MeTTa Architecture Proposal

1 Introduction

1.1 Early MeTTa implementations

1.2 Problems

1.2.1 Leaking Secrets

1.2.2 All paths

1.3 GSLTs as a way forward

1.3.1 Well-defined semantics

1.3.2 Compiles to Rholang

1.3.3 Rholang is optimal

2 Theories of Operational Semantics

Our approach to operational semantics is to define a category Th whose generating objects are *sorts*—the basic atoms out of which the state of the machine is assembled—and whose generating morphisms are term constructors that produce terms of those sorts.

Bill Lawvere pioneered the use of categories to capture grammars modulo equations for structural congruence, so such categories are often called *Lawvere theories*. Lawvere proved that his theories correspond to finitary monads. Our theories are Lawvere theories equipped with extra structure and stuff; specifically, they are *graph-structured magmal multi-sorted lambda theories*, or GSLTs for short.

The extra structure and stuff allow us to talk about reductions between terms, bind variables, and define modal types.

2.1 Multi-sorted

Lawvere only considered categories with one generating sort. We need at least two generating sorts, one for terms representing processes, and another for terms representing rewrites between processes. Todd Trimble gave a definition of multisorted Lawvere theories on the nLab and proved a similar monadicity theorem.

2.2 Graph-structured

We distinguish one of these sorts, say P, as the sort of complete machine states, which we call processes. Rewrites are represented as a different sort R equipped with source and target maps from R to P. Let $\mathsf{Th}(\mathsf{Gph})$ be the category with two objects and two parallel morphisms between them. A graph-structured theory Th is one equipped with a functor from $\mathsf{Th}(\mathsf{Gph})$ to Th.

2.3 Magmal

We restrict our attention to those programming languages where the source of any rewrite rule is an occurrence of a binary process constructor $\odot$. For example:

- In lambda calculus, beta reduction is of the form $((\lambda x.T)U) \to T[U/x]$, whose source is an application of one process to another.
- In pi calculus, the comm rule is of the form $\text{for}(y \leftarrow x)P \mid x!z \to P[z/y]$, whose source is a juxtaposition of two processes.

Consider the type of terms $\langle\odot([], A)\rangle B$ that when placed into a term context $\odot([], x)$, where $x : A$, may reduce to a term of type B. In lambda calculus, this corresponds to the arrow type $A \to B$. In pi calculus, this is a *possibility* modal type; the dual *necessity* modal type is Caires' rely-guarantee type $A \triangleright B$ [citation]. If B can depend on x, these are dependent product types.

2.4 Lambda theories

Lawvere's theories were cartesian categories; that is, the theories had finite products. While it is possible to define the machinery of bound variables and substitution in a Lawvere theory, it is a hassle and not particularly interesting. Instead, we use cartesian closed categories, which handle bound variables and substitution automatically.

3 Presentation of finitely generated GSLTs

A presentation of a GSLT consists of:

- A finite set of **generating sorts**, including two distinguished sorts P (processes) and R (rewrites) from the graph structure.
- A choice of a distinguished binary morphism $\odot : X \times Y \to P$, called the **interaction**, for some pair of sorts X, Y.
- A finite set of **generating morphisms**, including the two built-in **graph structure morphisms** $s, t : R \to P$. Generating morphisms with codomain P are **term constructors**, and those with codomain R are **rewrite constructors**:
$$r : \prod_i Z_i \to R$$

If the codomain does not contain a factor of R, it must factor through the interaction:

$$\exists f : \prod_i Z_i \to X \times Y. \quad \odot \circ f = s \circ r$$

- A finite set of **equations** between morphisms. We often write $r : A \to B$ as shorthand for $s(r(z_i \ldots)) = A$, $t(r(z_i \ldots)) = B$.

Rewrite term constructors usually come in two flavors:

- **Top-level rewrites** whose codomain does not contain a factor of R.
- **In-context rewrites** whose codomain contains a factor of R.

For example, here is a GSLT for the untyped lambda calculus with head normal form evaluation:

- **No generating sorts**
- **Generating morphisms:**
  - Graph structure morphisms: $s, t : R \to P$
  - Term constructors:
    - $\text{App} : P \times P \to P$
    - $\text{Lam} : (P \to P) \to P$
  - Rewrite constructors:
    - $\text{Beta} : (P \to P) \times P \to R$
    - $\text{Head} : R \times P \to R$
- **Interaction:** App
- **Equations:**
  - $\text{Beta}(A, B) : \text{App}(\text{Lam}(A), B) \to ev(A, B)$ (top-level)
  - $\text{Head}(A, B) : \text{App}(s(A), B) \to \text{App}(t(A), B)$ (in-context)

4 Generating typed GSLTs from untyped ones

Hypercube functor.

5 GSLT to Rholang Compilation

The general idea is to send the current state on a channel (e.g., @0) and then use:

```
Interpreter = Σ for(LHS_Pattern <- @0) { 0!(RHS) | Interpreter }
```

To avoid infinitely large interpreters, we enumerate possible left-hand sides:

```
Interpreter = PatternGen(0)
PatternGen(n) = for(LHS_Pattern(n) <- @0) { @0!(RHS(n)) | Interpreter } + PatternGen(n+1)
```

A smarter enumeration would use:

```
Interpreter = for(state <- @0) { PatternGen'(0, state) | @1!(state) }
PatternGen'(n, state) = for(LHS_Pattern'(n, state) <- @1) { @0!(RHS(n)) | Interpreter } + Pa
```

6 RSpace

The Rholang interpreter uses a very efficient data structure called **RSpace** for storing continuations and matching sends with receives.

6.1 RSpace on Mork

foo