INTRODUCTION TO SEMANTICS
(LECTURE 7) WHAT DOES IT TAKE TO PROVE TYPE SAFETY?

- You need to know what “well-typed” means - precisely ✅
- You need to know what “programs” means - precisely ✅
- You need to know what never “go” means - precisely ❓
- You need to know what “wrong” means - precisely

“Well-typed programs never go wrong”

This is the where the mathematics comes in!

- We give an inductively defined **typing relation** - a bit like a proof in propositional logic ✅
- We give an inductively defined **reduction relation** - i.e. how the program runs ❓
- We give a description of the **error** states of the program - these vary widely
- We use the mathematical descriptions to try prove that type safety holds for a given language

This is of course all very language dependent -
the larger the language the more work there is
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 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. $\left[\text{if } (0<1) \text{ then } 0 \text{ else } 1 \right] = 0$

  • **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 $1 \rightarrow \text{if } (\text{true}) \text{ then } 0 \text{ else } 1 \rightarrow 0$

  • **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.

• We'll look at the first two of these in a little more detail.
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:

\[ \begin{align*}
T, U & ::= \text{Int} \mid \text{Bool} \mid T \rightarrow T \\
E & ::= \ n \mid \text{true} \mid \text{false} \mid E < E \mid E + E \mid x \mid \\
& \quad \mid \text{if } E \text{ then } E \text{ else } E \mid \lambda (x : T) \ E \mid \\
& \quad \mid \text{let } (x : T) = E \text{ in } E \mid E \ E
\end{align*} \]

First, we choose the semantic domains. These will be \( \mathbb{N} \) and a different two element set \( \mathbb{B} = \{\text{true, false}\} \). We will also make use of the function spaces between these sets. Let’s define \([ [ T ] ]\) to be \( \mathbb{N} \) when \( T \) is \texttt{Int} and \( \mathbb{B} \) when \( T \) is \texttt{Bool} and define

\[ [[T \rightarrow U]] = [[T]] \rightarrow [[U]] \]

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 \( \sigma \) that maps each free variable in \( \Gamma \) to a value in the semantic domain. We write \([ [ E ] ]_{\sigma}\) to denote this.
  • We say that \( \sigma \) satisfies \( \Gamma \), 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 \( \sigma \) 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:

\[
\begin{align*}
[[ \text{true} ]] \, \sigma &= \text{true} \\
[[ \text{false} ]] \, \sigma &= \text{false} \\
[[ \text{n} ]] \, \sigma &= n \quad \text{where } n \text{ is the corresponding natural in } \mathbb{N} \\
[[ \times ]] \, \sigma &= v \quad \text{where } \sigma \text{ maps } x \text{ to } v \\
[[ E < E' ]] \, \sigma &= \text{true} \quad \text{if } [[ E ]] \, \sigma < [[ E' ]] \, \sigma \\
[[ E < E' ]] \, \sigma &= \text{false} \quad \text{otherwise} \\
[[ E + E' ]] \, \sigma &= [[ E ]] \, \sigma + [[ E' ]] \, \sigma \\
[[ \text{if } E \text{ then } E' \text{ else } E'' ]] \, \sigma &= [[ E' ]] \, \sigma \quad \text{if } [[ E ]] \, \sigma = \text{true} \\
[[ \text{if } E \text{ then } E' \text{ else } E'' ]] \, \sigma &= [[ E'' ]] \, \sigma \quad \text{if } [[ E ]] \, \sigma = \text{false} \\
[[ \lambda (x : T) \ E ]] \, \sigma &= v \mapsto [[ E ]] \, \sigma \[ x \mapsto v \] \\
[[ \text{let } (x : T) = E \text{ in } E' ]] \, \sigma &= [[ E' ]] \, \sigma \[ x \mapsto [[ E ]] \, \sigma \] \\
[[ E \ E' ]] \, \sigma &= [[ E ]] \, \sigma( [[ E' ]] ) \, \sigma
\end{align*}
\]

\( \sigma \[ x \mapsto v \] \) means update the mapping \( \sigma \) with a map from \( x \) to value \( v \)
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 \([\text{\#} \text{Int} \rightarrow \text{Int}]\) is all functions from \(\text{N}\) to \(\text{N}\) - 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.
An alternative approach to semantics is to build an inductive binary relation between terms of the language.

We call this approach **operational semantics**

There are two flavours of operational semantics: **big step** and **small step**

In big step semantics the binary relation is between terms and values. It represents the values that a term can evaluate to.

We typically write \( E \Downarrow V \) to mean program \( E \) evaluates to value \( V \).

- The meaning of a program is given by the values it can evaluate to
- Modelling the run time environment (e.g. a heap) is quite straightforward in this approach by defining the relation between run time states and values.
- The program operators are often easily specified in terms of “what they do” rather than “what they mean”.

One disadvantage of this approach is that it still doesn’t account for the effects of non-terminating programs very easily.
BIG STEP TOY SEMANTICS

- Let's give an inductive relation for the big-step semantics for the Toy language:
- The form of the relation will be \( E \Downarrow V \) “expression E evaluates to value V”

\[
\begin{align*}
\lambda(x: T) E & \Downarrow \lambda(x: T) E \\
E_1 \Downarrow n & \quad E_2 \Downarrow m & n < m \quad E_1 < E_2 \Downarrow \text{true} \\
E_1 \Downarrow n & \quad E_2 \Downarrow m & n \not< m \quad E_1 < E_2 \Downarrow \text{false} \\
E_1 \Downarrow n & \quad E_2 \Downarrow m & n + m = n' \quad E_1 + E_2 \Downarrow n' \\
E_1 \Downarrow \text{true} & \quad E_2 \Downarrow V \quad \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Downarrow V \\
E_1 \Downarrow \text{false} & \quad E_3 \Downarrow V \quad \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Downarrow V
\end{align*}
\]
Let's give an inductive relation for the big-step semantics for the Toy language:

The form of the relation will be \( E \Downarrow V \) “expression \( E \) evaluates to value \( V \)”

\[
\frac{E_1 \Downarrow V \quad E_2[V/x] \Downarrow V'}{\text{let } (x : T) = E_1 \text{ in } E_2 \Downarrow V'}
\]

We need to define what the substitution \( E[V/x] \) means exactly.

\[
\frac{E_1 \Downarrow \lambda(x : T)E'_1 \quad E_2 \Downarrow V_2 \quad E'_1[V_2/x] \Downarrow V'}{E_1E_2 \Downarrow V'}
\]

Big Step semantics still don't model the sequence of computation steps a program may take though - this can be problematic when modelling e.g. concurrency.
• In contrast, small step operational semantics are given by an inductive relation between terms representing run time states of programs.

• We typically write this as $E \rightarrow E'$

• This means, program state $E$ evaluates in ‘one’ step of evaluation to program state $E'$

• By considering the evaluation of a program step-by-step then we can see clearly how a program behaves.

• To determine whether a program calculates a given return value then we repeatedly follow the single steps of evaluation until a value is reached.

• non-termination is an activity (infinite sequence of steps) rather than failure to calculate a value.
  • useful for analysing at what point programs begin to diverge and what effects they may have during divergence.

• Let’s write a small-step semantics for the Toy language.
SMALL STEP TOY SEMANTICS

The form of this relation is \( E \rightarrow E' \) this represents a single step of computation of program state \( E \) reach the run time state \( E' \) ( which can be represented as another expression ).

\[
\begin{align*}
   n < m & \quad \Rightarrow \quad n < m \rightarrow \text{true} \\
   n \not< m & \quad \Rightarrow \quad n < m \rightarrow \text{false} \\
   E \rightarrow E' & \quad \Rightarrow \quad n < E \rightarrow n < E' \\
   E_1 \rightarrow E' & \quad \Rightarrow \quad E_1 < E_2 \rightarrow E' < E_2 \\
   n + m = n' & \quad \Rightarrow \quad n + m \rightarrow n' \\
   E \rightarrow E' & \quad \Rightarrow \quad n + E \rightarrow n + E' \\
   E_1 \rightarrow E' & \quad \Rightarrow \quad E_1 + E_2 \rightarrow E' + E_2 \\
   \text{if true then } E_2 \text{ else } E_3 \rightarrow E_2 & \quad \Rightarrow \quad \text{if false then } E_2 \text{ else } E_3 \rightarrow E_3 \\
   E_1 \rightarrow E' & \quad \Rightarrow \quad \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \rightarrow \text{if } E' \text{ then } E_2 \text{ else } E_3
\end{align*}
\]
The form of this relation is \( E \rightarrow E' \) this represents a single step of computation of program state \( E \) reach the run time state \( E' \) (which can be represented as another expression).

\[
\text{let } (x : T) = V \text{ in } E_2 \rightarrow E_2[V/x]
\]

\[
\frac{E_1 \rightarrow E'}{\text{let } (x : T) = E_1 \text{ in } E_2 \rightarrow \text{let } (x : T) = E' \text{ in } E_2}
\]

\[
\frac{\lambda(x : T)E_1 \ V \rightarrow E_1[V/x]}{\lambda(x : T)E_1 \ E_2 \rightarrow \lambda(x : T)E_1 \ E'}
\]

\[
\frac{E_2 \rightarrow E'}{E_1 \rightarrow E'}
\]

\[
\frac{E_1 \ E_2 \rightarrow E' \ E_2}{E_1 \rightarrow E'}
\]
EXAMPLE AND BIG STEP PROOF TREE

• We’ll take an example Toy program and consider how to evaluate it using big step semantics, and then with small step semantics.

• The example is:

```plaintext
let (x : Int) = 
    if (10 < 3) then 0 else (10 + 1)  
in x + 42
```

• The inductive rules in the big step semantics form a proof tree to justify the final conclusion.

• This tree is as follows

\[
\begin{align*}
10 & \Downarrow 10 & 3 & \Downarrow 3 \\
(10 < 3) & \Downarrow \text{false} \\
10 & \Downarrow 10 & 1 & \Downarrow 1 \\
(10 + 1) & \Downarrow 11 \\
11 & \Downarrow 11 & 42 & \Downarrow 42 \\
(x + 42)[11/x] & \Downarrow 53 \\
\text{let } (x : T) = & \text{ if } (10 < 3) \text{ then } 0 \text{ else } (10 + 1) \text{ in } x + 42 & \Downarrow 53
\end{align*}
\]
For our example, small step semantics requires five evaluation steps to reach the value 53. Each single step is given by a proof tree that justifies it.

1. \[
\frac{10 < 3 \rightarrow \text{false}}{
\text{if (10 < 3) then 0 else (10 + 1) } \rightarrow \text{if false then 0 else (10 + 1)}
}\]

2. \[
\frac{\text{let } (x : T) = \text{if (10 < 3) then 0 else (10 + 1) in } x + 42}{\text{let } (x : T) = \text{if false then 0 else (10 + 1) in } x + 42}
\]

3. \[
\frac{10 + 1 \rightarrow 11}{\text{let } (x : T) = \text{(10 + 1) in } x + 42 \rightarrow \text{let } (x : T) = \text{11 in } x + 42}
\]

4. \[
\text{let } (x : T) = \text{11 in } x + 42 \rightarrow (x + 42)[11/x]
\]

5. \[
(x + 42)[11/x] \rightarrow 53
\]
SMALL STEP PROOFTREES

For our example, small step semantics requires five evaluation steps to reach the value 53. Each single step is given by a proof tree that justifies it.

We sometimes write this sequence of steps without showing the proof trees.

\[
\begin{align*}
\text{let } (x : T) &= \text{if } (10 < 3) \text{ then } 0 \text{ else } (10 + 1) \text{ in } x + 42 \\
&\rightarrow \\
\text{let } (x : T) &= \text{if false then } 0 \text{ else } (10 + 1) \text{ in } x + 42 \\
&\rightarrow \\
\text{let } (x : T) &= (10 + 1) \text{ in } x + 42 \\
&\rightarrow \\
\text{let } (x : T) &= 11 \text{ in } x + 42 \\
&\rightarrow \\
11 + 42 \\
&\rightarrow \\
53
\end{align*}
\]
Relating Small Step and Big Step Semantics

• Ideally, if we have defined our semantics correctly, then we should have a strong relationship between the big step and small step semantics.
• They should specify the same behaviours - albeit in different ways.
• To formalise this we first define the following: $E \rightarrow^* E'$
• if and only if there exists a (possibly empty) sequence

$$E = E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_n = E'$$

• We can prove the following Theorem for the Toy language semantics.

**Theorem** For all $E, V$:

$$E \downarrow V \text{ if and only if } E \rightarrow^* V$$

The proof of this is a straightforward proof by induction over the big step and small step derivation trees.
NEXT LECTURE: TYPE SAFETY