Lecture 12: Denotational Semantics (Part II)

Example II: A pure LISP language (the λ-calculus)

We will define a denotational semantics for a simple functional, or applicative, language we call *pure LISP*.

This language is very similar to the mathematical calculus describing the behaviour of functions, i.e. the λ-calculus.

A program is just an expression in this language.

All variables in this language are single-assignment variables.

Step 1: Abstract Syntax

$\mathtt{E} \in$ Expression

$\mathtt{A} \in$ Atom

$\mathtt{I} \in$ Identifier

$$\begin{aligned}\mathtt{E}\ ::=\ & \mathtt{LET\ I} = \mathtt{E_1\ IN\ E_2}\ \mid \mathtt{LAMBDA\ (I)\ E}\ \mid \mathtt{E_1\ E_2} \\ & \mid \mathtt{CONS\ E_1\ E_2}\ \mid \mathtt{CAR\ E}\ \mid \mathtt{CDR\ E}\ \mid \mathtt{NIL} \\ & \mid \mathtt{I}\ \mid \mathtt{A}\ \mid (\mathtt{E})\end{aligned}$$

Example Programs

The `LAMBDA` construct represents an (anonymous) function as in LISP. For example, `LAMBDA (X) X` represents the identity function.

The construct $\mathtt{E_1\ E_2}$ represents function application. As the result of performing an application we expect the function body with all occurrences of the parameter replaced by the argument.

For example, `(LAMBDA (X) X) 1` should evaluate to 1.

The construct $\mathtt{LET\ I} = \mathtt{E_1\ IN\ E_2}$ binds the value of $\mathtt{E_1}$ to the variable `I` and returns the result of $\mathtt{E_2}$.

For example $\mathtt{LET\ X} = \mathtt{2\ IN}\ (*\ \mathtt{X\ X})$ should evaluate to 4.

Example Programs: Self Application

In this language it is possible to apply a function to itself.

For example:

```
LET Y = LAMBDA (X)(X X) IN (Y Y)          ⇒  expand Y
(LAMBDA (X) (X X)) (LAMBDA (X) (X X))     ⇒  substitute (LAMBDA (X)(X X)) for X
(LAMBDA (X) (X X)) (LAMBDA (X) (X X))     ⇒  ...
```

We have an infinite computation.

Note that a circular derivation has been achieved without defining a recursive function.

Example Programs: Recursion

We can use the same technique of self application to define a *recursion operator*:

```
Y = (LAMBDA (F) (LAMBDA (X) (F (X X)))  LAMBDA (F) (LAMBDA (X) (F (X X))))
```

With this operator we can define recursive functions such as factorial:

```
FACT = Y (LAMBDA (F) (LAMBDA (X) (COND (== X 0) 1 (* X (F (- X 1))))))
```

where the function COND is a predefined conditional, with implicit lazy evaluation on the then and else branches.

Normal Forms

An expression that cannot be reduced further is said to be in normal form.

We can think of normal forms as "fully evaluated data".

We can identify two important intermediate normal forms:

- an expression is in head normal form (HNF), if it is a function and its body cannot be reduced any further;
- an expression is in weak head normal form (WHNF), if it is a function and if the body still may be reducible;

Remember that lazy evaluation, as used in Haskell, performs computation up to WHNF, whereas eager evaluation, as used in SML, performs computation up to NF.

Basic Domain Theory

Sets can be composed forming *products* and *sums*.

The *product* of two sets R and S is written $R \times S$ and defined as follows:

$R \times S = \{(r, s) \mid r \in R \textit{ and } s \in S\}$

We use the following notation for constructing a product: (r, s).

We use the following operators to extract the components of a product:

$$\begin{aligned} \textit{fst}\ (r, s) &= r \\ \textit{snd}\ (r, s) &= s \end{aligned}$$

Basic Domain Theory (cont'd)

The *(disjoint) sum* of two sets R and S is written $R+S$ and defined as follows:

$R+S = \{(0,r) \mid r \in R\} \cup \{(1,s) \mid s \in S\}$

For each sum $R+S$ we have the following constructors:

$\text{in}R(x) = (0,x), \text{for x} \in \text{R}$

$\text{in}S(x) = (1,x), \text{for x} \in \text{S}$

and the following selector:

$$\begin{aligned} &\mathit{case}\ x\ \mathit{of}\ \{ \quad \mathit{isR}(r) \rightarrow \ldots \\ &\qquad\qquad\quad\ \mathit{isS}(s) \rightarrow \ldots \} \end{aligned}$$

Basic Domain Theory (cont'd)

We write $f : A \rightarrow B$ for a function that takes values from the set A as input and produces values from the set B as output.

$f(a)$ is the application of f to the value a.

The construct $[a/x]e$ ("a for x in e") replaces all instances of x in e with a.

We often use *lambda notation* to define functions: $f = \lambda x.e$. E.g. an increment function can be defined like this:

$$add = \lambda x.x + 1$$

The construct $[a \leftarrow b]f$ defines a function that behaves like f for all inputs except a, for which b is returned, i.e.

$$([a \leftarrow b]f)(a) = b$$
$$([a \leftarrow b]f)(a') = f(a'), \text{ if a}' \neq \text{a}$$

The Semantic Domain of Finite Lists

The following semantic domain specifies finite lists, as we have used them in the calculator example already.

Finite lists over D :

Domain : $D^* = \mathit{Unit} + D + (D \times D) + (D \times (D \times D)) + \cdots$

Operations :

$$
\begin{aligned}
&\mathit{nil} :: D^* \\
&\mathit{nil} = \mathit{inUnit}() \\
\\
&\mathit{cons} :: D \to D^* \to D^* \\
&\mathit{cons}\ d\ l = \mathit{case}\ l\ \mathit{of}\ \mathit{isUnit}() \to \mathit{inD}(d) \\
&\qquad\qquad\qquad\qquad \mathit{isD}(y) \to \mathit{inD{\times}D}(d, y) \\
&\qquad\qquad\qquad\qquad \ldots
\end{aligned}
$$

The Semantic Domain of Finite Lists (cont'd)

The operations for accessing the head and the tail of a list:

$$
\begin{aligned}
&\mathit{hd} :: D^* \to D \\
&\mathit{hd}(l) = \mathit{case}\ l\ \mathit{of}\ \mathit{isUnit}() \to \mathit{inError} \\
&\qquad\qquad\qquad \mathit{isD}(y) \to y \\
&\qquad\qquad\qquad \mathit{isD{\times}D}(y) \to \mathit{fst}(y) \\
&\qquad\qquad\qquad \ldots \\
\\
&\mathit{tl} :: D^* \to D^* \\
&\mathit{tl}(l) = \mathit{case}\ l\ \mathit{of}\ \mathit{isUnit}() \to \mathit{inUnit}() \\
&\qquad\qquad\qquad \mathit{isD}(y) \to \mathit{inUnit}() \\
&\qquad\qquad\qquad \mathit{isD{\times}D}(y) \to \mathit{snd}(y) \\
&\qquad\qquad\qquad \ldots
\end{aligned}
$$

The Semantic Domain of Finite Lists (cont'd)

Finally, an operation testing whether a list is empty:

$$
\begin{aligned}
&\mathit{null} :: D^* \to \mathbb{B} \\
&\mathit{null}(l) = \mathit{case}\ l\ \mathit{of}\ \ \begin{aligned}[t] &\mathit{isUnit}() \to \mathit{tt} \\ &\mathit{isD}(y) \to \mathit{ff} \\ &\mathit{isD}{\times}D(y) \to \mathit{ff} \\ &\ldots \end{aligned}
\end{aligned}
$$

Step 2: Semantic Domains

Atomic values :

Domain : $a \in \mathit{Atom}$

Identifiers :

Domain : $i \in \mathit{Id}$

Denotable values :

$d \in \mathit{DenotableValue} = (\mathit{Function} + \mathit{List} + \mathit{Atom} + \mathit{Error})_\perp$

$f \in \mathit{Function} = \mathit{DenotableValue} \to \mathit{DenotableValue}$

$t \in \mathit{List} = \mathit{DenotableValue}^*$

$\mathit{Error} = \mathit{Unit}$

Expressible values :

Domain : $x \in \mathit{ExpressibleValue} = \mathit{DenotableValue}$

Step 2: Semantic Domains (auxiliary)

An environment contains all bindings of values to variables (the “memory” in the execution).

Environments :

Domain : $e \in Env = Id \rightarrow DenotableValue$

Operations : $getenv :: Id \rightarrow Env \rightarrow DenotableValue$

$getenv = \lambda\, i\, .\, \lambda\, e\, .\, e\ (i)$

$setenv :: Id \rightarrow DenotableValue \rightarrow Env \rightarrow Env$

$setenv = \lambda\, i\, .\, \lambda\, d\, .\, \lambda\, e\, .\, [i \leftarrow d]e$

Step 3: Valuation Functions

The valuation functions determine the meaning of a program with the aid of an environment.

$\mathbf{E} :: \texttt{Expression} \rightarrow Env \rightarrow ExpressibleValue$

$\mathbf{E}[\![\texttt{LET I} = \texttt{E}_1 \texttt{ IN } \texttt{E}_2]\!] = \lambda\, e\, .\, \mathbf{E}[\![\texttt{E}_2]\!](setenv\ [\![\texttt{I}]\!]\ (\mathbf{E}[\![\texttt{E}_1]\!]e)\ e)$

$\mathbf{E}[\![\texttt{LAMBDA (I) E}]\!] = \lambda\, e\, .\, inFunction(\lambda\, d\, .\, \mathbf{E}[\![\texttt{E}]\!](setenv\ [\![\texttt{I}]\!]\ d\ e))$

$$
\begin{aligned}
\mathbf{E}[\![\texttt{E}_1\ \texttt{E}_2]\!] = \lambda\, e\, .\, & let\ x = \mathbf{E}[\![\texttt{E}_1]\!]e\ in \\
& case\ x\ of\ \{ \\
& \quad isFunction(f)\ \rightarrow\ f\ (\mathbf{E}[\![\texttt{E}_2]\!]e); \\
& \quad isList(f)\ \rightarrow\ inError(); \\
& \quad isAtom(f)\ \rightarrow\ inError(); \\
& \quad isError(f)\ \rightarrow\ inError()\}
\end{aligned}
$$

Step 3: Valuation Functions (cont'd)

$$
\begin{aligned}
\mathbf{E}[\![\texttt{CONS E}_1\ \texttt{E}_2]\!] = \lambda\, e\,.\, & \mathit{let}\ x = \mathbf{E}[\![\texttt{E}_2]\!] e\ \mathit{in} \\
& \mathit{case}\ x\ \mathit{of}\ \{ \\
& \quad \mathit{isFunction}(f) \ \rightarrow\ \mathit{inError}(); \\
& \quad \mathit{isList}(t) \ \rightarrow\ \mathit{inList}(\mathit{cons}\ (\mathbf{E}[\![\texttt{E}_1]\!] e)\ t); \\
& \quad \mathit{isAtom}(a) \ \rightarrow\ \mathit{inError}(); \\
& \quad \mathit{isError}() \ \rightarrow\ \mathit{inError}()\}
\end{aligned}
$$

Step 3: Valuation Functions (cont'd)

$$
\begin{aligned}
\mathbf{E}[\![\texttt{CAR E}]\!] = \lambda\, e\,.\, & \mathit{let}\ x = \mathbf{E}[\![\texttt{E}]\!] e\ \mathit{in} \\
& \mathit{case}\ x\ \mathit{of}\ \{ \\
& \quad \mathit{isFunction}(f) \ \rightarrow\ \mathit{inError}(); \\
& \quad \mathit{isList}(t) \ \rightarrow\ \mathit{if}\ (\mathit{null}\ t)\ \mathit{then}\ \mathit{inError}()\ \mathit{else}\ (\mathit{hd}\ t); \\
& \quad \mathit{isAtom}(a) \ \rightarrow\ \mathit{inError}(); \\
& \quad \mathit{isError}() \ \rightarrow\ \mathit{inError}()\}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{E}[\![\texttt{CDR E}]\!] \ = \ \lambda\, e\,.\, & \mathit{let}\ x = \mathbf{E}[\![\texttt{E}]\!] e\ \mathit{in} \\
& \mathit{case}\ x\ \mathit{of}\ \{ \\
& \quad \mathit{isFunction}(f) \ \rightarrow\ \mathit{inError}(); \\
& \quad \mathit{isList}(t) \ \rightarrow\ \mathit{if}\ (\mathit{null}\ t)\ \mathit{then}\ \mathit{inError}()\ \mathit{else}\ \mathit{inList}(\mathit{tl}\ t); \\
& \quad \mathit{isAtom}(a) \ \rightarrow\ \mathit{inError}(); \\
& \quad \mathit{isError}() \ \rightarrow\ \mathit{inError}()\}
\end{aligned}
$$

Step 3: Valuation Functions (cont'd)

$\mathbf{E}[\![\texttt{NIL}]\!] = \lambda\ e\ .\ \mathit{inList}(\mathit{nil})$

$\mathbf{E}[\![\texttt{I}]\!] = \mathit{getenv}\ [\![\texttt{I}]\!]$

$\mathbf{E}[\![\texttt{A}]\!] = \lambda\ e\ .\ \mathit{inAtom}(\mathbf{A}[\![\texttt{A}]\!])$

$\mathbf{E}[\![(\texttt{E})]\!] = \mathbf{E}[\![\texttt{E}]\!]$

Simplification

As an example for a simplification let's look again at our previous example for self application.

$$
\begin{array}{ll}
\mathbf{E}[\![\texttt{LET X} = \texttt{LAMBDA (X) (X X) IN (X X)}]\!]e_0 & = \\
\mathbf{E}[\![\texttt{(X X)}]\!]e_1,\ \text{where } e_1 = \mathit{getenv}\ [\![\texttt{X}]\!]\ (\mathbf{E}[\![\texttt{LAMBDA (X) (X X)}]\!]e_0)\ e_0 & = \\
\mathbf{E}[\![\texttt{LAMBDA (X) (X X)}]\!]e_0\ \ (\mathbf{E}[\![\texttt{X}]\!]e_1) & = \\
(\lambda\ d\ .\ \mathbf{E}[\![\texttt{(X X)}]\!](\mathit{setenv}\ [\![\texttt{X}]\!]\ d\ e_0))(\mathbf{E}[\![\texttt{X}]\!]e_1) & = \\
\mathbf{E}[\![\texttt{(X X)}]\!](\mathit{setenv}\ [\![\texttt{X}]\!]\ (\mathbf{E}[\![\texttt{LAMBDA (X) (X X)}]\!]e_0)\ e_0) & = \\
\mathbf{E}[\![\texttt{(X X)}]\!] & = \\
\ldots &
\end{array}
$$

We find the same expression as in the second line, thus the simplification will go on forever.

Summary

To accurately specify the meaning of a programming language we have to give a formal specification of its semantics.

We can choose between three approaches with different levels of abstraction:

- Operational Semantics,
- Denotational Semantics,
- Axiomatic Semantics.

In a denotational semantics a program's meaning is defined as a mathematical function mapping language constructs to semantic domains: $\mathbf{E}[\![E]\!]e$

Roadmap of defining a denotational semantics:

1. Define the *abstract syntax* of the language.
2. Specify the *semantic domains* describing the values that can be produced.
3. Specify one or more *valuation functions* mapping expressions to values.