Semantics of programming languages

Informatics 2A: Lecture 27

John Longley

School of Informatics
University of Edinburgh
`jrl@inf.ed.ac.uk`

21 November, 2011

Semantics for programming languages

We've seen that the syntax of NLs (as described by CFGs etc.) is concerned with what sentences are grammatical and what structure they have, whilst their semantics are concerned with what sentences mean.

A similar distinction can be made for programming languages. Rules associated with lexing, parsing and typechecking concern the form and structure of legal programs, but say nothing about what programs should do when you run them.

The latter is what programming language semantics is about. It thus concerns the later stages of the language processing pipeline.

Specification vs. implementation

In principle, one way to give a semantics (or 'meaning') for a programming language is to provide a working implementation of it, e.g. an interpreter or compiler for the language.

However, such an implementation will probably consist of thousands of lines of code, and so isn't very suitable as a readable definition or reference specification of the language.

The latter is what we're interested in here. In other words, we want to fill the blank in the following table:

	Specification	Implementation
Lexical structure	Regular exprs.	Lexer impl.
Grammatical structure	CFGs	Parser impl.
Execution behaviour	???	Interpreter/compiler

Kinds of semantics

We'll look at two styles of formal programming language semantics:

- Operational semantics. Typically consists of a bunch of rules for 'executing' programs given by syntax trees. Oriented towards implementations of the language. indeed, an op. sem. often gives rise immediately to a 'toy implementation'.
- Denotational semantics. Typically consists of a compositional description of the meaning of program phrases (close in spirit to what we've seen for NLs). Oriented towards mathematical reasoning about the language and about programs written in it. May be 'executable' or not.

These two styles are complementary: ideally, it's nice to have both. There are also other styles (e.g. axiomatic semantics), but we won't discuss them here.

Micro-Haskell: a crash course

In mathematics, we are used to defining functions via equations, e.g. $f(x) = x^2 + 3x + 7$.

The idea in functional programming is that programs should look as far as possible like mathematical definitions:

```
f x = x*x + 3*x + 7 ;
```

This function expects an argument `x` of integer type (let's say), and returns a result of integer type. We therefore say the type of `f` is `Integer -> Integer` ("integer to integer").

By contrast, the definition

```
g x = x*x < 3*x + 7 ;
```

returns a boolean result, so the type of `g` is `Integer -> Bool`.

Multi-argument functions

What about a function of two arguments, say `x` :: `Integer` and `y` :: `Bool` ? E.g.

```
h x y = if y then x else 0-x ;
```

Think of `h` as a function that accepts arguments one at a time. It accepts an integer and returns another function, which itself accepts a boolean and returns an integer.

So the type of `h` is `Integer -> (Bool -> Integer)`. By convention, we treat $->$ as right-associative, so we can write this just as `Integer -> Bool -> Integer`.

Note incidentally the use of `if` to create expressions rather than commands. In Java, the above if-expression could be written as

```
(y ? x : -x)
```

Typechecking in Micro-Haskell

In (Micro-)Haskell, the type of `h` is explicitly given as part of the function definition:

```
h :: Integer -> Bool -> Integer ;
h x y = if y then x else 0-x ;
```

The typechecker then checks that the expression on the RHS does indeed have type `Integer`, assuming `x` and `y` have the specified argument types `Integer` and `Bool` respectively.

Function definitions can also be recursive:

```
div :: Integer -> Integer -> Integer ;
div x y = if x<y then 0 else 1 + div (x-y) y ;
```

Here the typechecker will check that the RHS has type `Integer`, assuming that `x` and `y` have type `Integer` and also that `div` itself has the stated type.

Higher-order functions

The arguments of a function in MH can themselves be functions!

```
F :: (Integer -> Integer) -> Integer ;
F g = g 0 + g 1 + g 2 + g 3;
```

The typechecker then checks that the expression on the RHS does indeed have type `Integer`, assuming `x` and `y` have the specified argument types `Integer` and `Bool` respectively.

```
inc :: Integer -> Integer ;
inc x == x+1 ;
F inc                    -- evaluates to 10
```

In principle, the `->` constructor can be iterated to produce very complex types, e.g.

```
(((Integer->Bool)->Bool)->Integer)->Integer
```

Such monsters never arise in ordinary programs, however!

Operational semantics

We can often model the execution behaviour of programs as a series of reduction steps.

E.g. for Micro-Haskell:

$$
\begin{array}{ll}
 & \texttt{if 3+5<8 then 4 else 6*7} \\
\twoheadrightarrow & \texttt{if 8<8 then 4 else 6*7} \\
\twoheadrightarrow & \texttt{if False then 4 else 6*7} \\
\twoheadrightarrow & \texttt{6*7} \\
\twoheadrightarrow & \texttt{42}
\end{array}
$$

A (small-step) operational semantics is basically a bunch of rules for performing such reductions.

More complex example

Consider the Micro-Haskell declaration

$$\texttt{f x y = x*x + y*y ;}$$

This effectively introduces the definition

$$\texttt{f} = \lambda\texttt{x}.\lambda\texttt{y}.\ \texttt{x*x + y*y}$$

Now consider the evaluation of `f 3 4`:

$$\begin{aligned}
\texttt{f 3 4} &\twoheadrightarrow (\lambda\texttt{x}.\lambda\texttt{y}.\ \texttt{x*x + y*y})\ \texttt{3 4} \\
&\twoheadrightarrow (\lambda\texttt{y}.\ \texttt{3*3 + y*y})\ \texttt{4} \\
&\twoheadrightarrow \texttt{3*3 + 4*4} \\
&\twoheadrightarrow \texttt{9 + 4*4} \\
&\twoheadrightarrow \texttt{9 + 16} \\
&\twoheadrightarrow \texttt{25}
\end{aligned}$$

Notice that two of these steps are β-reductions!

Operational semantics for Micro-Haskell: general rules

Suppose E is a runtime environment associating a definition to each function symbol, e.g. $E(\texttt{f}) = \lambda\texttt{x}.\lambda\texttt{y}.\texttt{x*x + y*y}$.

Also let v range over variables of MH, and write $\overline{n}$ to mean the integer literal for n.

Relative to E, we can define $\twoheadrightarrow$ as follows:

- $v \twoheadrightarrow E(v) \quad$ (v a variable defined in E)
- $(\lambda v.M)N \twoheadrightarrow M[v \mapsto N] \quad$ (β-reduction)
- $\overline{m} \texttt{ + } \overline{n} \twoheadrightarrow \overline{m+n}$, and similarly for other infixes.
- $\texttt{if True then } M \texttt{ else } N \twoheadrightarrow M$
- $\texttt{if False then } M \texttt{ else } N \twoheadrightarrow N$

Continued on next slide ...

Operational semantics for Micro-Haskell (continued)

Let's say a term M is a value if it's an integer literal, a boolean literal, or a λ-abstraction. Let V range over values,

Intuition: values are terms that can't be reduced any further. We try to reduce all other terms to values.

To complete the definition of $\twoheadrightarrow$, we decree that if $M \twoheadrightarrow M'$ then:

- $MN \twoheadrightarrow M'N$
- $M \odot N \twoheadrightarrow M' \odot N$ ($\odot$ any infix symbol)
- $V \odot M \twoheadrightarrow V \odot M'$ (ditto)
- $\texttt{if } M \texttt{ then } N \texttt{ else } P \twoheadrightarrow \texttt{if } M' \texttt{ then } N \texttt{ else } P$

We then say $M \twoheadrightarrow^* V$ ("M evaluates to V") if there's a sequence

$$M \equiv M_0 \twoheadrightarrow M_1 \twoheadrightarrow \cdots \twoheadrightarrow M_r \equiv V$$

That defines the intended behaviour of Micro-Haskell programs. It's also how my little evaluator for MH works.

Operational semantics: further remarks

What happens if we encounter an expression that isn't a value but can't be reduced? E.g. `5 True`, or `(`λ`x.x)+4` ?

!!! If our original program typechecks, this can never happen !!!

Indeed, we can prove that

- if M is well-typed and $M \rightsquigarrow M'$, then M' is well-typed;
- if M is well-typed, either it's a value or it can be reduced.

That's one reason why type systems are so valuable: they can guarantee programs won't derail at runtime.

The form of operational semantics we've described works particularly smoothly for functional languages, but can also be applied to most other kinds of languages.

Denotational semantics

An operational semantics provides a kind of idealized implementation of the language in terms of symbolic rules.

That's fine, but doesn't give much 'structural' understanding. Conceptually and mathematically, more satisfying to assign meaning to (parts of) a program — in roughly the way that mathematical expressions (or indeed NL expressions) have meaning.

This is the idea behind denotational semantics: associate a denotation $[\![P]\!]$ to each program phrase P in a compositional way.

Denotational semantics for MH: first attempt

Let's try interpreting MH types by sets in an obvious way:

$$[\![\,\texttt{Integer}\,]\!] \;=\; \mathbb{Z} \qquad\qquad [\![\,\texttt{Bool}\,]\!] \;=\; \mathbb{B} \;=\; \{T, F\}$$

$$[\![\,\sigma\texttt{->}\tau\,]\!] \;=\; [\![\,\tau\,]\!]^{[\![\,\sigma\,]\!]} \quad \text{(set of all functions from } [\![\,\sigma\,]\!] \text{ to } [\![\,\tau\,]\!]\text{)}$$

A term $M :: \tau$ will receive a denotation $[\![\,M\,]\!] \in [\![\,\tau\,]\!]$. More accurately, if $M :: \tau$ is a term in the type environment $\Gamma = \langle x_1 :: \sigma_1, \ldots, x_n :: \sigma_n \rangle$, its denotation will be a function

$$[\![\,M\,]\!]_\Gamma \;:\; [\![\,\sigma_1\,]\!] \times \cdots \times [\![\,\sigma_n\,]\!] \rightarrow [\![\,\tau\,]\!]$$

We define $[\![\,M\,]\!]_\Gamma$ compositionally. E.g. writing $\vec{a}$ for $\langle a_1, \ldots, a_n \rangle$:

$$\begin{array}{rcl}
[\![\,\overline{n}\,]\!]_\Gamma & : & \vec{a} \mapsto n \\
[\![\,x_i\,]\!]_\Gamma & : & \vec{a} \mapsto a_i \\
[\![\,M\texttt{+}N\,]\!]_\Gamma & : & \vec{a} \mapsto [\![\,M\,]\!]_\Gamma(\vec{a}) + [\![\,N\,]\!]_\Gamma(\vec{a}) \\
[\![\,M\;N\,]\!]_\Gamma & : & \vec{a} \mapsto [\![\,M\,]\!]_\Gamma(\vec{a})([\![\,N\,]\!]_\Gamma(\vec{a})), \quad \text{etc.}
\end{array}$$

Denotational semantics for MH: the challenge

That works well as far as it goes. The problem comes when we try to interpret recursive definitions, e.g.

```
div  =  λx.λy.if x<y then 0 else 1 + div (x-y) y ;
```

Here we'd end up trying to define $[\![\,\texttt{div}\,]\!]$ in terms of itself!

Our simple 'set-theoretic' interpretation can't make sense of this. Need an alternative interpretation that builds in some deeper mathematical properties in order to allow 'circular definitions'. This is where it gets interesting (and where we stop . . .).
E.g. one can interpret the whole of MH using

- complete partial orders (non-executable semantics)
- game models (executable semantics)

Denotational semantics for regular expressions

Let's turn to an easier example. Recall our (meta)language of regular expressions:

$$R \;\rightarrow\; \epsilon \;\mid\; \emptyset \;\mid\; a \;\mid\; RR \;\mid\; R+R \;\mid\; R^*$$

In fact, we've already met two good den. sems. for this!

- $[\![R]\!]_1 = \mathcal{L}(R)$, the language (i.e. set of strings) defined by R.
- $[\![R]\!]_2 =$ the particular NFA for R constructed by the methods of Lecture 5.

Both of these are defined compositionally: e.g. $\mathcal{L}(R+R')$ is defined as $\mathcal{L}(R) \cup \mathcal{L}(R')$, and the standard NFA for $R+R'$ is constructed out of NFAs for R and R'. Note that:

- $[\![-]\!]_1$ is more abstract than $[\![-]\!]_2$: can have $[\![R]\!]_2 \neq [\![R']\!]_2$ but $[\![R]\!]_1 = [\![R']\!]_1$. So $[\![-]\!]_1$ is more useful for arguing that two regular expressions are 'equivalent'.
- However, $[\![-]\!]_2$ is naturally executable, while $[\![-]\!]_1$ is not.

Summary

- Formal semantics can be used to give a concise and precise reference specification for the intended behaviour of programs.
- Operational semantics is nowadays quite widely used. Denotational semantics gets quite mathematical and is at present more of a ‘research topic’.
- Operational semantics, and some kinds of denotational semantics, also offer a starting-point for building working implementations of the language.
- Denotational semantics also offers a framework for proving things about programs. E.g. if $[\![P]\!] = [\![P']\!]$, that shows that P can be replaced by P' *in any program context* without changing the program’s behaviour.
- Ideas from both op. and den. semantics have had a significant effect on the design of programming languages.