Logic as a distributive law

Michael Stay[1]
and L.G. Meredith[2]

[1] Pyrofex Corp.
stay@pyrofex.net
[2] Synereo, Ltd
greg@synereo.com

Abstract. We present an algorithm for deriving a spatial-behavioral type system from a formal presentation of a computational calculus. Given a 2-monad Calc: Cat $\to$ Cat for the free calculus on a category of terms and rewrites and a 2-monad BoolAlg for the free Boolean algebra on a category, we get a 2-monad Form = BoolAlg + Calc for the free category of formulae and proofs. We also get the 2-monad BoolAlg $\circ$ Calc for subsets of terms.
The interpretation of formulae is a natural transformation $[\![-]\!]$: Form $\Rightarrow$ BoolAlg$\circ$Calc defined by the units and multiplications of the monads and a distributive law transformation δ: Calc $\circ$ BoolAlg $\Rightarrow$ BoolAlg $\circ$ Calc. This interpretation is consistent both with the Curry-Howard isomorphism and with realizability.
We give an implementation of the "possibly" modal operator parametrized by a two-hole term context and show that, surprisingly, the arrow type constructor in the λ-calculus is a specific case. We also exhibit nontrivial formulae encoding confinement and liveness properties for a reflective higher-order variant of the π-calculus.

1 Introduction

Sylvester coined the term "universal algebra" to describe the idea of expressing a mathematical structure as set equipped with functions satisfying equations; the idea itself was first developed by Hamilton and de Morgan [6]. Most modern programming languages have a notion of a "module" or an "interface" with this same information; for example, consider this Agda definition of a monoid.

```
data Monoid (M : Set)(Eq : Equivalence M) : Set where
  monoid :
    (e   : M)
    (_*_ : M -> M -> M)
    (leftId : LeftIdentity Eq z _+_)
    (rightId : RightIdentity Eq z _+_)
    (assoc : Associative Eq _+_)
```

The code defines a sort M, a nullary term constructor e, and a binary term constructor $*$, subject to three equations. In universal algebra, such a structure is called an "equational theory".

In 1963, Lawvere [12] showed that an equational theory was a presentation of a category with finite products where all the objects are powers of a single generating object; such a category is now called a Lawvere theory. The Agda code above is a presentation of a category Th(Mon) whose objects are $1, M, M^2, M^3, \ldots$, and whose morphisms are generated by projections, e, and $*$, modulo the equations. This category is purely syntactic.

To interpret the equational theory as denoting sets and functions, we use a product-preserving functor from the Lawvere theory to Set: such a functor maps M to a set and e and $*$ to functions satisfying the equations. The category Set encodes what we mean by semantics; if instead of Set we used the category Vect of vector spaces and linear maps and mapped the product in Th(Mon) to the tensor product in Vect, a model of the theory would be an associative algebra instead. If we used a category of endofunctors and natural transformations and mapped the product in Th(Mon) to composition, a model of the theory would be a monad.

The category of product-preserving functors from Th(Mon) to Set and natural transformations between them is equivalent to the category Mon of monoids and monoid homomorphisms. There is a forgetful functor $U\colon \mathrm{Mon} \to \mathrm{Set}$ that forgets all the structure with a left adjoint $L\colon \mathrm{Set} \to \mathrm{Mon}$ that picks out the free monoid on a set. Abusing notation somewhat, we use Mon also to mean the monad UL that picks out the underlying set of the free monoid on a set. This pattern is a general one: every Lawvere theory corresponds to a finitary monad on Set and vice-versa [7].

Often modules or interfaces are used to model data structures. In computing, however, we consider not only the structure of data but the behavior of processes that change the data. The λ-calculus is the paradigmatic model of functional programming. It has a single data type, called lambda term, described by a one-line grammar. In this model, computation consists of repeatedly applying a single rewrite rule called β-reduction; the rewrite rule matches the current term to a pattern, then rearranges the parts of the term. When the term no longer matches the pattern, the rewriting stops and the resulting term is taken as the "answer". Similarly, the π-calculus, arguably the paradigmatic model of concurrent programming, has a structured data type called a process that is also described by a simple grammar. Computations in the π-calculus are carried out by repeatedly applying rewrite rules, primarily the comm-rule.

This pattern has been recognized and formalized many times. In his seminal paper, "Functions as Processes" [18], Milner developed what is now the standard presentation of a computational calculus. The presentation consists of a grammar describing the primary data type over which computations are carried out, a structural equivalence, used to erase syntactic differences that are irrelevant to computation, and a set of rewrite rules describing how to realize computation through operations on the data structures. On one hand, this generalizes univer-

sal algebra's generators and relations presentation, with the grammar replacing the generators as the free construction, the structural equivalence replacing the relations, and the rewrites providing the computational semantics. On the other hand, this presentation maps quite well onto Plotkin's SOS format for operational semantics [19], but the scope is much larger. Even the presentation of the Java VM can be seen as essentially a rewrite system with the state of the virtual machine as instances of a data type over which computation is carried out, and the transitions of the virtual machine as the rewrite rules [8]. Capturing this higher-order computational phenomena, expressed as generalized rewrites, is what motivates our movement to higher categorical semantics.

Lawvere theories can be generalized from 1-categories to 2-categories. As one might expect, the corresponding notion of equational theory involves one more level of structure; we have sorts, term constructors, *rewrites* instead of equations between term constructors, and equations between rewrites. We typically interpret such a theory in Cat, the 2-category of categories, functors and natural transformations.

We can use Lawvere 2-theories to describe terms in a computational calculus and the permissible rewrites. We have a sort for terms; our examples are necessarily simple calculi due to space constraints, but our approach should work with any formalization of a computational model such as the K framework [20], the "kiama" Scala package [23], even packages largely focused on documentation and specification, such as Ott [22]. We also have term constructors for building up specific terms, rewrites between term constructors for running programs, and equations between rewrites to preserve invariants. The Lawvere 2-theory captures the syntax of a computational calculus; a 2-functor from the theory into Cat assigns operational semantics. Note that this is fully consistent with Curry-Howard style interpretations.

A spatial-behavioral type system is a language in which one can describe both the structure and future behavior of terms. In keeping with realizability style semantics [9], the interpretation of a formula should be a set of terms satisfying the formula. The interpretation of a proof should be a function that maps a set of assumptions to a set of consequents.

The obvious language for describing the structure of a term is the 2-theory Th(Calc) of the calculus itself, equipped with extra term constructors for true, false, disjunction, conjunction, and negation. This suggests adding the 2-monad Calc to the 2-monad BoolAlg for Boolean algebras to get Form, the 2-monad for formulae.

The obvious language for interpreting formulae is the 2-category of subsets of terms with pointwise rewrites between them. This suggests composing the monad BoolAlg with the monad Calc.

The obvious way to interpret formulae is then a natural transformation $[\![-]\!]\colon \mathrm{Form} \Rightarrow \mathrm{BoolAlg} \circ \mathrm{Calc}$ defined by the units and multiplications of the monads and a distributive law transformation $\delta\colon \mathrm{Calc} \circ \mathrm{BoolAlg} \Rightarrow \mathrm{BoolAlg} \circ \mathrm{Calc}$. Soundness and completeness follow easily.

We can extend this naive type system with fixed points and with modal operators that denote sets of terms that all exhibit a particular behavior; for an example of the latter, the arrow type $A \Rightarrow B$ in the λ-calculus is the type of terms that when applied to a term of type A eventually rewrite to a term of type B. Other formulae can capture eventual properties like

- Authority: it is the case that if my banking service eventually evolves to a state in which Alice has the ability to withdraw money from my account, then the banking service both sent an email to my address requesting confirmation and received the confirmation email in response.
- Confinement: this process will only ever send a messages on channels in this set.
- Liveness: this process will always eventually handle another message.
- Termination: this computation will always halt.
- Structure: at the end of sorting, the data will be ordered.

The slogan "logic via distributive law" is more apt than might first be supposed, however. At heart, the algorithm says that logical formula can be thought of as describing collections of terms in terms of terms built over collections. Since a very broad class of collections are captured via the notion of monad, the distributive law, together with the monad laws are iteratively applied to instances of the latter to arrive at instances of the former. In this sense BoolAlg is just one notion of "collection", namely that of Set. There are other notions of collection, such as list, bag, tree, all of which are captured as monads and for which we could use a distributive law to provide a semantics for a notion of formula that has utility and considerable expressive power. Thus, for example, if the notion of collection were *sets of sequences*, then the semantics generated by our algorithm would interpret formulae as sets of sequences of terms, instead of sets of terms, and the term language extracted for the collection monad would be the familiar operators of classical linear logic and this would correspond to the well-known fact that quantales provide full and faithfull models of the *formulae* of linear logic. The higher categorical machinery provides the right setting in which to express this idea in it's full generality. In this paper, however, we focus the exegesis of these results on BoolAlg to conserve space, and highlight the core ideas.

2 Related work

There is a very large body of work on Lawvere theories and their generalizations, and we cannot give a proper review here; instead we list a few key papers with many references. Hyland and Power [7] reviewed the history of both Lawvere theories and monads as an approach to universal algebra, how monads came to the fore, Plotkin's work, and the renewed interest in Lawvere theories in a computer science setting. Trimble [25] constructed multisorted Lawvere theories. Lack and Rosický [10] unified the work of several earlier authors by considering

models in categories other than Set, extending the notion of Lawvere theory to enriched categories, and using all limits rather than just products all at once.

Our results can be seen as a natural consequence of Abramsky's programme, laid out in "Domain theory in Logical Form" [1], and expanded in "Proofs as Processes" [2]. Moreover, they can be seen as bringing the seminal work of Caires [3] under the rubric of that programme.

3 A 1-categorical example

Here we return in slightly greater detail to the example of the logic for the language of monoids. Because we are working with categories instead of 2-categories, the "virtual machine" is particularly bland; it only has terms (the elements of the monoid), not rewrites.

Let FinSet be a skeleton of the category of finite sets. The Lawvere theory of a computational calculus is a category Th(Calc) with finite products equipped with an identity-on-objects functor $\theta\colon \mathrm{FinSet}^{\mathrm{op}} \to \mathrm{Th(Calc)}$. Because the objects of FinSet are coproducts of the one-element set, the objects of category Th(Calc) are therefore products of a generating object we will write as S, for "sort". The morphisms of Th(Calc) are generated from a set of morphisms from finite powers of S to S by products and composition, so Th(Calc) may be presented by

- a sort S,
- a set of term constructors $f_i\colon S^{n_i} \to S$, where i ranges over some index set I and $n_i \in \mathbb{N}$,
- and a set of equations between term constructors.

We say the arity of f_i is n_i.

A product-preserving functor from Th(Calc) to Set picks out a set and equips it with structure maps satisfying the equations. The category Prod(Th(Calc), Set) of product-preserving functors and natural transformations between them is equivalent to the category Calc of calculi and calculus homomorphisms. There is a forgetful functor $U\colon \mathrm{Calc} \to \mathrm{Set}$ that forgets the extra structure. The functor U has a left adjoint $L\colon \mathrm{Set} \to \mathrm{Calc}$ that picks out the free calculus on a set of base terms. The monad $\mathrm{Calc} = UL\colon \mathrm{Set} \to \mathrm{Set}$ picks out the underlying set ULX of the free calculus LX on a set X. Lawvere theories are in bijection with finitary monads; the qualifier "finitary" means that each term constructor has a finite arity.

Here is a presentation of the "Lawvere theory of a monoid" Th(Mon):

Sorts:	Equations:
– S	– $\cdot \circ (S \times \cdot) = \cdot \circ (\cdot \times S)$ (associativity)
Term constructors:	– $\cdot \circ (e \times S) \circ \mathrm{left}^{-1} = S$ (left unit)
– $\cdot\colon S^2 \to S$	– $\cdot \circ (S \times e) \circ \mathrm{right}^{-1} = S$ (right unit)
– $e\colon 1 \to S$	

where left$\colon 1 \times S \xrightarrow{\sim} S$ and right$\colon S \times 1 \xrightarrow{\sim} S$ are the canonical isomorphisms.

An implementation of this specification is a product-preserving functor from Th(Mon) to Set; such a functor will assign a set of values to the sort and functions to the term constructors such that the equations are satisfied, *i.e.* it will pick out a monoid. The category of product-preserving functors from Th(Mon) to Set and natural transformations between them is equivalent to the category of monoids and monoid homomorphisms. There is a forgetful functor $U\colon \mathrm{Mon} \to \mathrm{Set}$ that forgets the multiplication and unit, and outputs the underlying set of elements. The functor U has a left adjoint $L\colon \mathrm{Set} \to \mathrm{Mon}$ that outputs the free monoid on a set. The composite functor $\mathrm{Mon} = UL\colon \mathrm{Set} \to \mathrm{Set}$ is the corresponding monad.

All our formulae about monoids will denote subsets of the elements of the monoid; we use Th(BoolAlg) to describe them. We take as our universe of subsets those describable with a finite formula, so that the logic is complete by construction.

Sorts:	Equations:
– S	– associativity, commutativity and unit laws for $\wedge$ and $\vee$
Term constructors:	– distributivity of $\wedge$ over $\vee$
– $\wedge\colon S^2 \to S$	– involution for $\neg$
– $\vee\colon S^2 \to S$	– de Morgan's laws
– $\top\colon 1 \to S$	
– $\bot\colon 1 \to S$	
– $\neg\colon S \to S$	

From this theory, we can derive the monad BoolAlg for the free Boolean algebra on a set.

The sum of the two theories above is a new theory Th(Form) = Th(Mon) + Th(BoolAlg) whose terms are our formulae. Th(Form) is presented by identifying the sorts and taking the union of the term constructors and the union of the equations.

Sorts:	Equations:
– S Term constructors: – $\cdot\colon S^2 \to S$ – $e\colon 1 \to S$ – $\wedge\colon S^2 \to S$ – $\vee\colon S^2 \to S$ – $\top\colon 1 \to S$ – $\bot\colon 1 \to S$ – $\neg\colon S \to S$	– associativity and unit laws for $\cdot$ – associativity, commutativity, and unit laws for $\wedge$ and $\vee$ – involution for $\neg$ – de Morgan's laws

The process of deriving a monad from a theory preserves sums. Since the sum of two monads is the free product of the two, a general formula will be a term in an alternating composition of the two monads. For example, suppose that $a, b, c, d \in X$; then one formula is

$$((a \vee (b \cdot d)) \cdot (c \vee d)),$$

which is a term in Mon(BoolAlg(Mon(X))). The interpretation of this formula should be the set of monoid elements

$$\{ac, ad, bdc, bdd\},$$

or in other words, the term

$$(a \cdot c) \vee (a \cdot d) \vee ((b \cdot d) \cdot c) \vee ((b \cdot d) \cdot d)$$

in BoolAlg(Mon(X)).

In order to move all the uses of Mon to the right of the uses of BoolAlg in the alternating composition, we need a distributive law natural transformation

$$\delta\colon \mathrm{Mon} \circ \mathrm{BoolAlg} \Rightarrow \mathrm{BoolAlg} \circ \mathrm{Mon}.$$

Given δ and the monad units and multiplications, we can define an interpretation natural transformation

$$[\![-]\!]\colon \mathrm{Form} \Rightarrow \mathrm{BoolAlg} \circ \mathrm{Mon}$$

in the obvious way. Below, we write subsets of $\mathrm{Mon}(X)$ using set notation:

$$\begin{aligned}
[\![\top]\!]_X &= \mathrm{Mon}(X) \\
[\![\bot]\!]_X &= \emptyset \\
[\![A \vee B]\!]_X &= [\![A]\!]_X \cup [\![B]\!]_X \\
[\![A \wedge B]\!]_X &= [\![A]\!]_X \cap [\![B]\!]_X \\
[\![\neg A]\!]_X &= \mathrm{Mon}(X) - [\![A]\!]_X \\
[\![A \cdot B]\!]_X &= \mathrm{Mon}(\cdot)([\![A]\!]_X \times [\![B]\!]_X) \\
[\![e]\!]_X &= \{e\} \\
[\![x \in X]\!]_X &= \{x\}
\end{aligned}$$

Even in this simple example, we have nontrivial formulae; for example,

$$prime = \neg e \wedge \neg(\neg e \cdot \neg e)$$

is a 1-line formula for primality. For the monoid of natural numbers under multiplication, this formula says a number is prime if it is neither 1 nor has a nontrivial factorization. It is easy to verify that $[\![prime]\!]_X = X$.

4 Moving to 2-categories

In a computational context, when a data structure has some symmetry we do not care about, we often test two instances of the structure for equality by computing a normal form for each instance and then comparing the normal forms. Monoids are associative, but we can imagine storing words of a monoid as binary trees internally and then comparing them by computing a normal form. For the case of a normal form for the trees, we can eliminate 1 in a product by rewriting $(1 \cdot x)$ and $(x \cdot 1)$ to x, and we can shift all the parentheses to the right by rewriting $((x \cdot y) \cdot z)$ to $(x \cdot (y \cdot z))$.

The Lawvere 2-theory Th(Mon) is much the same as the 1-theory, except for the weakening of some equations to rewrites and the addition of new equations between the rewrites.

Sorts:	Rewrites:
– S	– $a\colon\ \cdot \circ (S \times \cdot) \Rightarrow \cdot \circ (\cdot \times S)$
Term constructors:	– $l\colon\ \cdot \circ (e \times S) \circ \mathrm{left}^{-1} \Rightarrow S$
– $\cdot\colon S^2 \to S$	– $r\colon\ \cdot \circ (S \times e) \circ \mathrm{right}^{-1} \Rightarrow S$
– $e\colon 1 \to S$	Equations:
	– $a \circ a = (S \times a) \circ a \circ (a \times S)$ (pentagon equation)
	– $r \times S = (S \times l) \circ a$ (triangle equation)

From Th(Mon) we can derive a 2-monad Mon that essentially produces the free monoid on a set, but keeps track of the internal representation of the monoid—a binary tree—and accounts for the work needed to convert the internal representation to its normal form. From this perspective, we can think of the trees as programs for a simple computational calculus and the process of normalization as the execution of the program. Later in the paper, we will examine the case of the SKI combinator calculus, a Turing-complete programming language that was a predecessor to the λ-calculus; it, too, uses binary trees as programs, and execution of the program is a normalization process.

When we add the 2-monad BoolAlg to Mon, we get formulae like $(1 \cdot \top)$ denoting the set of trees whose leftmost child is the identity; because Mon is a 2-monad, we also get proofs like

$$((r \circ l) \vee y) \colon ((e \cdot (x \cdot e)) \vee y) \to (x \vee y)$$

whose interpretations are homomorphisms of Boolean algebras.

The formulae in these examples have been propositions about the structure of terms. Later in the paper, we will also show how to add modal operators that are propositions about the behavior of terms; the arrow type constructor from the λ-calculus is a prominent example.

5 Multisorted Lawvere theories

In the motivation section, each theory had only one sort; practical theories for virtual machines are usually multisorted. Given a finite set of sorts Σ, the category FinSet/Σ is the category whose objects are pairs $(S, s \colon S \to \Sigma)$, where S is a finite set, and whose morphisms are functions $f \colon S \to S'$ such that the relevant triangle commutes.

A multisorted Lawvere theory is a category Th(Calc) with finite products equipped with an identity-on-objects functor $\theta \colon (\mathrm{FinSet}/\Sigma)^{\mathrm{op}} \to \mathrm{Th(Calc)}$. The category $\mathrm{Prod}(\mathrm{Th(Calc)}, \mathrm{Set}^{\Sigma})$ is equivalent to the category Calc of calculi and calculus homomorphisms. There is a forgetful functor $U \colon \mathrm{Calc} \to \mathrm{Set}^{\Sigma}$ with a left adjoint $L \colon \mathrm{Set}^{\Sigma} \to \mathrm{Calc}$ that picks out the free calculus on a $|\Sigma|$-tuple of sets. The monad $UL \colon \mathrm{Set}^{\Sigma} \to \mathrm{Set}^{\Sigma}$ picks out the underlying $|\Sigma|$-tuple of sets ULX of the free calculus LX on a $|\Sigma|$-tuple of sets X.

An example of a multisorted Lawvere theory is that of a group action on a set, which involves a choice of both a group G and a set V to act on. The presentation of Th(GrpAct) has a pair of sorts (G, V), all the term constructors and equations as the theory of a group (where we replace S by G), together with a new term constructor

- $a \colon G \times V \to V$

and equations

- $a \circ (e \times V) \circ \mathrm{left}_V^{-1} = V$ (identity action)
- $a \circ (m \times V) = a \circ (G \times a)$ (compatibility).

Another example is the theory of a directed graph, with one sort for vertices and another for edges, with source and target maps for term constructors.

6 Lawvere 2-theories

In this paper, the multisorted Lawvere 2-theory of a calculus is a 2-category Th(Calc) with strict finite products (that is, its underlying category has products) equipped with an identity-on-objects functor $\theta\colon (\mathrm{FinSet}/\Sigma)^{\mathrm{op}} \to \mathrm{Th(Calc)}$, where we promote (FinSet/Σ) to a 2-category by adding identity 2-morphisms to every 1-morphism. As noted in the related work section, other authors have considered much more powerful notions of 2-theory, but we will, for the most part, not need the extra features; however, see the notion of "lambda theory" below. Our notion of a multisorted Lawvere 2-theory may be presented by a finite set of sorts, a set of term constructors with finite arity, a set of rewrites, and a set of equations between rewrites.

Our models of multisorted Lawvere 2-theories are functors into Cat^{Σ} that preserve products up to isomorphism, not merely up to equivalence; that is, the 2-functor has an underlying functor that preserves products. As with 1-theories, the 2-category of product-preserving functors from Th(Calc) to Cat^{Σ}, natural transformations, and modifications is equivalent to the 2-category of calculi, calculus homomorphisms, and calculus transformations.

As mentioned above, the SKI combinator calculus is a Turing-complete language; it was invented by Schönfinkel [21] and Curry [4] in the 1920s as a way to clarify the role of quantified variables in logic, essentially by eliminating them. The single-sorted Lawvere 2-theory Th(SKI) has a presentation

Sorts:	Rewrites:
– T	– $\forall x, y, z \in T, \quad \sigma\colon (((S\ x)\ y)\ z) \Rightarrow ((x\ z)\ (y\ z))$ – $\forall y, z \in T, \quad \kappa\colon ((K\ y)\ z) \Rightarrow y$ – $\forall z \in T, \quad \iota\colon (I\ z) \Rightarrow z$
Term constructors: – $S\colon 1 \to T$ – $K\colon 1 \to T$ – $I\colon 1 \to T$ – $(- -)\colon T^2 \to T$,	No equations.

In this context, the Church-Rosser theorem for the SKI calculus says that any two terminating rewrites out of an SKI term have the same codomain. We do not usually want to impose equality on the rewrites, since they can differ greatly in computational complexity. For example, suppose that we have the term $((K\ I)\ x)$, where x is some term that takes a long time to reduce to its normal form; a rewrite that reduces x first and then uses κ takes much longer than just doing κ first, though both rewrites begin and end at the same term.

The free model of Th(SKI) on a category takes its objects as terms and its morphisms as rewrites, then freely adjoins S, K, and I and all applications of one object to another, as well as new morphisms generated by σ, κ, and ι. The free model on the empty category will contain only terms and rewrites from the SKI calculus.

Just as Lawvere theories capture the notion of a set equipped with functions satisfying equations, 2-Lawvere theories capture the notion of a category equipped with functors and natural transformations satisfying equations. The former describes what might be called data structures, while the latter captures both the state of a system and the processes that update that state. In the next section, we introduce a language for talking about sets of states and updates that satisfy some criterion, terms and rewrites that share some property.

7 Categories of formulae and proofs

Any 1-theory can be promoted to a 2-theory by turning each equation into a rewrite, then adding an equation asserting that the rewrite is equal to the identity rewrite. Therefore as before, given a single-sorted Lawvere 2-theory Th(Calc), we get a 2-theory of formulae Th(Form) by adding the 2-theory of Boolean algebras Th(BoolAlg). The models of Th(Form) are categories of formulae and proofs.

For multisorted Lawvere 1- and 2-theories, there is no canonical choice of sorts to identify between the theory of the calculus and the theory of Boolean algebras; one may choose to add one or more different copies of Th(BoolAlg) and identify each single sort with different sorts in the theory of the calculus. For example, given the theory of a group action, one could add a copy of Th(BoolAlg) for both G and V and write formulae involving subsets of the group and of the set it acts on. Later in the paper, we will see how we can use formulae involving names to create namespaces that enforce confinement on processes in the π-calculus.

8 Interpretation

When describing the semantics of a logic, questions of soundness and completeness are natural. The whole point of our approach is that it is correct by construction. Soundness and completeness for the modal-free fragment follow directly from the definitions; soundness, in particular, is a direct consequence of the way realizability is incorporated into the approach. For the formulae, the intuitions underlying the completeness argument amount to the fact that the term language for the logical operators associated with the collections are extractions of a term language for the collection monad. Thus, they are in perfect correspondence with the model. In the case of Set, this is equivalent to the well-known fact that boolean algebras are effectively realized by the powerset lattice.

Likewise, in the case of Set, the semantics that maps formulae to the composition of the collection monad with the monad for the term language supporting computation is nothing more than a pointwise lifting of the powerset semantics. The categorical presentation provides an appropriately abstracted form of this simple observation.

9 Modal operators

So far, all the examples of formulae have dealt with the structure of the term. Far more interesting are sets of terms that all share some behavior. For example, in the SKI calculus, we want to add the idea of an arrow type to our formulae, and have the interpretation of $A \Rightarrow B$ be the subset of terms t such that given a term $u \in [\![A]\!]$, the term $(t\ u)$ eventually evolves to a term $v \in [\![B]\!]$.

This notion of eventuality is an example of a modal operator. Possibility is another, where possibility is to eventuality as "there exists" is to "for all". In the SKI calculus, they coincide due to the fact that the calculus is confluent, but they differ in the general case. Many interesting properties can be stated in terms of eventual and possible states as noted earlier.

To add a modal operator to a formula language, we first add it formally as a term constructor to the term theory, then add the collection theory as before.

To interpret the modal operator, we first interpret the formulae as above, then interpret modal terms as collections of terms, then use the join from the collection monad.

9.1 Example: Arrow types in the SKI calculus

The Lawvere 2-theory of the arrow type Th(Arrow) has one term constructor, no rewrites, and no equations. The lack of rewrites and equations are because the arrow is a purely formal type, and it is only in our choice of semantics that it acquires its customary interpretation.

In order to avoid notational confusion between the arrow type constructor and 2-morphisms, we will use a triple-arrow in the theory.

Sorts: – T Term constructors: – $\Rrightarrow : T^2 \to T$	No rewrites; no equations.

We get the theory of SKI with arrow Th(SKIArr) by adding Th(SKI) to Th(Arrow), and we get the theory of formulae Th(Form) by adding Th(SKIArr) to Th(BoolAlg).

To interpret terms from Th(Form), we compose the interpretation natural transformation above with another natural transformation $\alpha \colon \mathrm{SKIArr} \Rightarrow \mathrm{BoolAlg} \circ \mathrm{SKI}$, followed by the join from the collection monad. In particular,

$$\alpha(u \Rrightarrow v) = \{t \mid \exists \rho \colon (t\ u) \Rightarrow v\}.$$

In Lambek's 1980 paper [11] on the denotational semantics of the λ-calculus, he defined a category whose objects were types and whose morphisms were

equivalence classes of lambda terms with one free variable. In a future paper, we will show how Mellies and Zeilberger's approach to type refinement lets us recapitulate Lambek's construction and extend the arrow to a profunctor.

9.2 Modal operators parametric in a term constructor

A term context is a term with a "hole" that can be filled by some other term. Given a Lawvere theory with a sort S for terms, one can derive a new theory of term contexts by replacing each term constructor $f\colon S^n \to S$ with n term constructors $f_i\colon S^{n-1} \to S$ where $1 \leq i \leq n$. We think of f_i as being f with the ith input being a hole. Equivalently, if we have a theory with coproducts, we can replace each occurrence of S on the left-hand side of a term constructor with $S + 1$, where the new point represents the hole.

The arrow type is a special case of a more general modal operator parametrized by a two-hole term context C. We denote the operator itself with angle brackets as a reminder of the diamond "possibly" modality: $u\langle C\rangle v$. The interpretation is similar to that of the arrow:

$$[\![u\langle C\rangle v]\!] = \{t \mid \exists \rho\colon C[t, u] \Rightarrow v\};$$

that is, $u\langle C\rangle v$ denotes those terms t that when put in the context u may possibly evolve to v.

This definition is inspired by the work of Sewell, Leifer, and Milner, in which they formalize the notion of a context-labelled transition [13]. Given the now standard Hennessy-Milner interpretation of modal logics which interpret possibility in terms of actions labelling transitions, when Sewell *et al.* gave a notion of context-labelled transitions, it was only natural to consider modal operators based on contexts.

10 Recursion

Suppose that we have a single-sorted Lawvere 2-theory of a calculus with sort S. We add recursion to our formulae by introducing a new sort V for type variables and new term constructors

- $-\colon V \to S$ to let us use type variables in formulae and
- $\mu\colon V \times S \to S$ to express fixed points.

We interpret terms of the form $\mu X.P[X]$ as the greatest fixed point of $P[X]$.

10.1 The reflective higher-order π-calculus

Lawvere theories are limited in that they only talk about products of sorts. A lambda theory is a generalization of a Lawvere theory that has the ability to talk about function sorts like $A \Rightarrow B$ and sums like $A + B$ or $1 + A + A^2 + \cdots$, more commonly denoted with the Kleene star A^*. One can think of lambda theories

as having access to a "library" that takes care of the details of bound variables and substitution for us so we do not have to implement all that machinery. Much of the work on nominal logics has been about exactly this factorization [5]. A lambda theory Th(Calc) is a bicartesian closed 2-category equipped with an identity-on-objects functor from $(\mathrm{FinSet}/\Sigma)^{\mathrm{op}}$ to Th(Calc). Models of Th(Calc) are functors to Cat that preserve all the structure.

The π-calculus was invented in the early 1990s by Robin Milner as a model of networks of processes with a dynamically changing topology; two processes initially unaware of each other can be introduced by a third process. The reflective higher order π-calculus uses quoted processes as names; the term constructors for quote and eval replace the more traditional nu and replicate constructors. We also add a "comm" term to restrict the contexts in which reduction can occur [24].

Here is a presentation of the multisorted lambda theory Th(RHOpi) for the reflective higher-order π-calculus:

Sorts:	Rewrites:
– N for names – P for processes	– α: $(p_1 \mid p_2) \mid p_3 \Rightarrow p_1 \mid (p_2 \mid p_3)$ – β: $p_1 \mid p_2 \Rightarrow p_2 \mid p_1$ – ι: $0 \mid p \Rightarrow p$
Term constructors – send: $N \times P^* \to P$ – recv: $N \times (N^* \Rightarrow P) \to P$ – $\mid$: $P^2 \to P$ – 0: $1 \to P$ – comm: $1 \to P$ – "$-$": $P \to N$ – eval$(-)$: $N \to P$	– χ: send$(x, p_1, \ldots, p_n) \mid$ recv$(x, q) \mid$ comm $\Rightarrow$ $q(\text{“}p_1\text{”}, \ldots, \text{“}p_n\text{”}) \mid$ comm – ϵ: eval(“p”) $\Rightarrow p$ Equations: – $\alpha = P^3, \beta = P^2, \iota = P$ ($\mid$ and 0 form a commutative monoid) – $\epsilon = P$ (evaluating a quoted process is the same as the process itself)

The simplest RHOpi processes are 0, the "do nothing" process; and comm, a "catalyst" process that enables communication on a channel. The only rewrite that is not an identity is χ, the communication event. The χ rewrite is neither confluent nor deterministic. For example, we can model contention for resources with the term

$$\text{recv}(x, P) \mid \text{recv}(x, Q) \mid \text{send}(x, R) \mid \text{comm}$$

which has two rewrites out of it, one where the continuation P is invoked on the name "R" and the other where the continuation Q is invoked on it. We can model message arrival order nondeterminism with the term

$$\text{send}(x, P) \mid \text{send}(x, Q) \mid \text{recv}(x, R) \mid \text{comm}$$

which has two similar rewrites out of it, one where the continuation P is invoked on the name "R" and one where P is invoked on the name "Q".

In the theory above, comm is preserved by the rewrites; one can think of each comm instance as representing a processor. An alternative would be to consume comm in the χ rewrite; then comm would track clock ticks; an application of a consumable comm is the formal verification of billing code for tracking compute resources.

The free RHOpi calculus R on the pair of empty categories has only *unguessable* names, since they can be enumerated; the free RHOpi calculus on a category of generating names and a category of generating terms has extra names that are *unforgeable*. For more on this distinction, see [17].

Replication of processes, and therefore general recursion, can be encoded [15] via

$$D(x) = \text{recv}(x, y \mapsto \text{send}(x, \text{eval}(y))|\text{eval}(y))$$

$$!P = \text{send}(x, D(x)|P)|D(x).$$

Caires' [3] operator $\triangleright$ for rely-guarantee properties, the adjunct to $|$, is an instance of our generic modal operator, where $u \triangleright v = u\langle -|-\rangle v$:

$$[\![u \triangleright v]\!] = \{t \mid \exists \rho\colon (t \mid u) \Rightarrow v\}$$

The logic is generated as above, adding modal operators and recursion to the term language, then adding the Boolean algebra; it is equivalent to the logic we generated by hand in [14]. As an example of a formula in this logic, here is a one-line formula for a process that will always be able to handle another message:

$$\mu X.\text{recv}(\top, x \mapsto X) \mid \top.$$

It says that the process must factor into a piece that is waiting for a message on some channel, and when it receives that message, the continuation will have the same form.

Namespaces When we add a copy of BoolAlg to RHOpi for both N and P, we can write down formulae that talk about sets of names. The formula $\top$ denotes the set of all names; the formula "$\top$" denotes the set of names that are quoted processes. These two sets differ when we have a nonempty set of generating names.

An application of namespace logic is a formula for a "firewall": any process satisfying this formula receives no messages except on a name in "ϕ".

$$\mu X.\text{recv}(\text{“}\phi\text{”}, x \mapsto (X \vee 0) \mid \neg\text{recv}(\text{“}\neg\phi\text{”}, \top)) \mid \neg\text{recv}(\text{“}\neg\phi\text{”}, \top)$$

It is, perhaps, surprising that this can be accomplished with a compile-time check, since we usually think of firewalls as a dynamic check. An application of namespaces to statically proving security properties of code in an object capabilities language can be found in [16].

11 Conclusion and future work

We have presented an algorithm for generating a logic from three data: a 2-Lawvere theory describing a notion of computation, a monad for describing a notion of collection, and a distributive law. As mentioned in the introduction, the picture emerging from this view of logic is that logically useful collections of individuals look like nothing so much as the individuals that represent the formulae that collect them. The range of applications of this algorithm is quite broad. Any of the languages given semantics in the K Framework, including, C/C++, Java, Javascript, and many others can automatically be equipped with logics that serve as a basis for of program specification and automatically checked verification, and simultaneously as a basis equipping these languages with new and powerful type systems. This considerably lowers the barrier to embuing popular untyped languages with type systems. Likewise, the algorithmic nature of the approach also serves as an aid in future language design. For example, Synereo is designing a language for the smart contracts on the blockchain based on the RHO-calculus. Equipping this language with a type system amounts to nothing more than deciding on the type of collection one wants to collect type inhabitants in.

References

1. Samson Abramsky, *Domain theory in logical form*, Ann. Pure Appl. Logic **51** (1991), no. 1-2, 1–77.
2. Samson Abramsky, *Proofs as processes*, Theor. Comput. Sci. **135** (1992), no. 1, 5–9.
3. Luis Caires, *Logical semantics of types for concurrency*, *Algebra and Coalgebra in Computer Science, Lecture Notes in Computer Science (2007), 2007, pp. 16–35.*
4. *H Curry,* Grundlagen der kombinatorischen logik*,* American Journal of Mathematics **52 (3)** (1930), 509–536.
5. M. J. Gabbay, *The π-calculus in FM*, Thirty-five years of Automath (Fairouz Kamareddine, ed.), Kluwer, 2003.
6. G. Grätzer, universal algebra, Van Nostrand, 1968.
7. Martin Hyland and John Power, *The category theoretic understanding of universal algebra: Lawvere theories and monads*, Electr. Notes Theor. Comput. Sci. **172** (2007), 437–458.
8. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler, *Featherwieght java: A minimal core calculus for java and GJ*, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999. (Brent Hailpern, Linda M. Northrop, and A. Michael Berman, eds.), ACM, 1999, pp. 132–146.
9. Jean-Louis Krivine, *The curry-howard correspondence in set theory*, Proceedings of the Fifteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2000 (Martin Abadi, ed.), IEEE Computer Society Press, June 2000.
10. Stephen Lack and Jirí Rosický, *Notions of lawvere theory*, Applied Categorical Structures **19** (2011), no. 1, 363–391.
11. J. Lambek, *From lambda calculus to cartesian closed categories*, (1980), 376–402.

12. William F. Lawvere, *Functorial Semantics of Algebraic Theories*, Ph.D. thesis, 2004, pp. 1–121.
13. James J. Leifer and Robin Milner, *Deriving bisimulation congruences for reactive systems*, CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings (Catuscia Palamidessi, ed.), Lecture Notes in Computer Science, vol. 1877, Springer, 2000, pp. 243–258.
14. L. Gregory Meredith and Matthias Radestock, *Namespace logic: A logic for a reflective higher-order calculus.*, in *TGC* [15], pp. 353–369.
15. ———, *A reflective higher-order calculus.*, Electr. Notes Theor. Comput. Sci. **141** (2005), no. 5, 49–67.
16. Lucius Gregory Meredith, Mike Stay, and Sophia Drossopoulou, *Policy as types*, CoRR **abs/1307.7766** (2013).
17. M Miller, K.-P. Yee, and J Shapiro, *Capability myths demolished.*
18. Robin Milner, *Functions as processes*, Mathematical Structures in Computer Science **2** (1992), no. 2, 119–141.
19. Gordon D. Plotkin, *The origins of structural operational semantics*, Journal of Logic and Algebraic Programming, 2004, pp. 60–61.
20. Grigore Rosu and Traian-Florin Serbanuta, *An overview of the K semantic framework*, J. Log. Algebr. Program. **79** (2010), no. 6, 397–434.
21. Moses Schönfinkel, *On the Building Blocks of Mathematical Logic*, From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931 (Jean van Heijenoort, ed.), iuniverse.com, 1924, pp. 355–366.
22. Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa, *Ott: Effective tool support for the working semanticist*, J. Funct. Program. **20** (2010), no. 1, 71–122.
23. Anthony M. Sloane, *Lightweight language processing in kiama*, Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers (João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, eds.), Lecture Notes in Computer Science, vol. 6491, Springer, 2009, pp. 408–425.
24. Mike Stay and Lucius Gregory Meredith, *Higher category models of the pi-calculus*, CoRR **abs/1504.04311** (2015).
25. T Trimble, *Multisorted lawvere theories.*