

COMP3161/COMP9164 Supplementary Lecture Notes

Semantics

Gabriele Keller, Liam O'Connor

September 29, 2019

After discussing syntactic semantics, we will discuss static semantics. We will look at static semantics in the context of programming languages and how they relate to the static semantics of a language. We will distinguish between the static and dynamic semantics of a language.

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 extent 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 at compile time. We will look at static semantics in the context of programming languages and how they relate to the static semantics of a language.

In our language of simple expressions, we have a single type, and every expression has a single type. An interesting static semantic feature of this language is the scoping of variables.

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

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* which holds for a given expression *e* if *e* is correct in the sense that it contains no free variables at all, therefore *e* is *ok*. 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:

$$\frac{}{(\text{Num } int) \text{ ok}}$$

However, variables are more difficult, so we leave it for later, and look at addition instead: a term `(Plus e1 e2)` is *ok* if both *e*₁ and *e*₂ are *ok*:

$$\frac{t_1 \text{ ok} \quad t_2 \text{ ok}}{(\text{Plus } t_1 t_2) \text{ ok}}$$

Obviously, multiplication works in exactly the same way. Now, let's look at let-expressions: the term `(Let e1 x. e2)` is *ok* if *e*₁ is *ok*. What about *e*₂, though? It may contain free occurrences of *x*. So, to determine if *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 changing adding a *context* Γ , a set of assumptions, to our judgment, and write

$$\Gamma \vdash e \text{ ok}$$

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 do not change otherwise, for these cases, it is neither necessary to add anything to the environment nor to check the content.

$$\frac{\Gamma}{\vdash (\text{Plus } 1 \ 2)} \quad \frac{\Gamma}{\vdash (\text{Times } 1 \ 2)}$$

We can now also handle the case where the expression consists of a single variable: the expression is ok only if the variable is already 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:

$$\frac{x \text{ bound} \in \Gamma}{\Gamma \vdash t_1 \text{ ok}} \quad \frac{\Gamma \vdash t_1 \text{ ok} \quad \Gamma, x \text{ bound} \vdash t_2 \text{ ok}}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 \text{ ok}}$$

Initially, the environment $\vdash e \text{ ok}$ (with an empty context).

2 Dynamic Semantics

Semantics of a programming language connect the syntax of the language to a 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 a 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 $s_0, s_1, s_2, \dots, s_n$ where $s_0 \in I$ and $s_0 \mapsto s_1 \mapsto s_2 \mapsto \dots \mapsto s_n$.

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 *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 in some sense, it corresponds to run-time errors in a program. Obviously, stuck states should be avoided, and transition systems defined in a way that stuck states cannot be reached from any initial state.

How should S, I, F and the transition system be defined? In order to simplify them syntactically, we simplify them syntactically. We define the set of initial states I to be all syntactically correct arithmetic expressions without free variables. Similarly, we define the set of final states F to be all syntactically correct arithmetic expressions which can be evaluated to an integer value. These are all syntactically correct expressions without free variables:

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

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

The Set of Final States F we define the set of final states to be all syntactically correct arithmetic expressions which can be evaluated to an integer value. These are all syntactically correct expressions without free variables:

$$F = \{(\text{Num } i) \mid i \in \mathbb{Z}\}$$

Operations The next step is to determine the operations of the abstract machine. For the arithmetic operations like addition and multiplication, we need to define an 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 reasonable complex operation requiring the traversal of the whole expression, so by assuming substitution is an operation of the machine means that it is pretty far from a realistic machine. We will later look at alternative definition of abstract machines which do without 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 $(\text{Num } n)$, 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”:

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

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:

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 e_2) \mapsto (\text{Plus } e'_1 e_2)}$$

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

$$\frac{e_2 \mapsto e'_2}{(\text{Plus } (\text{Num } n) e_2) \mapsto (\text{Plus } (\text{Num } n) e'_2)}$$

Multiplication works in exactly the same way:

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

$$\frac{e_1 \mapsto e'_1}{(\text{Times } e_1 e_2) \mapsto (\text{Times } e'_1 e_2)}$$

<https://eduassistpro.github.io/>

Let-bindings are slightly more interesting. Again, we have to make up our mind in which order we want to evaluate the arguments. 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:

<https://eduassistpro.github.io/>

$$\frac{e_1 \mapsto e'_1}{\vdots}$$

<https://eduassistpro.github.io/>

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

Example Given the rules listed above, the evaluation of an expression $(\text{Let } (\text{Plus } 5, 3) x. (\text{Times } x, 4))$ proceeds as follows:

$(\text{Let } (\text{Plus } (\text{Num } 5) (\text{Num } 3)) x. (\text{Times } x (\text{Num } 4)))$

\mapsto

$(\text{Let } (\text{Num } 8) x. (\text{Times } x (\text{Num } 4)))$

\mapsto

$(\text{Times } (\text{Num } 8) (\text{Num } 4))$

\mapsto

$(\text{Num } 32)$

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

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

$$\frac{}{s \xrightarrow{*} s}$$

$$\frac{s_1 \mapsto s_2 \quad s_2 \xrightarrow{*} s_3}{s_1 \xrightarrow{*} s_3}$$

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

$$\xrightarrow{!} \quad \xrightarrow{*}$$

2.2 Big Step Semantics

Let us now look at the big step semantics. Similar to the initial states in SOS, we have to define a set of *evaluable expressions* E , a set of values (corresponding to the final states in SOS), which can, but do not have to be a subset of T . 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 : $e \quad e \text{ ok}$
- the set V of

To define \Downarrow , we again add rules for $e = (\text{Num } n)$:

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

On the other hand, we only need a single rule for each addition and multiplication semantics does not specify which of the arguments is evaluated first, since the evaluation of the two expressions is independent:

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

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

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:

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

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:

$$\frac{\frac{(\text{Num } 5) \Downarrow 5 \quad (\text{Num } 3) \Downarrow 3}{(\text{Plus } (\text{Num } 5) \ (\text{Num } 3)) \Downarrow 8} \quad \frac{(\text{Num } 8) \Downarrow 8 \quad (\text{Num } 4) \Downarrow 4}{(\text{Times } (\text{Num } 8) \ (\text{Num } 4)) \Downarrow 32}}{(\text{Let } (\text{Plus } (\text{Num } 5) \ (\text{Num } 3)) \ x.(\text{Times } x \ (\text{Num } 4))) \Downarrow 32}$$

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