UNIVERSITY OF Southampton
School of Electronics and Computer Science

COMP2212 PROGRAMMING LANGUAGE CONCEPTS

Julian Rathke and Pawel Sobocinski

INTRODUCTION TO SEMANTICS

BUT WHAT DOES IT ALL MEAN?

- In this lecture we look at the topic of Semantics of Programs.
- "Semantics" refers to the **meaning** of programs.
  - a semantics of a program is a specification of a program's runtime behaviour. That is, what values it computes, what side-effects it has etc.
  - the semantics of a **programming language** is a specification of how each language construct affects the behaviour of programs written in that language.

- Perhaps the most definitive semantics of any given programming language is simply its compiler or interpreter.
  - If you want to know how a program behaves then just run it !
- However, there are reasons we shouldn't be satisfied with this as a semantics.

WHY WE NEED FORMAL SEMANTICS

- Compilers and interpreters are not so easy to use for reasoning about behaviour. Why ?
  - not all compilers agree!
  - compilers are large programs, it is possible (and common) that they contain bugs themselves. So the meaning of programs is susceptible to compiler writer error!
  - the produced low-level code is often inscrutable. It is hard to use compiler source code to trace the source of subtle bugs in your code due to strange interpretations of language operators.
  - compilers optimise programs (allegedly in semantically safe ways) for maximum efficiency. This can disturb the structure of your code and make reasoning about it much harder.

ADVANTAGES OF FORMAL SEMANTICS

- In contrast, a formal semantics should be precise (like a compiler) but written in a formalism more amenable to analysis.
  - this could be some form of logic or some other mathematical language.
  - don't need to worry about efficiency of execution and can focus on unambiguous specification of the meaning of the language constructs.
  - can act as reference 'implementations' for a language: any valid compiler must produce results that match the semantics.
  - they can be built in compositional ways that reflect high-level program structure.

APPROACHES TO SEMANTICS

- There are three common approaches to giving semantics for programs:

- **Denotational Semantics** advocates mapping every program to some point in a mathematical structure that represents the values that the program calculates.
  - e.g. [[`if (0<1) then 0 else 1`]] = *0*

- **Operational Semantics** uses relational approaches to specify the behaviour of programs directly. Typically inductively defined relations between programs and values they produce, or states the programs can transition between are used.
  - e.g. `if (0<1) then 0 else 1` → `if (true) then 0 else 1` → `0`

- **Axiomatic Semantics** take the approach that the meaning of a program is just what properties you can prove of it using a formal logic.
  - e.g. Hoare Logic.

DENOTATIONAL SEMANTICS

- To give a denotational semantics one must first identify the **semantic domain** in to which we will map programs.
- Elements in the semantic domain represent the ‘meanings’ of programs.
  - e.g. for programs that return a positive integer, a reasonable choice of semantic domain is the natural numbers.
  - for programs that represent functions from integers to integers we would choose the set of all functions between naturals.
  - for programs that return pairs of integers we take the semantic domain to be the cartesian product of the set of naturals with itself, etc.

- Semantic domains are built by following the structure of the types of the language.
- In an ideal language, the structures on the types would make for well-known, simple mathematical structures in the semantic domain!
- This is not always the case, side-effects, loops and recursion complicate things

DENOTATIONAL SEMANTICS FOR THE TOY LANGUAGE

- Let's try write a denotational semantics for our Toy language:

```
T , U ::=  Int | Bool | T → T
E ::=  n | true | false | E < E | E + E | x |
       | if E then E else E | λ (x : T) E |
       | let (x : T) = E in E | E E
```

First, we choose the semantic domains. These will be **N** and a different two element set **B** = {*true*, *false*}. We will also make use of the function spaces between these sets.

Let's define [[T]] to be **N** when T is `Int` and **B** when T is `Bool` and define

[[T → U]] = [[T]] → [[U]]

Our aim now is to provide a function [[-]] from well-typed programs E of type T to the semantic domain [[T]] , that is …

Given $\vdash$ E : T , then [[E]] should be a value in [[T]].

INTERPRETING TYPE ENVIRONMENTS

- Of course, in trying to interpret functions we will need to interpret function bodies.
  - These may contain free variables.
  - We will need to interpret terms with possibly free variables in them.
- We need to have an environment to provide values for the variables.

- Given a term $\Gamma \vdash E : T$ then we need an interpretation [[E]] that makes use of an environment σ that maps each free variable in Γ to a value in the semantic domain. We write $[[\ E\ ]]_\sigma$ to denote this.
  - We say that σ **satisfies** Γ, written as $\sigma \models \Gamma$, if whenever $\Gamma(x) = T$ then $\sigma(x)$ is a value in [[T]]
  - We require the property that, for $\Gamma \vdash E : T$ and for all σ such that $\sigma \models \Gamma$, then $[[\ E\ ]]_\sigma : [[\ T\ ]]$

DEFINING THE DENOTATION FUNCTION FOR TOY

Let's start with the values and variables of the language and arithmetic expressions:

$[\![\, \text{true} \,]\!]\, \boldsymbol{\sigma} = \textit{true}$

$[\![\, \text{false} \,]\!]\, \boldsymbol{\sigma} = \textit{false}$

$[\![\, \text{n} \,]\!]\, \boldsymbol{\sigma} = n$ where n is the corresponding natural in $\mathbf{N}$

$[\![\, \text{x} \,]\!]\, \boldsymbol{\sigma} = v$ where $\boldsymbol{\sigma}$ maps x to v

$[\![\, \text{E} < \text{E'} \,]\!]\, \boldsymbol{\sigma} = \textit{true}$ if $[\![\, \text{E} \,]\!]\, \boldsymbol{\sigma} < [\![\, \text{E'} \,]\!]\, \boldsymbol{\sigma}$

$[\![\, \text{E} < \text{E'} \,]\!]\, \boldsymbol{\sigma} = \textit{false}$ otherwise

$[\![\, \text{E} + \text{E'} \,]\!]\, \boldsymbol{\sigma} = [\![\, \text{E} \,]\!]\, \boldsymbol{\sigma} + [\![\, \text{E'} \,]\!]\, \boldsymbol{\sigma}$

$[\![\, \text{if E then E' else E''} \,]\!]\, \boldsymbol{\sigma} = [\![\, \text{E'} \,]\!]\, \boldsymbol{\sigma}$ if $[\![\, \text{E} \,]\!]\, \boldsymbol{\sigma} = \textit{true}$

$[\![\, \text{if E then E' else E''} \,]\!]\, \boldsymbol{\sigma} = [\![\, \text{E''} \,]\!]\, \boldsymbol{\sigma}$ if $[\![\, \text{E} \,]\!]\, \boldsymbol{\sigma} = \textit{false}$

$[\![\, \lambda\, (\text{x} : \text{T})\ \text{E} \,]\!]\, \boldsymbol{\sigma} = \text{v} \mapsto [\![\, \text{E} \,]\!]\, \boldsymbol{\sigma}\, [\, \text{x} \mapsto \text{v} \,]$

$[\![\, \text{let } (\text{x} : \text{T}) = \text{E in E'} \,]\!]\, \boldsymbol{\sigma} = [\![\, \text{E'} \,]\!]\, \boldsymbol{\sigma}\, [\, \text{x} \mapsto [\![\, \text{E} \,]\!]\, \boldsymbol{\sigma} \,]$

$[\![\, \text{E}\ \ \text{E'} \,]\!]\, \boldsymbol{\sigma} = [\![\, \text{E} \,]\!]\, \boldsymbol{\sigma}(\, [\![\, \text{E'} \,]\!]\,)\, \boldsymbol{\sigma}$

$\boldsymbol{\sigma}\, [\, \text{x} \mapsto \text{v} \,]$ means update the mapping $\boldsymbol{\sigma}$ with a map from x to value v

COMMENTS ON DENOTATIONAL SEMANTICS

- A criticism one might have of denotational semantics at this point is that they don't give a very clear account of **how** the program is actually supposed to execute.
- Instead, they give a very precise and nicely compositional account of what values the program is supposed to calculate. This abstracts away all of the execution steps.
- This can be useful for modelling pure functional languages, but it can be trickier for modelling languages with, say, mutable state or concurrency.
- Modelling recursion denotationally can also be challenging - what value does a non-terminating recursive loop get mapped to ?

- A major criticism of the above denotational model of the Toy language is that there is a lot of "junk" in the model …
- The semantic domain [[Int → Int]] is **all** functions from $\mathbf{N}$ to $\mathbf{N}$ - this will include uncomputable functions. The model is "too big" in a sense.
- There is lots of research in to finding denotational models that are "just right" - this is a difficult task in general, even for small Toy languages.