Logic as a distributive law

Michael Stay
Pyrofex Corp.
stay@pyrofex.net

L.G. Meredith
Synereo, Ltd
greg@synereo.com

October 2, 2016

Abstract

We present an algorithm for deriving a spatial-behavioral type system from a formal presentation of a virtual machine. Given a 2-monad $\mathrm{Mach}\colon \mathrm{Cat} \to \mathrm{Cat}$ for the free machine on a category of states and rewrites and a 2-monad BoolAlg for the free Boolean algebra on a category, we get a 2-monad $\mathrm{Form} = \mathrm{BoolAlg} + \mathrm{Mach}$ for the free category of formulae and proofs. We also get the 2-monad $\mathrm{BoolAlg} \circ \mathrm{Mach}$ for subsets of machine states.

The interpretation of formulae is a natural transformation $[\![-]\!]\colon \mathrm{Form} \Rightarrow \mathrm{BoolAlg}\circ\mathrm{Mach}$ defined by the units and multiplications of the monads and a distributive law transformation $\delta\colon \mathrm{Mach} \circ \mathrm{BoolAlg} \Rightarrow \mathrm{BoolAlg} \circ \mathrm{Mach}$. 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 lambda 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 [**?**]. 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 [?] 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 [?].

Often modules or interfaces are used to model structured data, aka data structures. For example, monoidal structure underlies the familiar data type of strings, the inhabitants of which include such famous examples as "colorless green ideas sleep furiously", and "the quick brown fox", the composition of which is string concatenation, and the identity of which is the empty string. In computing, however, we consider a wider range of phenomena than simply structural phenomena. Specifically, our primary focus is computation, which – though it utilizes structured data – is of a different order.

This is recognized in most formal representations of computation. The λ-calculus, the paradigmatic model of functional programming, has a structured data type, namely lambda terms, described by a 1-line grammar, over which computations are carried out via β-reduction. Likewise, the π-calculus, arguably the paradigmatic model of concurrent computation, enjoys a structured data type, usually called processes, also described by a simple grammar, over which computations are carried out via the comm-rule, and a small set of additional rewrite rules. In point of fact, this pattern has been recognized and formalized many times. Milner, in his seminal paper, Functions as Processes, developed what is now the standard presentation of a computational calculus, decomposed into a grammar, describing the primary data type over which computations are carried out, a structural equivalence, used to erase syntactic differences in inhabitants of the data structure that are irrelevant to computation, and a set of rewrite rules describing how computation is realized through operations on the data structures. This presentation maps quite well onto Plotkin's SOS format for operational semantics; 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 the inhabitants of a data type over which computation is carried out, and the transitions of the virtual machine as the rewrite rules. Capturing this higher order computational phenomena, expressed as generalized rewrites, is what motivates our movement to higher categorical semantics.

Specfically, 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 the state of a virtual machine and the permissible rewrites. We have a sort for machine states; since our examples are necessarily simple term calculi due to space constraints, we will also refer to machine states as "terms". We also have term constructors for building up specific states, rewrites between term constructors for running the machine, and equations between rewrites to preserve invariants. The Lawvere 2-theory captures the syntax of a virtual machine; a 2-functor from the theory into Cat assigns operational semantics.

A spatial-behavioral type system is a language in which one can describe both the structure and future behavior of virtual machine states. 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. This is consistent both with the Curry-Howard isomorphism and with realizability.

The obvious language for describing the structure of a machine state is the 2-theory Th(Mach) of the machine itself, equipped with extra term constructors for true, false, disjunction, conjunction, and negation. This suggests adding the 2-monad Mach 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 machine states with pointwise rewrites between them. This suggests composing the monad BoolAlg with the monad Mach.

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

2 Related work

[[Greg: Curry-Howard related work?]] [[Lots of stuff on Lawvere theories and generalizations.

- Lawvere, Functorial Semantics of Algebraic Theories

The original insight.

- Hyland & Power, "The Category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads"

Lawvere theories and monads have been the two main category theoretic formulations of universal algebra, Lawvere theories arising in 1963 and the connection with monads being established a few years later. Monads, although mathematically the less direct and less malleable formulation, rapidly gained precedence. A generation later, the definition of monad began to appear extensively in theoretical computer science in order to model computational effects, without reference to universal algebra. But since then, the relevance of universal algebra to computational effects has been recognised, leading to renewed prominence of the notion of Lawvere theory, now in a computational setting. This development has formed a major part of Gordon Plotkins mature work, and we study its history here, in particular asking why Lawvere theories were eclipsed by monads in the 1960s, and how the renewed interest in them in a computer science setting might develop in future.

- Lack & Rosický, "Notions of Lawvere theory"

Categorical universal algebra can be developed either using Lawvere theories (single-sorted finite product theories) or using monads, and the category of Lawvere theories is equivalent to the category of finitary monads on Set. We show how this equivalence, and the basic results of universal algebra, can be generalized in three ways: replacing Set by another category, working in an enriched setting, and by working with another class of limits than finite products.

- Trimble, "multisorted Lawvere theories"

Theorem 1. Let C be a category which admits general colimits and finite products that distribute over colimits. Then the forgetful functor ModC(Theta) to CLambda is monadic.

- Lack & Power, “Lawvere 2-theories”

http://www.mat.uc.pt/ categ/ct2007/slides/lack.pdf

- Power, “Enriched Lawvere theories”

We define the notion of enriched Lawvere theory, for enrichment over a monoidal biclosed category V that is locally finitely presentable as a closed category. We prove that the category of enriched Lawvere theories is equivalent to the category of finitary monads on V. Moreover, the V-category of models of a Lawvere V-theory is equivalent to the V-category of algebras for the corresponding V-monad. This all extends routinely to local presentability with respect to any regular cardinal. We finally consider the special case where V is *Cat*, and explain how the correspondence extends to pseudo maps of algebras.

- more]]

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 states (the elements of the monoid), not rewrites.

Let FinSet be a skeleton of the category of finite sets. The Lawvere theory of a machine is a category Th(Mach) with finite products equipped with an identity-on-objects functor $\theta\colon \mathrm{FinSet}^{\mathrm{op}} \to \mathrm{Th(Mach)}$. Because the objects of FinSet are coproducts of the one-element set, the objects of category Th(Mach) are therefore products of a generating object we will write as S, for “sort”. The morphisms of Th(Mach) are generated from a set of morphisms from finite powers of S to S by products and composition, so Th(Mach) 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(Mach) to Set picks out a set and equips it with structure maps satisfying the equations. The category Prod(Th(Mach), Set) of product-preserving functors and natural transformations between them is equivalent to the category Mach of machines and machine homomorphisms. There is a forgetful functor $U\colon \mathrm{Mach} \to \mathrm{Set}$ that forgets the extra structure. The functor U has a left adjoint $L\colon \mathrm{Set} \to \mathrm{Mach}$ that picks out the free machine on a set of base states. The monad $\mathrm{Mach} = UL\colon \mathrm{Set} \to \mathrm{Set}$ picks out the underlying set ULX of the free machine 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:
  - S
- Term constructors:
  - $\cdot\colon S^2 \to S$
  - $e\colon 1 \to S$
- Equations:
  - $\cdot \circ (S \times \cdot) = \cdot \circ (\cdot \times S)$ (associativity)
  - $\cdot \circ (e \times S) \circ \mathrm{left}^{-1} = S$ (left unit)
  - $\cdot \circ (S \times e) \circ \mathrm{right}^{-1} = S$ (right unit)

where $\mathrm{left}\colon 1 \times S \xrightarrow{\sim} S$ and $\mathrm{right}\colon S \times 1 \xrightarrow{\sim} S$ 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. [[Can squeeze this since not minimal.]]

- Sorts:
  - S
- Term constructors:
  - $\wedge\colon S^2 \to S$
  - $\vee\colon S^2 \to S$
  - $\top\colon 1 \to S$
  - $\bot\colon 1 \to S$
  - $\neg\colon S \to S$

- Equations:
  - associativity, commutativity and unit laws for $\wedge$ and $\vee$
  - distributivity of $\wedge$ over $\vee$
  - involution for $\neg$
  - de Morgan's laws

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:
  - S
- Term constructors:
  - $\cdot \colon S^2 \to S$
  - $e \colon 1 \to S$
  - $\wedge \colon S^2 \to S$
  - $\vee \colon S^2 \to S$
  - $\top \colon 1 \to S$
  - $\bot \colon 1 \to S$
  - $\neg \colon S \to S$
- Equations:
  - 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:

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

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

$$\mathit{prime} = \neg e \wedge \neg(\neg e \cdot \neg e)$$

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 $[\![\mathit{prime}]\!]_X = X$.

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:
  - S
- Term constructors:
  - $\cdot\colon S^2 \to S$
  - $e\colon 1 \to S$
- Rewrites:
  - $a\colon\ \cdot \circ (S \times \cdot) \Rightarrow \cdot \circ (\cdot \times S)$
  - $l\colon\ \cdot \circ (e \times S) \circ \mathrm{left} \Rightarrow S$
  - $r\colon\ \cdot \circ (S \times e) \circ \mathrm{right} \Rightarrow S$
- 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 virtual machine 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 lambda 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 lambda 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(Mach) with finite products equipped with an identity-on-objects functor $\theta\colon (\mathrm{FinSet}/\Sigma)^{\mathrm{op}} \to \mathrm{Th(Mach)}$. The category Prod(Th(Mach), Set^Σ) is equivalent to the category Mach of machines and machine homomorphisms. There is a forgetful functor $U\colon \mathrm{Mach} \to \mathrm{Set}^\Sigma$ with a left adjoint $L\colon \mathrm{Set}^\Sigma \to \mathrm{Mach}$ that picks out the free machine 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 machine 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

- $a \circ (e \times V) \circ \mathrm{left}_V = V$ (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 machine is a 2-category Th(Mach) with strict finite products (that is, its underlying category has products) equipped with an identity-on-objects functor $\theta\colon (\mathrm{FinSet}/\Sigma)^{\mathrm{op}} \to \mathrm{Th(Mach)}$, 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 not need the extra features. 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(Mach) to Cat^Σ, natural transformations, and modifications is equivalent to the 2-category of machines, machine homomorphisms, and machine transformations.

As mentioned above, the SKI combinator calculus is a Turing-complete language; it was invented by Moses Schoenfinkel and Haskell Curry 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:
  - T
- Term constructors:
  - $S\colon 1 \to T$
  - $K\colon 1 \to T$
  - $I\colon 1 \to T$
  - $(-\ -)\colon T^2 \to T$,
- Rewrites:
  - $\forall x, y, z \in T, \quad \sigma\colon (((S\ x)\ y)\ z) \Rightarrow ((x\ z)\ (y\ z))$
  - $\forall y, z \in T, \quad \kappa\colon ((K\ y)\ z) \Rightarrow y$
  - $\forall z \in T, \quad \iota\colon (I\ z) \Rightarrow z$
- 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.

[[Concluding comment.]]

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(Mach), 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 2-theories, there is no canonical choice of sorts to identify between the theory of the machine and the theory of Boolean algebras, but in practical applications, there is often an obvious choice. [[Add example.]]

That said, one may also choose to add two different copies of Th(BoolAlg) and identify each single sort with different sorts in the theory of the machine; [[example]]. Later in the paper, we will see how we can use formulae involving names to create namespaces that enforce containment on processes in the π-calculus.

8 Interpretation

[[Greg: Thm defining $\wedge$ as intersect, $\vee$ as union, $\top$ as the whole set, $\perp$ as the empty set, $\neg$ as complement, and terms over collections as mapping the term constructor over the cartesian product of the collections is sound and complete.]]

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 $v \in [\![B]\!]$.

[[Fix up eventually/possibly since they're the same here.]] 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". 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 [?] on the denotational semantics of the lambda 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 $1 \le i \le n$. 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.

[[Ref to Sewell Milner Leifer. Very close to labelling transitions with contexts.]]

10 Recursion

Suppose that we have a single-sorted Lawvere 2-theory of a virtual machine 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

[[Contradicting earlier assertion.]] 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. A lambda theory Th(Mach) is a bicartesian closed 2-category equipped with an identity-on-objects functor from $(\mathrm{FinSet}/\Sigma)^{\mathrm{op}}$ to Th(Mach). Models of Th(Mach) 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 [**?**].

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

- Sorts:
  - N for names
  - P for processes
- Term constructors
  - $\mathrm{send}\colon N \times P^* \to P$
  - $\mathrm{recv}\colon N \times (N^* \Rightarrow P) \to P$
  - $|\colon P^2 \to P$
  - $0\colon 1 \to P$
  - $\mathrm{comm}\colon 1 \to P$
  - $``-"\colon P \to N$
  - $\mathrm{eval}(-)\colon N \to P$
- Rewrites:
  - $\alpha\colon (p_1|p_2)|p_3 \Rightarrow p_1|(p_2|p_3)$
  - $\beta\colon p_1|p_2 \Rightarrow p_2|p_1$
  - $\iota\colon 0|p \Rightarrow p$
  - $\chi\colon \mathrm{send}(x, p_1, \ldots, p_n) \mid \mathrm{recv}(x, q) \mid \mathrm{comm} \Rightarrow q(``p_1", \ldots, ``p_n") \mid \mathrm{comm}$
  - $\epsilon\colon \mathrm{eval}(``p") \Rightarrow p$

- Equations:
  - $\alpha = P^3, \beta = P^2, \iota = P$ ($|$ 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

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

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

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

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 usage of a processor. [[Formal verification of billing for using resources.]]

[[Unguessable (quoted processes with no unforgeable names) vs unforgeable (quoted processes with names from the category of generators)]]

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

$$D(x) = \mathrm{recv}(x, y \mapsto \mathrm{send}(x, \mathrm{eval}(y))|\mathrm{eval}(y))$$
$$!P = \mathrm{send}(x, D(x)|P)|D(x).$$

Caires' adjunct to $|$ is an instance of our generic modal operator, where $u \triangleright v = u\langle -|-\rangle v$:

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

Th 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 [?].

10.1.1 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.

[[formula for liveness]]

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

[[formula for compile-time firewall: receives no messages except on the names "ϕ"]]

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

[[Point at paper with Sophia for security example.]]

11 Conclusion and future work

[[Greg]]

References

[1] Grätzer, George. *Universal Algebra*, Van Nostrand Co., Inc. (1968) v. [[Greg: add refs]]