Graph Model for the untyped lambda calculus and for Verse

November 3, 2023

This document describes a denotational semantics for a based on a "graph" model.

1 Part one: untyped lambda calculus

The file `simple/model.v` contains a sketch of this semantics for the call-by-value untyped lambda calculus.

This semantics interprets lambda calculus terms as a graph: a set of input-output pairs, each pair written $w \mapsto v$, where w and v are elements of some domain $\mathcal{W}$.

The graph semantics also includes a special term $\circ$ as part of this set, to signal function values. See below.

1.1 Examples

Here are some examples.

- The graph of the identity function $\lambda x.x$ looks like this:

$$\{w \mapsto w \mid w \in \mathcal{W}\} \cup \{\circ\}$$

  This graph maps all inputs w to w.

- If we have a nonterminating expression, like ω, i.e. $(\lambda x.x)(\lambda x.x)$, then its denotation is the emptyset. It is not a value and we cannot apply it to any arguments to get a results.

$$\{\}$$

- The denotation of a function that takes an argument and then diverges i.e. $\lambda x.\omega$ is the special term only.

$$\{\circ\}$$

The term $\circ$ marks that this is a value, but there are still no mappings in the graph. This term lets us distinguish the denotation of a diverging expression from that of a value, even if we cannot use the value.

- The denotation of K, i.e. $\lambda x.\lambda y.x$ is

$$\{w_1 \mapsto \{w_2 \mapsto w_1 \mid w_2 \in \mathcal{W}\} \; \cup \{\circ\} \mid w_1 \in \mathcal{W}\} \cup \{\circ\}$$

  but we will see below that this definition could also be

$$\{w_1 \mapsto (w_2 \mapsto w_1) \mid w_1, w_2 \in \mathcal{W}\} \cup \{\circ\}$$

  because this set is extensionally equivalent. (SCW: need to define this equivalence).

1.2 Representing the domain

What is this domain $\mathcal{W}$? Informally, we would like it to be the powerset of all mappings (plus the trivial term $\circ$):

$$\mathcal{W} = \mathscr{P}(\{w_1 \mapsto w_2 \mid w_1, w_2 \in \mathcal{W}\} \cup \circ)$$

But that is not a well founded definition.

And how do we represent this set in a proof assistant?

We can represent the powerset of some type by its characteristic function, i.e. a proposition that tells us whether the value is in the set.

```
Definition P (A : Type) := A -> Prop.
```

However, this definition does not give us an *inductive* or finite representation of values. For that we need to consider only finite sets of values. (We'll use the type constructor `fset A` to refer to a finite set containing elements of type `A`). This type can be inductively defined and injected into the non-inductive variant `P A`. Using this finite set type, we can build up our representation out of an inductive representation of a mapping (`v_map`) and trivial term (`v_fun`):

```
Inductive Value : Type :=
  | v_map : fset Value -> Value -> Value
  | v_fun : Value.
```

And then represent lambda calculus expressions to a potentially infinite set of these terms.

```
Definition W := P Value.
```

As an example, let's consider the identity function. The semantics of this function is:

```
Definition idset : P Value :=
  fun t => match t with
          | v_fun => True
          | v_map INPUT OUT => OUT ∈ mem INPUT
        end.
```

Consider how this definition reacts with the semantic function for application:

```
Definition APPLY (D1 : P Value) (D2 : P Value) : P Value :=
  fun w ⇒ exists V, (v_map V w ∈ D1) ∧ (mem V ⊆ D2) ∧ (valid_mem V).
```

Or, equally expressed as an inductive:

```
Inductive APPLY : P Value -> P Value -> (Value -> Prop) :=
  | BETA : forall D1 D2 w V,
     (v_map V w) ∈ D1 ->
     (mem V ⊆ D2) -> valid_mem V ->
     APPLY D1 D2 w.
```

2 Part two: Extending the model to verse

The files in the `verse/` subdirectory contain the semantics of a much richer language.

- Verse contains multiple types of values: not only functions, but also integers and finite lists of values.
- Because of the multiple types, there is the possibility that the meaning of a term might be a *run-time error*. For example, if we try to apply the number 3 to itself.
- Verse terms, if they don't produce an error, may also produce multiple results, using choice.