LADL
or

How to derive a logic from a term calculus, a collection, and a distributive law

Michael Stay (Pyrofex)
Lucius Gregory Meredith (RChain)

Problem

While at Microsoft, Greg was exploring the idea of putting pi calculus on a chip. The engine was small enough that he could tile the chip with copies of it. Now he had to reason about which processes were running on which chips. He thought about using the ambient calculus for that, but quickly realized that he had no type system for reasoning about the whole system.

Another example: HTML contains CSS and JavaScript as DSLs. How can we use a type system to reason about the operational semantics of a web browser?

Naive version

Given finitary monads Term, Coll on Poset, we get a monad for structural types

$$\text{Type} = \text{Term} + \text{Coll}.$$

Given a distributive law

$$\delta\colon \text{Term} \circ \text{Coll} \Rightarrow \text{Coll} \circ \text{Term}$$

we get an "interpretation" natural transformation

$$[\![-]\!]\colon \text{Type} \Rightarrow \text{Coll} \circ \text{Term}$$

that maps a type to the collection of values of that type. Here's how...

Naive version

In general, coproducts of monads are hard to form, but every finitary monad M corresponds to a Lawvere theory Th(M). A presentation of a Lawvere theory has

- a sort T
- a set of function symbols $f_i : T^{n_i} \to T$
- a set of equations between compositions of function symbols

When the sets of function symbols and equations are finite, taking the coproduct of two Lawvere theories is just identifying the sorts and taking the unions of the sets of function symbols and equations.

Naive version

Given two such Lawvere theories Th(Term) and Th(Coll)

(Term + Coll) X = Type X

=

X
+ Term X + Coll X
+ Term Term X + Term Coll X
  + Coll Term X + Coll Coll X
+ Term Term Term X + ...

Naive version

The monad unit lets us put a copy of Coll on the left and Term on the right.

The monad join lets us eliminate duplicates.

The distributive law δ: Term ○ Coll ⇒ Coll ○ Term shifts all the Colls to the left of the Terms.

Taken together, we get ⟦-⟧: Type ⇒ Coll ○ Term.

Naive version: example

For example, let Th(Term) be the theory of SKI combinators:

1. T
2. S, K, I: $1 \to T$
   (- -): $T^2 \to T$
3. (((S x) y) z) = ((x z) (y z))
   ((K x) y) = x
   (I x) = x

Naive version: example

Let Th(Coll) be the theory of idempotent commutative monoids (aka countable sets):

1. T
2. $\cup : T^2 \to T$
   $\{\} : 1 \to T$
3. commutativity: $A \cup B = B \cup A$
   associativity: $(A \cup B) \cup C = A \cup B) \cup C$
   unit laws: $\{\} \cup A = A$
   idempotence: $A \cup A = A$

Naive version: example

A structural type is a term like

$$(S\ \{K,\ (S\ \{I,\ K\})\})$$

that alternates between the term language and the collection language. Distributing in the obvious way gives the interpretation

$$[\![(S\ \{K,\ (S\ \{I,\ K\})\})]\!] = \{(S\ K),\ (S\ (S\ I)),\ (S\ (S\ K))\},$$

so the type has three inhabitants.

Naive version: example

Not all Term / Coll pairs allow for the existence of a distributive law. SKI doesn't work with lists because

$$((K\ S)\ [S, K, I])$$

reduces to [S] one way, but distributes to

$$[((K\ S)\ S), ((K\ S)\ K), ((K\ S)\ I)]$$

which reduces to

$$[S, S, S] \neq [S].$$

Using linear combinators (e.g. BCI) works with multisets (commutative lists). The BI fragment doesn't braid items, so it works with lists.

Enrichment over Gph (reflexive directed multigraphs)

For example, let Th(Term) be the Gph-enriched theory of WHNF SKI combinators. We model a theory as a graph equipped with graph homs and graph transforms.

1. T
2. $S, K, I: 1 \to T$
   $(- -): T^2 \to T$
   $R: T \to T$
3. $R(x\ y) = (Rx\ y)$
4. $\sigma: R(((S\ x)\ y)\ z) \Rightarrow R((x\ z)\ (y\ z))$
   $\kappa: R((K\ x)\ y) \Rightarrow Rx$
   $\iota: R(I\ x) \Rightarrow Rx$

Proof theory

Sequent calculus:

- entailment = Gph-enriched profunctor, namely the hom functor in Th(Type)
- inference = Gph-enriched (di)natural transformation
- proof rewrite = Gph-enriched "modification"

Very coarse. We would like to use elements of T as types, so we switch to the coslice category 1/Th(Type). The underlying poset gives the subtyping relation. Can also use Melliès & Zeilberger's "Functors are Type Refinement Systems" with the forgetful functor U: 1/Th(Type) → Th(Type) to describe how types refine sorts.

Details of term assignment still need working out.

Modalities

Structural types now include behavioral information, since we can talk about edges.

Modalities:

- ♢X = "those states in the graph that *may* reach a term of type X in one step"
- □X = "those states in the graph that *must* reach a term of type X in one step"

$$\neg\Diamond\, \exists\, x.(x!(*secret) \mid P)$$

Modalities

Generalized necessity modality

- $X[R,K]Y = \{t \mid \exists u \in X, v \in Y, \rho : R[K[t, u]] \to R[v]\}$
  "Those terms t that when you put them in a certain context with something of type X, it must reduce to something of type Y."

If K[-, -] is application, X[R,K]Y is the usual arrow type constructor $X \to Y$, i.e. "Those terms t that when you apply them to an X you get a Y."

If K[-, -] is parallel execution in the pi calculus, we get Caires' rely-guarantee modality $X \triangleright Y$, i.e. "Those terms t that when you run them in parallel with an X you get a Y."

Extensional and intensional predicates

When Coll is not the covariant power object monad on a topos, we lose the equivalence between (Coll X) and (X → Coll 1): when Coll is the monad for lists,

- Coll X is lists of Xs, while
- X → Coll 1 assigns to each x in X a natural number.

So we have to consider extensional (Coll X) and intensional (X -> Coll 1) versions of predicates. One is a collection, the other a filter, and filters become more like "bandpass filters", since e.g. for lists they can return 2 or 10 instead of just 0, 1.

Mitchell-Benabou / Wadler

Language for the Kleisli category of a strong monad:

$$x.[\ h(x, y, z) \mid y \leftarrow f(x)\ ;\ z \leftarrow g(x, y)\ ]$$

means

$$x \mapsto \text{join map } (y \mapsto \text{join map } (z \mapsto \text{unit } (h\ x\ y\ z))\ (g\ x\ y))\ (f\ x).$$

We can think of the variable x as a named hole in a term constructor, but the rest of the variables appear internally in the expression. This is new, nominal syntax (Gabbay & Pitts, Clouston) for collections that only appears on the LHS and needs an interpretation coherent with the naive part.

Non-uniqueness of certain important concepts

- What's equality?
  - =: Coll(X × X)

    An explicit collection of (item, item) pairs (i.e. of witnesses)?
    - E.g. for lists, what if a pair occurs more than once in the
  - =: X × X → Coll 1

    An assignment of a "truth value" to each (item, item) pair? If so, what truth value? E.g. for lists
    - 0/1?
    - Can some x's be "more equal" than others?
    - something else?

Non-uniqueness of certain important concepts

- What's inhabitation?
  - $\in : \mathrm{Coll}(X \times \mathrm{Coll}\ X)$
    An explicit collection of (item, collection containing it) pairs (i.e. of witnesses), or
  - $\in : X \times \mathrm{Coll}\ X \to \mathrm{Coll}\ 1$
    An assignment of a "truth value" to each (item, collection containing it) pair? If so, what truth value? E.g. for lists
    - 0/1?
    - 0/the initial position of the item in the list?
    - something else?
  - In a topos, $x \in \eta(x)$. How do we generalize this? When Coll is lists and we're using an intensional inhabitation, do we say
    - $x \in \eta(x) > 0$ or
    - $x \in \eta(x) = 1$?

Non-uniqueness of certain important concepts

- $\top$: Coll X
  - Is there a collection containing all other collections as subcollections?
- $\top$: X → Coll 1
  - E.g. for lists, are certain x's "more belonging" to $\top$ than others?

Non-uniqueness of certain important concepts

- $\bot : \text{Coll } X$
  - Do all collections allow for the existence of an empty collection?
- $\bot : X \to \text{Coll } 1$
  - Does Coll 1 always allow for a "false" value?

Quantification

When Coll 1 has the structure of a rig, we can do quantification. Note extra parameter since we don't have an extensional $\top$ in general.

$\sum, \prod$: Coll Coll 1 $\to$ Coll 1

$\exists_f$: Coll X × (X $\to$ Coll 1) $\to$ (Y $\to$ Coll 1)
(xs, g) $\mapsto$ y $\mapsto \sum$ [g(x) | x $\leftarrow$ xs; $\star \leftarrow$ f(y)]

$\forall_f$: Coll X × (X $\to$ Coll 1) $\to$ (Y $\to$ Coll 1)
(xs, g) $\mapsto$ y $\mapsto \prod$ [g(x) | x $\leftarrow$ xs; $\star \leftarrow$ f(y)]

Multisorted theories: Algebraic data types

Programmers don't use the Lawvere theory of free monoids for describing lists. Usually it's the theory of an X action:

1. X, T
2. nil: $1 \to T$
   cons: $X \times T \to T$

"Summing" the theories gets more complicated, but also allows enriching over 1/Th(Meaning) so that, e.g. entailment gives a Collection of witnesses instead of a subgraph of witnesses.

Nominal multisorted finite-limits enriched theories

- Nominal: many languages have binders
- Multisorteds: lots of components to the state, e.g. stack, heap, registers, program counter, task queue, etc.
- Finite limits: would like to use e.g. pullbacks in our theories to describe things like categories.

Projects like K Framework (kframework.org) exist to give operational semantics for real programming languages for formal analysis; K theories are essentially very powerful versions of Lawvere theories.

We'd like to be able to take arbitrary compositions of languages, pick a collection and a distributive law, and derive a type system. Lots of work left to do.

We're hiring!

jobs@pyrofex.net