COMP3161/COMP9164 Supplementary Lecture Notes
Semantics

Gabriele Keller, Liam O'Connor, Johannes Åman Pohjola

September 27, 2021

After discussing syntactic properties of languages, let us now look at the semantics of programming languages and how they can be specified using inference rules. In a programming language, we distinguish between the static semantics and the dynamic semantics.

1 Static Semantics

Static semantics includes all those semantic properties which can be checked at compile time. What these properties actually are, and to which extend a compiler is able to extract information about them depends on the programming language. In general, they are either related to the scoping or the typing rules of a language. In some languages, the type of an expression or the scope of a variable can only be determined during runtime, so this is not part of the language's static semantics. We will look at both dynamic typing and dynamic scoping later on in the course.

In our language of arithmetic expressions, typing rules are pretty pointless, since we only have a single type, and every expression which is syntactically correct is also type correct. So the only interesting static semantic feature of this language is the scoping of the variables. The expression

```
let
   x = x + 1
in
   x * x
```

is syntactically correct, but not semantically, since `x` in the subexpression `x + 1` is not bound to a value. We would like to have a set of inference rules which define a predicate *ok*, such that e *ok* holds when e contains no free variables. As we know from the definition of the abstract syntax, the expression can either be a number, a variable, an addition, a multiplication, or a let-binding. Therefore, we need an inference rule for each of these cases stating under which conditions the expression is ok. For expressions representing simple numbers, it is straight-forward, as they never contain a free variable, and we might write:

$$\frac{}{(\texttt{Num}\ \mathit{int})\ \mathit{ok}}$$

However, variables are more difficult, so we leave it for later, and look at addition instead. A term $(\texttt{Plus}\ e_1\ \ e_2)$ is ok if both e_1 and e_2 are ok:

$$\frac{t_1\ \mathit{ok} \qquad t_2\ \mathit{ok}}{(\texttt{Plus}\ t_1\ t_2)\ \mathit{ok}}$$

Obviously, multiplication works in exactly the same way. Now, let's look at let-expressions: the term $(\texttt{Let}\ e_1\ \ x.e_2)$ is ok if e_1 is ok. What about e_2, though? It may contain free occurrences of x. So, to determine is e_2 is ok, we have to somehow remember that x is bound in e_2. It seems it is not sufficient to define *ok* depending solely on the expression, but we need a way to keep track of the variables which are bound in the expression. This can be done by adding a *context* Γ, a set of assumptions, to our judgment, and write

$$\Gamma \vdash e\ ok$$

to mean e is ok under the assumptions in the context Γ. In this case, Γ will consist of assumptions written $x\ bound$, that indicates that x is in scope.

We extend the rules for numbers, addition and multiplication to include the context Γ. They are otherwise unchanged.

$$\frac{}{\Gamma \vdash (\texttt{Num}\ int)\ ok}$$

$$\frac{\Gamma \vdash t_1\ ok \qquad \Gamma \vdash t_2\ ok}{\Gamma \vdash (\texttt{Plus}\ t_1\ t_2)\ ok} \qquad \frac{\Gamma \vdash t_1\ ok \qquad \Gamma \vdash t_2\ ok}{\Gamma \vdash (\texttt{Times}\ t_1\ t_2)\ ok}$$

We can now handle expressions consisting of a single variable: the expression is ok only if the variable is in the environment. In case of a let-binding, we first check the right-hand side of the binding with the old environment, and then insert the new variable into the environment and check the body expression:

$$\frac{x\ bound \in \Gamma}{\Gamma \vdash x\ ok} \qquad \frac{\Gamma \vdash t_1\ ok \quad \Gamma, x\ bound \vdash t_2\ ok}{\Gamma \vdash (\texttt{Let}\ t_1\ x.t_2)\ ok}$$

Note that sets of assumptions are traditionally written without the $\{\}$ brackets. In the same vein, notation S, x here can be read as shorthand for the set $S \cup \{x\}$.

Initially, the environment is empty, and an arithmetic expression is ok if and only if we can derive $\vdash e\ ok$ (with an empty context, i.e. syntactic sugar for $\emptyset \vdash e\ ok$) using the rules listed above.

2 Dynamic Semantics

The semantics of a programming language connects the syntax of the language to some kind of computational model. There are different techniques to describe the semantics of programming languages: axiomatic semantics, denotational, and operational semantics. In this course, we only use operational semantics, that is, we specify how programs are being executed.

Small Step Operational Semantics, also called Structured Operational Semantic (SOS) or Transitional Semantics achieves this by defining an abstract machine and step-by-step execution of a program on this abstract machine. Big Step, Evaluation or Natural Semantics specifies the semantics of a program in terms of the results of the complete evaluation of subprograms.

2.1 Small Step or Structural Operational Semantics(SOS)

Let us start by giving the SOS for the arithmetic expression example. We first have to define a *transition system*, which can be viewed as an abstract machine together with a set of rules defining how the state of the machine changes during the evaluation of a program. To be more precise, we need to define:

- a set of states S on an abstract computing device
- a set of initial states $I \subseteq S$
- a set of final states $F \subseteq S$
- a relation $\mapsto \in S \times S$ describing the effect of a *single* evaluation step on state s

A machine can start up in any of the initial states, and the execution terminates if the machine is in a final state. That is, for $s_f \in F$, there is no $s \in S$ such that $s_f \mapsto s$.

According to this notation $s_1 \mapsto s_2$ can be read as *state s_1 evaluates to s_2 in a single execution step.* A *execution sequence* or *trace* is simply a sequence of states $s_0, s_1, s_2, \ldots, s_n$ where $s_0 \in I$ and $s_0 \mapsto s_1 \mapsto s_2 \mapsto \ldots \mapsto s_n$

We say that a execution sequence is *maximal* if there is no s_{n+1} such that $s_n \mapsto s_{n+1}$, and *complete* if s_n is a final state.

Note that, although every complete execution sequence is maximal, not every maximal sequence is complete, as there may be states for which no follow-up state exist, but which are not in F. Such a state is called a *stuck state*, and intuitively corresponds to run-time errors in a program. Obviously, stuck states should be avoided, and transition systems defined in such a way that stuck states cannot be reached from any initial state.

How should S, I, F and $\mapsto$ be defined for our arithmetic expressions? If we evaluate arithmetic expressions, we simplify them stepwise, until we end up with the result. We can define our transition system similarly.

The Set of States S we include all syntactically correct arithmetic expressions.

$$S = \{e \mid \exists\Gamma.\Gamma \vdash \ e\ ok\}$$

The Set of Initial States I then should be all expressions which can be evaluated to a integer value. These are all syntactically correct expressions without free variables:

$$I = \{e \mid \vdash \ e\ ok\}$$

The Set of Final States F as every expression should be evaluated to a number eventually, we define the set of final states

$$F = \{(\texttt{Num}\ int)\}$$

Operations The next step is to determine the operations of the abstract machine. For all the arithmetic operations like addition and multiplication, we need the corresponding "machine" operation. To evaluate let-bindings, we also have to be able to replace variables in an expression by a value, so we add substitution of variables to the set of operations our abstract machine can perform. Substitution is a reasonably complex operation requiring the traversal of the whole expression, so by assuming substitution as an operation we are pretty far from a realistic machine. We will later look at alternative definitions of abstract machines which don't use substitution.

The $\mapsto$-Relation Finally, we do have to define the $\mapsto$-relation inductively over the structure. We do not have to provide any rules for terms of the form (`Num` n), since they represent final states. We start with the evaluation of addition. Keep in mind that $\mapsto$ only describes a single evaluation step of the machine. If both arguments of `plus` are already fully evaluated, we can simply add the two values using the "machine addition":

$$\frac{}{(\texttt{Plus (Num}\ \ n\texttt{) (Num}\ \ m\texttt{))} \mapsto \texttt{(Num}\ \ (n+m)\texttt{)}}$$

What should happen if the arguments are not yet fully evaluated? We have to decide which argument to evaluate first — it does not really matter. So, we start with the leftmost:

$$\frac{e_1 \mapsto e_1'}{\texttt{(Plus}\ e_1\ e_2\texttt{)} \mapsto \texttt{(Plus}\ e_1'\ e_2\texttt{)}}$$

This step is repeated until the first argument is fully evaluated, at which point be continue with the second argument:

$$\frac{e_2 \mapsto e_2'}{\texttt{(Plus (Num}\ n\texttt{)}\ e_2\texttt{)} \mapsto \texttt{(Plus (Num}\ n\texttt{)}\ e_2'\texttt{)}}$$

Multiplication works in exactly the same way:

$$\frac{}{\texttt{(Times (Num } n\texttt{) (Num } m\texttt{))} \mapsto \texttt{Num } (n * m)}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{(Times } e_1\ e_2\texttt{)} \mapsto \texttt{(Times } e_1'\ e_2\texttt{)}}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{(Times (Num } n\texttt{) } e_2\texttt{)} \mapsto \texttt{(Times (Num } n\texttt{) } e_2'\texttt{)}}$$

Let-bindings are slightly more interesting. Again, we have to decide which order we want to evaluate the arguments in. If we evaluate the first argument (i.e., the right-hand side of the binding) first and then replace all occurrences of the variable by this value, we have the following rules:

$$\frac{e_1 \mapsto e_1'}{\texttt{(Let } e_1\ x.e_2\texttt{)} \mapsto \texttt{(Let } e_1'\ x.e_2\texttt{)}}$$

$$\frac{}{\texttt{(Let (Num } n\texttt{) } x.e_2\texttt{)} \mapsto e_2[x := \texttt{(Num } n\texttt{)}]}$$

Note that we could have decided to replace the variable immediately with the expression e_1 in e_2. If x occurs in e_2 multiple times, it means, however, that we copy the expression, and consequently have to evaluate it more than once.

Example Given the rules listed above, the evaluation of an expression `(Let (Plus (Num 5) (Num 3)) x.(Times x (Num 4)))` proceeds as follows:

```
(Let (Plus (Num 5) (Num 3)) x.(Times x (Num 4)))
```

$$\mapsto$$

```
(Let (Num 8) x. (Times x (Num 4)))
```

$$\mapsto$$

```
(Times (Num 8) (Num 4))
```

$$\mapsto$$

```
(Num 32)
```

Note that we did not give the derivation for each of the evaluation steps.

More Notation We use the relation $s_1 \overset{\star}{\mapsto} s_2$ to denote that a state s_1 evaluates to a state s_2 in zero or more steps. In other words, the relation $\overset{\star}{\mapsto}$ is the reflexive, transitive closure of $\mapsto$. That is

$$\frac{}{s \overset{\star}{\mapsto} s}$$

$$\frac{s_1 \mapsto s_2 \quad s_2 \overset{\star}{\mapsto} s_3}{s_1 \overset{\star}{\mapsto} s_3}$$

Furthermore, we write $s_1 \overset{!}{\mapsto} s_2$ to express that s_1 fully evaluates in zero or more steps to a state s_2, or more formally,

$$s \overset{!}{\mapsto} s', \text{ if } s \overset{\star}{\mapsto} s' \text{and } s' \in F$$

2.2 Big Step Semantics

Let us now look at big step semantics. Similar to the initial states in SOS, we have to define a set of *evaluable expressions* E, and a set of values V (corresponding to the final states in SOS). The values can, but do not have to, be a subset of E. Finally, we define a relation "evaluates to" $\Downarrow \, \in \, E \times V$. Note that in contrast to SOS, the relation does not say anything about the number of steps the evaluation requires.

Applied to our example, we define

- the set E of evaluable expressions to be: $\{e \mid \vdash \ e \ ok\}$
- the set V of values to be the integers.

To define $\Downarrow$, we again have to consider all possible cases for e. This time, we do need rules for $e = (\texttt{Num}(n))$:

$$\frac{}{(\texttt{Num}\ n) \Downarrow n}$$

On the other hand, we only need a single rule for each addition and multiplication. For these operators, big step semantics does not need to distinguish which of the arguments is evaluated first, since the two expressions are independent:

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (n_1 + n_2)}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{(\texttt{Times}\ e_1\ e_2) \Downarrow (n_1 \times n_2)}$$

The rules for the let-binding, however, state that e_1 has to be evaluated first, because the variable is replaced by the resulting value, and therefore we have a data dependency:

$$\frac{e_1 \Downarrow n_1 \quad e_2[x := (\texttt{Num}\ \ n_1)] \Downarrow n_2}{(\texttt{Let}\ e_1\ x.e_2) \Downarrow n_2}$$

Now consider the example expression used to demonstrate evaluation using the SOS rules. Not surprisingly, the expression evaluates to the same value using big step semantics. Here is the derivation:

$$\frac{\dfrac{(\texttt{Num 5}) \Downarrow 5 \qquad (\texttt{Num 3}) \Downarrow 3}{(\texttt{Plus (Num 5) (Num 3)}) \Downarrow 8} \qquad \dfrac{(\texttt{Num 8}) \Downarrow 8 \qquad (\texttt{Num 4}) \Downarrow 4}{(\texttt{Times (Num 8) (Num 4)}) \Downarrow 32}}{(\texttt{Let (Plus (Num 5) (Num 3)) x.(Times x (Num 4))}) \Downarrow 32}$$

The concept of an evaluation sequence does not make sense for big step semantics, as expressions evaluate in a single "step".

2.3 Denotational semantics

Denotational semantics, which is not emphasised at all in this course, is the compositional construction of a mathematical object for each form of syntax. It is typically presented as a function $[\![\cdot]\!]$ from a piece of syntax to a mathematical object representing its meaning (its *denotation*).

For example, here is a denotational semantics which maps arithmetic expressions (and an environment $E : \mathsf{String} \mapsto \mathbb{Z}$) to elements of $\mathbb{Z}$.

$$\begin{array}{lcl}
[\![\texttt{Num}\ n]\!] & = & \lambda E.\ n \\
[\![\texttt{Var}\ x]\!] & = & \lambda E.\ E(x) \\
[\![\texttt{Plus}\ e_1\ e_2]\!] & = & \lambda E.\ [\![e_1]\!]E + [\![e_2]\!]E \\
[\![\texttt{Times}\ e_1\ e_2]\!] & = & \lambda E.\ [\![e_1]\!]E \times [\![e_2]\!]E \\
[\![\texttt{Let}\ x\ e_1\ e_2]\!] & = & \lambda E.\ [\![e_2]\!]\ \ (E[x := [\![e_1]\!]E])
\end{array}$$

What kind of mathematical object your program should denote depends on the programming language, but also on how much detail you want to capture. For a sequential, imperative programming language, the denotation of a program might be a function from (initial) states to (final) states. For concurrent programming, the intermediate states are important, so we might add more detail to our denotations to capture how our program interacts with its environment.