CS655 Lecture 5 | Whirlwind Denotational Semantics | 02 February, 2006

Scribe: Wei Hu | Lecturer: Wes Weimer

1 Introduction

Learning **denotational semantics**(*DS*) is a tricky practice. Originally developed by Christopher Strachey and Dana Scott, denotational semantics is also known as Scott-Strachey semantics. Due to its mathematical approach, *DS* is sometimes called fixed-point semantics, or mathematical semantics.

In constrast to DS, operational semantics is commonly used in modern research, perhaps because as an abstract interpreter it is simpler to understand. Operational semantics is not compositional (e.g., the evaluation rule for `while b do c` does not depend solely on `b` and `c`).

Students sometimes have difficulty understanding denotational semantics and seeing where they should use it. In this class the most important things to know about DS are:

1. DS is compositional. (The **most important** part of today's lecture.)
2. DS was used frequently in the past. It is less common in modern research.
3. In DS, the meaning of something is a mathematical object.
4. $\perp$ (called *bottom*) is used to represent non-termination.
5. Fixed points and domains are used to handle `while`.

DS can be quite complicated. **Compositionality** is the most commonly-mentioned property of DS.

DS is very useful if you are studying a complex computational model. SPARK Ada was formally specified with DS. In DS, everthing can be reduced to mathematical objects, so it is handy when compare two programs in different languages.

2 Getting Started

We assign meanings to programs in DS. The distinguishable properties are:

- The meaning is a mathematical object.
- The meaning is defined compositionally, that is, the denotation of a command is only based on the meaning of its immediate sub-commands.

We use a new notation $[\![]\!]$ (called double/semantic/square brackets) to obtain the meaning. Sometimes we put a letter $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$ before $[\![]\!]$ to indicate that the denotation is for `Aexp`, `Bexp` or `Com`.

We begin by defining denotations for `Aexps` and `Bexps` inductively. Denotational meaning functions are high order functions. For example, $\mathcal{A}[\![x + 1]\!]$ yields a function that, given a state σ, looks up x in σ, adds one to it and returns the number. Denotations for expressions are total functions because they are defined for every expression and every state. A formal denotational definition can be found in the book.

3 Denotational Semantics for Commands

DS becomes more complicated when we try to denote commands and handle non-termination.

When a command c does not terminate, or *diverges*, we map it to a special element $\perp$. We can extend a function $f \in X \to X_\perp$ to $X_\perp \to X_\perp$ so that $f(\perp) = \perp$. This is called *strictness*. Intuitively strictness means that if one step of a command does not terminate, then the whole command diverges.

We start to specify DS for commands:

$$\mathcal{C}[\![\mathsf{skip}]\!]\sigma = \sigma$$
$$\mathcal{C}[\![x := e]\!]\sigma = \sigma[x := \mathcal{A}[\![e]\!]]$$
$$\mathcal{C}[\![c_1; c_2]\!]\sigma = \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)$$
$$\mathcal{C}[\![\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\sigma = \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ \mathcal{C}[\![c_1]\!]\sigma\ \mathsf{else}\ \mathcal{C}[\![c_2]\!]\sigma$$

The semantics does not care about intermediate states, so it can be handy for checking the equivalence of two commands.

Let W be $\mathcal{C}[\![\mathsf{while}\ b\ \mathsf{do}\ c]\!]$. A first attempt to model while is called the *unwinding equation*:

$$W(\sigma) = \mathsf{if}\ \mathcal{B}[\![b]\!]\sigma\ \mathsf{then}\ W(\mathcal{C}[\![c]\!]\sigma)\ \mathsf{else}\ \sigma$$

However this denotation is not compositional, contradicting our requirement for DS. A consequence of this recursive definition is that the equation does not specify W uniquely. For example, the equation for $\mathcal{C}[\![\mathsf{while\ true\ do\ skip}]\!]$ reduces to $W(\sigma) = W(\sigma)$, so every function qualifies as the denotation of the command while true do skip. Our intuition suggests that $W(\sigma) = \perp$ should be the answer, but we must develop more machinery before we can arrive at that conclusion.

Examining the unwinding equation, we can abstract it to $W(\sigma) = F(W(\sigma))$. W is then a *fixed point* of an equation.

The second attempt to model while tries to solve the equation in an operational fashion. Imagine that the resources of our machine are limited so that we can only evaluate W for k iterations. If W yields a result within k iterations, we assign this result to W_k, otherwise we claim W_k diverges. Then we say $W(\sigma) = W_k(\sigma)$ if there exists a k such that $W_k(\sigma) \neq \perp$ and $W(\sigma) = \perp$ otherwise. This solution has merit, but it is still not good enough because it is operational in nature.

The last attempt to model while involves domain theory, an ordering on possible soltuions, and least fixed points. We restricted the language to only have one variable x in order to simplify our presentation. Recall that the unwinding equation had multiple solutions, but only want a single solution: we want a solution that removes all arbitrariness and replaces it with divergence. We introduce an ordering of semantic functions, and choose the least fixed point of the equation as the denotation. A smaller function in this ordering has less information.

We note that whether a least fixed point (LFP) of $W = FW$ exists depends on the properties of F. Fortunately, for everything we do in realistic programming languages, a LFP exists. We did not cover the details of the fixed-point theorem; it suffices to know the conclusion. The key concepts included: domains, monotonic functions, continuous function, and fixed points.

Although the form of the LFP solution is elegant, it is not always easy to take the infinite limit to F^k and find a closed-form solution. This usually requires some human intelligence.

4 Summary

The high-level goals about DS are more important than memorizing the details of the proofs.

1. DS is compositional.
2. DS was used commonly in the past but is less common in modern research.
3. In DS, the meaning of a program is a mathematical object.
4. $\perp$ is used to mean non-termination.
5. Fixed points and domains are used to handle while.