Graph Model for the untyped lambda calculus and for Verse

November 5, 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 functions 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 that cannot be applied. See below.

1.1 Examples

Here are some examples.

- The graph of the increment function $\lambda x.x + 1$ looks like

$$\{k \mapsto k + 1 \mid k \in \mathcal{N}\}$$

  This graph maps natural numbers k to their sucessors.

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

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

  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 term 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

  $$\{w_1 \mapsto \{w_2 \mapsto w_1 \mid w_2 \in \mathcal{W}\} \mid w_1 \in \mathcal{W}\}$$

  but we will see below that this definition could also be

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

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

1.2 Semantic operations: specification

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

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

But that is not a well founded definition. For this section, however, we will use this as a specification of what we want the domain to look like.

To define a semantics for the untyped lambda calculus, we need to say how domain values can be constructed and used.

For that, we will specify application and abstraction operators, $\blacksquare$ and Λ:

Specification 1.1 (APPLY)**.** For $w_1, w_2 \in \mathcal{W}$, we want

$$w_1 \blacksquare w_2 \ = \{w \mid (w_2 \mapsto w) \in w_1\}$$

Specification 1.2 (LAMBDA)**.** Given $F \in \mathcal{W} \to \mathcal{W}$ and $w_1 \in \mathcal{W}$, we want

$$\Lambda\ F = \{\circ\} \cup \{(w_1 \mapsto w_2) \mid w_2 \in F(w_1)\}$$

Specification 1.3. Forall $F \in \mathcal{W} \to \mathcal{W}$, and $w \in \mathcal{W}$, we want

$$\Lambda F \blacksquare w = F\ w$$

Now that we have apply, we can specify what it means for two elements of our domain to be equal. We want a relation that allows us to approximate the individual mappings. It shouldn't require that both sets have exactly the same graph, just that the two graphs can be used in the same way.

Specification 1.4 (Extensional Equality)**.**

$$w_1 \equiv w_2 = \mathit{inhabited}\ w_1 \leftrightarrow \mathit{inhabited}\ w_2\ \ \wedge \forall w \in \mathcal{W}, w_1 \blacksquare w \equiv w_2 \blacksquare w$$

For example, (assuming that we also have numbers in our domain), the specification above allows us to consider these two graphs as equivalent.

$$\{\{1\} \mapsto \{3,4\}\} \equiv \{\{1\} \mapsto \{3\}, \{1\} \mapsto \{4\}\}$$

1.3 Representing the domain

So how do we represent $\mathcal{W}$ 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_nat : nat -> Value
  | v_map : 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
           | _ => False
         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 ->
     (v ∈ D2) ->
     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.