Graph Model for the untyped lambda calculus and for Verse

May 16, 2024

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

1 Part one: graph model of the untyped lambda calculus with numbers

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

This semantics denotes lambda calculus functions by their *graphs*: a set of input-output pairs, each pair written $w \mapsto v$, where w and v are elements of some value domain $\mathcal{W}$. When applied to an argument that matches w, the function produces v.

This is an *extensional* semantics. The semantics of a function is defined in terms of what you can do with it.

This semantics also naturally supports *approximation*. Infinite sets can be difficult to work with sometimes, so we might want to think about finite approximations. And, it is the case that in a terminating computation, any function will be called only a finite number of times, so we can denote a function by some finite subset without changing the meaning of a computation.

We want the following to be true: if $w_1 \subseteq w_2$ then everything you can do with w_1 you can also do with w_2.

1.1 Examples

Here are some examples.

- The denotation of a number is a singleton set containing only that number.

$$\{3\}$$

- The graph of the increment function, $\lambda x.x + 1$, is an infinite set. It maps each natural number k to its sucessor.

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

- 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, such as $(\lambda x.x)(\lambda x.x)$, then its denotation is the empty set. 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 a singleton set that contains a special term, signaling function values that cannot be applied.

  $$\{\circ\}$$

  The term $\circ$ marks that this term is a value but there are no associations for this function in its 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.

1.2 Semantic operations: specification

What is this domain $\mathcal{W}$? Informally, we would like it to be the powerset of all mappings (plus natural numbers and 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. However, it is a *specification* of what we want the domain $\mathcal{W}$ to look like, so we can use it to specify operations that work with the domain.

In particular, we specify the application and abstraction operators, $\blacksquare$ and Λ as follows:

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

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

In other words, the result of an application is the set of all results in the graph when provided a (potentially overapproximating) argument.

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, the specification above allows us to consider these two graphs as equivalent.

$$\{\{1 \mapsto \{3 \mapsto 2, 4 \mapsto 5\}\} \equiv \{\{1 \mapsto (3 \mapsto 2), 1 \mapsto (4 \mapsto 5)\}\}$$

Both of these are for a function two arguments. If the first argument is anything but 1, then it diverges. If the second argument is 3 or 4, then the function returns a result. Otherwise it diverges.

```
fun x => fun y => if x == 1 then if y == 3 then 2 else if y == 4 then 5 else diverge else di
```

This identity simplifies the definition of our because it means that we don't need infinite sets in our codomain. We can just add more mappings to the toplevel infinite set.

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 a 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 a representation that uses only finite *sets* of values. (In Coq, 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 the operation `mem`.) Using this finite set type, we can build up our representation out of an inductive representation of a mapping (`v_map`), numbers (`v_nat`) and trivial term (`v_fun`):

```
Inductive Value : Type :=
  | v_nat : nat -> Value
  | v_map : nfset Value -> Value -> Value
  | v_fun : Value.
```

And then represent lambda calculus expressions using 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).
```

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.