Keywords
higher category theory, concurrency, message-passing, types, Curry-Howard

ABSTRACT
We present a realizability interpretation of logics as distributive laws over monads. Roughly speaking, if formulae are to denote the collection of individual computations that satisfy them, we must provide three data to specify a logic: the language of individual computations, which is typically captured by the algebras of a monad, say T, for terms; the notion of collection, such as set, or bag, or list, or ... used to gather the terms that satisfy a particular formulae, which is typically captured by some monad, C, for collection. Finally, given a distributive law, $l : TC \to CT$, we find a logic whose formulae are isomorphic to TC, the semantics of which are given in CT by l. The logic will enjoy three kinds of formulae: formulae corresponding to the structure of collections unconstrained by either requirements on term structure, or requirements on the evolution of computation; formulae corresponding to the expression of constraints on term structure; formulae corresponding to the expression of constraints on computational evolution. We present several examples of the logics so generated, including logics for monoids, the lambda calculus, the π-calculus, as well as formulae illustrating the expressive power of the logics so generated.

Higher category models of the pi-calculus

Michael Stay
Google
stay@google.com

L.G. Meredith
Biosimilarity, LLC
lgreg.meredith@biosimilarity.com

1. INTRODUCTION

So far, the picture looks like this. A term calculus has three parts:

- a BNF grammar describing the syntax of the terms
- an equivalence, called "structural equivalence" that says what variations in the syntax we ignore; for example, changing the name of a dummy variable in a lambda term ("alpha equivalence"), or the commutativity of concurrent processes.
- a reduction relation that captures what we mean by the process of computation; for example, beta reduction in lambda calculus or synchronization on a name in π-calculus.
- a congruence on reduction that tells what computational processes we consider equivalent.

We encode this information as a Lawvere 2-theory Th. We have

- one sort for the terms,
- a function symbol for each term constructor,
- rewrites for both structural equivalence and the rewrite relation, and
- relations on rewrites that say that structural equivalence rewrites are the same as the identity (e.g. we treat a lambda term as independent of the choice of dummy variable).

If reductions only occur in certain contexts, then we add function symbols to reify the contexts and use those in the rewrite rules.

If reductions only occur in certain contexts, then we add function symbols to reify the contexts and use those in the rewrite rules. We take models in Cat; the 2-category of functors, transformations and modifications is equivalent to the 2-category Model(Th) of models of the calculus.

There's a forgetful functor from Model(Th) to Cat with a left adjoint. Together they form a monad on Cat.

If we assume that our 2-categories have coproducts, then the sum of the left hand sides of the term constructors gives an endofunctor on the theory. LHS is a monad on Model(Th) by construction.

The sum of the term constructors themselves, together with T, form an algebra. The free model on no generators gives the initial algebra in the model 2-category.

Next take another 2-theory of a data type, i.e. no unidirectional 2-morphisms (in the simple version). This time we interpret it in Model(Th). Call the 2-category of functors, transformations, and modifications Collect(Th). Then there's a forgetful functor from Collect(Th) to Model(Th) with a left adjoint. Together they form the free collection monad on Model(Th).

Next we need a "distributive law" LHS(C(T)) -> C(LHS(T)) in Model(Th).

We define formulae with operations from C, spatial-behavioral types from LHS & 2-morphisms of Th. When considering normalization of data types, 2-morphisms from the data type will probably show up in the modalities.

Need a proper accounting of the interpretation functor from formulae to collections of terms, and how modalities give properties like closedness.

1.0.1 Related work
TBD

1.0.2 Organization of the rest of the paper
TBD

2. SOME MOTIVATING EXAMPLES
In this section we motivate the construction by way of a few examples familiar both to computer scientists and category theorists.

2.1 A logic for monoids
The notion of monoid captures fundamental, yet minimal structure in a wide variety of computational phenomena. Data structures, such as lists, enjoy monoidal structure; arithmetic phenomena, such as the natural numbers under addition, enjoy monoidal structure; executive or control flow phenomena, such as parallel composition of communicating processes enjoy monoidal structure. For this reason, they are widely studied and relatively well understood by the computing community.

They also play a fundamental role in category theory. If category theory is a theory of composition, and monoids expose one of the most basic notions of composition, then categorical accounts of monoids and monoidal phenomena say as much about category theory as the other way around. Thus, the fact that the free monoid, $T[G]$, on a set of n generators, say $G = \{g_i : 1 \leq i \leq n\}$, is a monad and that monads are monoid objects in a category of endofunctors says a great deal about the notions of composition at play in category theory. In particular, it illustrates the process of categorification raises the level of abstraction, revealing fundamental patterns in operators in a much broader range of phenomena.

Where computer science and category theory come together to study these phenomena we see a new kind of language arise, one with the computer scientist's emphasis on effective presentations, yet with the category theorists view to higher and higher levels of abstraction and expressiveness. In this language, we can present the basic data of the free monoid on a set of n generators in a manner that supports both computation and higher levels of abstraction.

$$\begin{array}{lr} T[G] ::= e & \text{identity} \\ \quad \mid g_i & \text{generator in G} \\ \quad \mid T[G] * T[G] & \text{composition} \end{array}$$

This sort of notation follows the computer scientist's use of grammars to present abstract syntax. Such a presentation provides a natural and compact way to present and study algebraic and computational structure. It does so in two stages. Firstly, it recursively describes a set of purely syntactic entities, monoid expressions, if you will, in terms of a set of building blocks, namely the identity, e, and the generators $g_i \in G$. Secondly, it describes a structural equivalence relation, $\equiv$, as the smallest equivalence relation on $T[G]$ satisfying

$$t * e \equiv t \equiv e * t$$
$$t_1 * (t_2 * t_3) \equiv (t_1 * t_2) * t_3$$

This presentation should also be somewhat familiar to the category theorist who is well acquainted with generators and relations style presentations of algebras. The notion of grammar generalizes the notion of generators, while the notion of structural equivalence generalizes the notion of relations.

At a higher level of abstraction, the recursive specification of $T[G]$ makes it clear that $T[G]$ computes the transitive closure of some operator T on G. Recalling that closure operators are monads, this presentation reveals the monadic structure of T immediately. Meanwhile, the structural equivalence relation can be seen as the data required to present an algebra of the monad, T, illustrating that an algebra of a monad, in this case $(T, \equiv)$, can also be a monad, in this case the free monad on n generators.

To construct a logic where a formula, say ϕ, denotes the collection of monoid expressions in $T[G]$ satisfying ϕ, we need to specify what *kind* of collection. This is a particular insistence on specificity arising from the computer scientist's desire for effective presentations. To turn specifications into computations, we need to know what kind of collection to use because different kinds of collections enjoy different constraints, which affect how we might recursively compute a particular collection from a specification of its elements.

For instance lists are sensitive to order where sets are not. Bags are sensitive to multiplicity where sets are not. In some sense, sets are the most insensitive monadically presented notion of collection, and thus are ideal for erasing detail about collectivity that might distract when bootstrapping an understanding of what it means to collect or gather things together. It may be that this insensitivity is why sets have enjoyed such a central role in the foundations of mathematics.

In this context, it is irresistible to observe that collectivity is a kind of composition, and thus, it is no accident that set theory and category theory enjoy the relationship that they do: both are presentations of mathematics as built up from a notion of composition. Set theory begins with the most basic of notions, pure collection, erasing all other details or qualities of collecting components together into a composition. Category theory generalizes the notion of composition, providing a simpler rubric for a much broader range of phenomena. To say more, however, would be too much of a digression.

Instead, let's use this discussion to motivate the selection of the set monad as our notion of collection, not only because it is quite common for the semantics of formulae to be sets, but because set's insensitivity allow us to focus on more important aspects of the construction we are describing.

- collection constraints. The first block of formulae enjoy no structural constraints, but only collection constraints. As such, they should be quite familiar. They constitute the well known relationship between boolean algebras and sets as their models.
$$\begin{aligned}[\![\mathsf{true}]\!] &= T[G]\\ [\![\neg\phi]\!] &= T[G]\backslash[\![\phi]\!]\\ [\![\phi\&\psi]\!] &= [\![\phi]\!]\cap[\![\psi]\!]\end{aligned}$$
- structural constraints. The second block of formulae enjoy no collection constraints, but only structural constraints. These will be familiar to those who have worked with separation logic, behavioral spatial logic, or linear logic.
$$\begin{aligned}[\![\mathsf{e}]\!] &= \{t\in T[G] : t\equiv e\}\\ [\![\mathsf{g}_i]\!] &= \{t\in T[G] : t\equiv g_i\}\\ [\![\phi{*}\psi]\!] &= \{t\in T[G] : t\equiv t_1 * t_2, t_1\in[\![\phi]\!], [\![\psi]\!]\}\end{aligned}$$

2.1.1 Some formulae in a logic for monoids

While this might seem a regular construction, does it yield interesting logics? Indeed it does. Already in this simple logic we can write down a 1-line formula for primality.

$$prime = \neg\mathsf{e}\&\neg(\neg\mathsf{e}\&\neg\mathsf{e})$$

It is easy to verify that

$$[\![prime]\!] = \bigcup_i [\![g_i]\!]$$

2.1.2 Varying the collection monad

TBD

2.2 A logic for lambda

TBD

2.2.1 Some formulae in a logic for lambda

TBD

2.3 A categorical rewrite

In the categorical presentation of the term calculus for monoids, we use a Lawvere 2-theory [?, ?]. We have a sort for terms, morphisms for the term constructors, rewrite rules for structural equivalence and reduction, and relations between rewrites.

In particular, we have a Lawvere 2-theory Th(Mon) with one sort T for terms; the function symbols

$$\begin{array}{ll} T[G] ::= e & e : 1\to T\\ \quad\mid g_i & g_i : 1\to T\\ \quad\mid T[G] * T[G] & m : T\times T\to T,\end{array}$$

corresponding to the term constructors generate a $(G+1)$-pointed magma.

Structural equivalence is interpreted as bi-directional morphisms $T\to T$. Most programming languages have free monoids built into the language in the form of lists; some though, like Scheme and Haskell, use an operator called "cons" to build lists out of ordered pairs, exactly as we are doing with m. In order to compare two words of a monoid using the constructors above, we have to compute a normal form.

We arbitrarily choose to have the parentheses at the right and no occurrences of $e()$ unless the normal form is $e()$ itself. Our rewrite rules are

$$\begin{array}{rl} a: & ((x\ y)\ z)\Rightarrow(x\ (y\ z))\\ l: & (e()\ x)\Rightarrow x\\ r: & (x\ e())\Rightarrow x\end{array}$$

Finally, we impose relations on the rewrite rules. The rewrites a, l, and r satisfy the pentagon and triangle equations from a monoidal category.

We can take models in any 2-category with finite products; if we take models in the category Set thought of as a 2-category, then the rewrite rules are identities, thus invertible, and we get monoids generated by at least G. If instead we take models in Cat, we get a category T of words in a $(G+1)$-pointed magma with rewrites between them. Once we begin to consider words with rewrites between them, we call the words "processes".

2.4 A logic for π-calculus

2.4.1 Our running process calculus

2.4.2 Syntax

$$\begin{array}{lr} P ::= 0 & \text{stopped process} \\ \quad \mid x?(y_1, \dots, y_n) \Rightarrow P & \text{input} \\ \quad \mid x!(y_1, \dots, y_n) & \text{output} \\ \quad \mid (\mathsf{new}\ x)P & \text{new channel} \\ \quad \mid P \mid Q & \text{parallel} \end{array}$$

Due to space limitations we do not treat replication, $!P$.

2.4.3 Free and bound names

$$\begin{aligned} &\mathcal{FN}(0) \coloneqq \emptyset \\ &\mathcal{FN}(x?(y_1, \dots, y_n) \Rightarrow P) \coloneqq \\ &\quad \{x\} \cup (\mathcal{FN}(P) \setminus \{y_1, \dots y_n\}) \\ &\mathcal{FN}(x!(y_1, \dots, y_n)) \coloneqq \{x, y_1, \dots, y_n\} \\ &\mathcal{FN}((\mathsf{new}\ x)P) \coloneqq \mathcal{FN}(P) \setminus \{x\} \\ &\mathcal{FN}(P \mid Q) \coloneqq \mathcal{FN}(P) \cup \mathcal{FN}(Q) \end{aligned}$$

An occurrence of x in a process P is *bound* if it is not free. The set of names occurring in a process (bound or free) is denoted by $\mathcal{N}(P)$.

2.4.4 Structural congruence

The *structural congruence* of processes, noted $\equiv$, is the least congruence containing α-equivalence, $\equiv_\alpha$, making $(P, |, 0)$ into commutative monoids and satisfying

$$(\mathsf{new}\ x)(\mathsf{new}\ x)P \equiv (\mathsf{new}\ x)P$$

$$(\mathsf{new}\ x)(\mathsf{new}\ y)P \equiv (\mathsf{new}\ y)(\mathsf{new}\ x)P$$

$$((\mathsf{new}\ x)P) \mid Q \equiv (\mathsf{new}\ x)(P \mid Q)$$

2.4.5 Operational Semantics

$$\frac{|\vec{y}| = |\vec{z}|}{x?(\vec{y}) \Rightarrow P \mid x!(\vec{z}) \to P\{\vec{z}/\vec{y}\}} \quad (\textsc{Comm})$$

In addition, we have the following context rules:

$$\frac{P \to P'}{P \mid Q \to P' \mid Q} \quad (\textsc{Par})$$

$$\frac{P \to P'}{(\mathsf{new}\ x)P \to (\mathsf{new}\ x)P'} \quad (\textsc{New})$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \quad (\textsc{Equiv})$$

2.4.6 Bisimulation

Definition 2.4.1. *An* observation relation, $\downarrow$ *is the smallest relation satisfying the rules below.*

$$\frac{}{x!(\vec{y}) \downarrow x} \quad (\textsc{Out-barb})$$

$$\frac{P \downarrow x \ \text{or}\ Q \downarrow x}{P \mid Q \downarrow x} \quad (\textsc{Par-barb})$$

$$\frac{P \downarrow x,\ x \neq u}{(\mathsf{new}\ u)P \downarrow x} \quad (\textsc{New-barb})$$

Notice that $x?(y) \Rightarrow P$ has no barb. Indeed, in π-calculus as well as other asynchronous calculi, an observer has no direct means to detect if a sent message has been received or not.

Definition 2.4.2. *An* barbed bisimulation, *is a symmetric binary relation $\mathcal{S}$ between agents such that $P\ \mathcal{S}\ Q$ implies:*

1. *If $P \to P'$ then $Q \to Q'$ and $P'\ \mathcal{S}\ Q'$.*
2. *If $P \downarrow x$, then $Q \downarrow x$.*

P is barbed bisimilar to Q, written $P \dot{\approx} Q$, if $P\ \mathcal{S}\ Q$ for some barbed bisimulation $\mathcal{S}$.

3. LOGIC AS DISTRIBUTIVE LAW

TBD

4. CONCLUSIONS AND FUTURE WORK

TBD

Acknowledgments. TBD