. <https://doi.org/10.1145/3088525.3088562>
- Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. <https://doi.org/10.1145/3371119>