Semantics of programming languages

Informatics 2A: Lecture 28

Mary Cryan

School of Informatics
University of Edinburgh
mcryan@inf.ed.ac.uk

21 November 2018

Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.

Formal Language		Natural Language	
Program text		Text	
⇓	lexing	⇓	segmentation, tagging
Lexemes		Tagged words	
⇓	parsing	⇓	parsing
Syntax tree		Parse tree	
⇓	type checking	⇓	agreement checking
(Annotated) syntax tree		(Annotated) parse tree	
		⇓	semantics
		Logical form	

Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.

Formal Language		Natural Language	
Program text		Text	
⇓	lexing	⇓	segmentation, tagging
Lexemes		Tagged words	
⇓	parsing	⇓	parsing
Syntax tree		Parse tree	
⇓	type checking	⇓	agreement checking
(Annotated) syntax tree		(Annotated) parse tree	
⇓	semantics	⇓	semantics
Program behaviour		Logical form	

Two parallel pipelines

A large proportion of the course thus far can be organised into two parallel language processing pipelines.

Formal Language		Natural Language	
Program text		Text	
⇓	lexing	⇓	segmentation, tagging
Lexemes		Tagged words	
⇓	parsing	⇓	parsing
Syntax tree		Parse tree	
⇓	type checking	⇓	agreement checking
(Annotated) syntax tree		(Annotated) parse tree	
⇓	semantics	⇓	semantics
Program behaviour		Logical form	

Today we look at methods of specifying program behaviour.

Semantics for programming languages

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 formal 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

Semantic paradigms

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: recap

We use Micro-Haskell (recall Lecture 13 and Assignment 1) as a vehicle for introducing the methods of operational and denotational semantics.

The format of MH declarations is illustrated by:

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

This declares a function `div`, of the type specified, such that, when applied to two (non-negative) integer literals $\overline{m}$ and $\overline{n}$, the function application

$$\texttt{div}\ \overline{m}\ \overline{n}$$

returns, as result, the integer literal representing the integer division of m by n.

Micro-Haskell: recap

We use Micro-Haskell (recall Lecture 13 and Assignment 1) as a vehicle for introducing the methods of operational and denotational semantics.
The format of MH declarations is illustrated by:

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

This declares a function `div`, of the type specified, such that, when applied to two (non-negative) integer literals $\overline{m}$ and $\overline{n}$, the function application

$$\texttt{div}\ \overline{m}\ \overline{n}$$

returns, as result, the integer literal representing the integer division of m by n.
Q: How would `div 2 0` behave?

Micro-Haskell: recap

We use Micro-Haskell (recall Lecture 13 and Assignment 1) as a vehicle for introducing the methods of operational and denotational semantics.
The format of MH declarations is illustrated by:

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

This declares a function `div`, of the type specified, such that, when applied to two (non-negative) integer literals $\overline{m}$ and $\overline{n}$, the function application

$$\texttt{div}\ \overline{m}\ \overline{n}$$

returns, as result, the integer literal representing the integer division of m by n.
Q: How would `div 2 0` behave?
A: Loop indefinitely!

Semantic paradigms in the case of MH

Operational semantics:

This explains the computational process by which MH calculates the value of a function application, such as `div` $\overline{m}\ \overline{n}$.

Denotational semantics:

This defines a mathematical denotation

$$[\![\,\texttt{div}\,]\!] \in [\![\,\texttt{Integer -> Integer -> Integer}\,]\!]$$

Roughly, $[\![\,\texttt{Integer->Integer->Integer}\,]\!]$ is some set of binary functions on integers, and $[\![\,\texttt{div}\,]\!]$ is the integer-division function.

In reality, denotational semantics is more complicated than this.

Operational semantics

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

E.g. for Micro-Haskell:

$$
\begin{array}{ll}
 & \texttt{if 4+5 <= 8 then 4 else 6+7} \\
\twoheadrightarrow & \texttt{if 9 <= 8 then 4 else 6+7} \\
\twoheadrightarrow & \texttt{if False then 4 else 6+7} \\
\twoheadrightarrow & \texttt{6+7} \\
\twoheadrightarrow & \texttt{13}
\end{array}
$$

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

More complex example

Consider the Micro-Haskell declaration

```
f x y = x+y+x ;
```

This effectively introduces the definition

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

Now consider the evaluation of `f 3 4`:

$$\begin{aligned}
\texttt{f 3 4} &\twoheadrightarrow (\lambda\texttt{x}.\lambda\texttt{y}.\ \texttt{x+y+x})\ \texttt{3 4} \\
&\twoheadrightarrow (\lambda\texttt{y}.\ \texttt{3+y+3})\ \texttt{4} \\
&\twoheadrightarrow \texttt{3+4+3} \\
&\twoheadrightarrow \texttt{7 + 3} \\
&\twoheadrightarrow \texttt{10}
\end{aligned}$$

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. $E(\mathtt{f}) = \lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x+y+x}$.

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:

- $v \twoheadrightarrow E(v)$ (v a variable defined in E)
- $(\lambda v.M)N \twoheadrightarrow M[v \mapsto N]$ (β-reduction)
- $\overline{m}$ + $\overline{n} \twoheadrightarrow \overline{m+n}$, and similarly for other infixes.
- `if True then` M `else` $N \twoheadrightarrow M$
- `if False then` M `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 (roughly) how the Assignment 1 evaluator for MH works.

Longer example

Within a run-time environment that records the definition of `div`, we have:

```
    div 3 2
↠  (λx.λy. if x < y then 0 else 1 + div (x + -y) y) 3 2
↠  (λy.  if 3 < y then 0 else 1 + div (3 - y) y) 2
↠  if 3 < 2 then 0 else 1 + div (3 - 2) 2
↠  if False then 0 else 1 + div (3 - 2) 2
↠  1 + div (3 - 2) 2
↠  1 + (λx.λy. if x < y then 0 else 1 + div (x + -y) y)
               (3 - 2) 2
↠  ...
```

Exercise: Finish this off!

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 `(λx.x)+4` ?

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

Indeed, it can be proved that:

- if M can be typed, either it's a value or it can be reduced;
- if M has type t and $M \twoheadrightarrow M'$, then M' has type t.

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

The general form of operational semantics we've described is immensely flexible. It works beautifully for functional languages like MH. But it can also be adapted to most other kinds of programming language.

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, it is 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 natural way:

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

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

A closed term $M :: \tau$ will receive a denotation $[\![\,M\,]\!] \in [\![\,\tau\,]\!]$.
More generally, 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

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

We define $[\![\,M\,]\!]_\Gamma$ compositionally (just as in NL semantics).
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 ;
```

Two issues here:

1. Value of `div 2 0` is undefined. So should now include a special value $\perp$ ('undefined') in the sets $[\![$ `Integer` $]\!]$ and $[\![$ `Bool` $]\!]$.
2. The definition of $[\![$ `div` $]\!]$ will be circular: we get an equation that defines $[\![$ `div` $]\!]$ in terms of itself. How can we be sure this equation even has a solution? What if it has more than one?

To make the idea work, we need to change the way we define $[\![\sigma\texttt{->}\tau]\!]$. We should take some set of functions that is ...

- rich enough to interpret all programs in our language, but
- constrained enough that circular definitions make sense.

At this point (reluctantly) we move on to something simpler ...

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 (ϵ-)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.