Lecture Notes on Static Semantics

15-411: Compiler Design
Frank Pfenning

Lecture 12
October 2, 2014

1 Introduction

After lexing and parsing, a compiler will usually apply *elaboration* to translate the parse tree to a high-level intermediate form often called *abstract syntax*. Then we verify that the abstract syntax satisfies the requirements of the *static semantics*. Sometimes, there is some ambiguity whether a given condition should be enforced by the grammar, by elaboration, or while checking the static semantics. We will not be concerned with details of attribution, but how to describe and then implement various static semantic conditions. The principal properties to verify for C0 and the sublanguages discussed in this course are:

- *Initialization:* variables must be defined before they are used.
- *Proper returns:* functions that return a value must have an explicit return statement on every control flow path starting at the beginning of the function.
- *Types:* the program must be well-typed.

Type checking is frequently discussed in the literature, so we use initialization as our running example and discuss typing at the end, in Section 9.

2 Abstract Syntax

We will use a slightly restricted form of the abstract syntax in Lecture 10 on IR trees, with the addition of variable declaration with their scope. This fragment exhibits all the relevant features for the purposes of the present lecture.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & n \mid x \mid e_1 \oplus e_2 \mid e_1 \texttt{ \&\& } e_2 \\
\text{Statements} & s & ::= & \mathsf{assign}(x, e) \mid \mathsf{if}(e, s_1, s_2) \mid \mathsf{while}(e, s) \\
& & & \mid \mathsf{return}(e) \mid \mathsf{nop} \mid \mathsf{seq}(s_1, s_2) \mid \mathsf{decl}(x, \tau, s)
\end{array}
$$

3 Definition and Use

Initialization guarantees that every variable is defined before it is used. The natural way to specify this in two parts: when is a variable is defined, and when it is used. An error is signaled if we cannot show that every variable in the program is defined before it is used. As usual, this property is an *approximation* of what actual behaviors can be exhibited at runtime.

First, we define when a variable is used in an expression, written as $\mathsf{use}(e, x)$. This is entirely straightforward, since we have a clear separation of expressions and statements in our language.

$$\text{no rule for } \mathsf{use}(n, x) \qquad \frac{}{\mathsf{use}(x, x)} \qquad \text{no rule for } \mathsf{use}(y, x), y \neq x$$

$$\frac{\mathsf{use}(e_1, x)}{\mathsf{use}(e_1 \oplus e_2, x)} \qquad \frac{\mathsf{use}(e_2, x)}{\mathsf{use}(e_1 \oplus e_2, x)}$$

$$\frac{\mathsf{use}(e_1, x)}{\mathsf{use}(e_1 \texttt{ \&\& } e_2, x)} \qquad \frac{\mathsf{use}(e_2, x)}{\mathsf{use}(e_1 \texttt{ \&\& } e_2, x)}$$

We see already here that $\mathsf{use}(e, x)$ is a so-called *may*-property: x *may* be used during the evaluation of x, but it is not guaranteed to be actually used. For example, the expression $y > 0$ `&&` $x > 0$ may or may not actually use x. The expression false `&&` $x > 0$ will actually never use x, and yet we flag it as possibly being used.

This is appropriate: we would like to raise an error if there is a possibility that an unitialized variable may be used. Because determining this in general is undecidable, we need to approximate it. Our approximation essentially says that any variable occurring in an expression may be used. The rule above express this more formally.

For a language to be usable, it is important that the rules governing the static semantics are easy to understand for the programmer and have some internal coherence. While it might make sense to allow false `&&` $x > 0$ in particular, what is the general rule? Designing programming languages and their static semantics is difficult and requires a good balance of formal understanding of the properties of programming languages and programmer's intuition.

Once we have defined use for expressions, we should consider statements. Does an assignment $x = e$ use x? Our prior experience with liveness analysis for register allocation on abstract machine could would say: *only if it is used in* e. We stay consistent with this intuition and terminology and write $\mathsf{live}(s, x)$ for the judgment that x is live in s. This means the value of x is relevant to the execution of s.

Before we specify liveness, we should specify when a variable is *defined*. This is because, for example, the variable x is not live before the statement $x = 3$, because

its current value does not matter for this statement, or any subsequent statement. We write $\mathsf{def}(s, x)$ is the execution of statement s will define s. This is an example of a *must*-property: we want to be sure that whenever s executes (and completes normally, without returning from the current function or raising an exception of some form), the x has been defined.

$$\frac{}{\mathsf{def}(\mathsf{assign}(x, e), x)} \qquad \text{no rule for } \mathsf{def}(\mathsf{assign}(y, e), x), y \neq x$$

$$\frac{\mathsf{def}(s_1, x) \quad \mathsf{def}(s_2, x)}{\mathsf{def}(\mathsf{if}(e, s_1, s_2), x)} \qquad \text{no rule for } \mathsf{def}(\mathsf{while}(e, s), x)$$

The last two rules clearly illustrate that $\mathsf{def}(s, x)$ is a *must*-property: A conditional only defines a variable if is it defined along both branches, and a while loop does not define any variable (since the body may never be executed).

$$\text{no rule for } \mathsf{def}(\mathsf{nop}, x) \qquad \frac{\mathsf{def}(s_1, x)}{\mathsf{def}(\mathsf{seq}(s_1, s_2), x)} \qquad \frac{\mathsf{def}(s_2, x)}{\mathsf{def}(\mathsf{seq}(s_1, s_2), x)}$$

$$\frac{\mathsf{def}(s, x) \quad y \neq x}{\mathsf{def}(\mathsf{decl}(y, \tau, s), x)}$$

The side condition on the last rule apply because s is the scope of y. If we have already checked variable scoping, then in the particular case of C0, y could not be equal to x because that would have led to an error earlier. However, even in this case it may be less error-prone to simply check the condition even if it might be redundant.

A strange case arises for return statement. Since a return statement never completes normally, any subsequent statements are unreachable. It is therefore permissible to claim that *all* variables currently in scope have been defined. We capture this by simply stating that $\mathsf{return}(e)$ defines any variable.

$$\frac{}{\mathsf{def}(\mathsf{return}(e), x)}$$

4 Liveness

We now lift the $\mathsf{use}(e, x)$ property to statements, written as $\mathsf{live}(s, x)$ (x is live in s). Liveness is again a *may*-property.

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{assign}(y, e), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)} \qquad \frac{\mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{if}(e, s_1, s_2), x)}$$

We observe that liveness is indeed a *may*-property, since a variable is live in a conditional if is used in the condition or live in one or more of the branches. Similarly, if a variable is live in the body of a loop, it is live before because the loop body *may* be executed.

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{while}(e, s), x)} \qquad \frac{\mathsf{live}(s, x)}{\mathsf{live}(\mathsf{while}(e, s), x)}$$

$$\frac{\mathsf{use}(e, x)}{\mathsf{live}(\mathsf{return}(e), x)} \qquad \text{no rule for } \mathsf{live}(\mathsf{nop}, x) \qquad \frac{\mathsf{live}(x, s) \quad y \neq x}{\mathsf{live}(\mathsf{decl}(y, \tau, s), x)}$$

In some way the most interesting case is a sequence of statements, $\mathsf{seq}(s_1, s_2)$. If a variable is live in s_2 it is only live in the composition if it is not defined in s_1!

$$\frac{\mathsf{live}(s_1, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)} \qquad \frac{\neg\mathsf{def}(s_1, x) \quad \mathsf{live}(s_2, x)}{\mathsf{live}(\mathsf{seq}(s_1, s_2), x)}$$

5 Initialization

Given liveness, we can now say when proper initialization is violated: If a variable is live at the site of its declaration. That means that its value would be used somewhere before it is defined. Assume we have a program p and we write "s in p" if s is a statement appearing in p. The the following rule captures the general condition.

$$\frac{\mathsf{decl}(x, \tau, s) \text{ in } p \quad \mathsf{live}(s, x)}{\mathsf{error}}$$

Unlike the previous rules in the lecture, this one should be read from the premises to the conclusion. In this way it is similar to our rules for liveness from Lecture 4.

This brings out an important distinction when we try to convert the specification rules into an implementation. We have to decide if the rules should be read

from the premises to the conclusion, or from the conclusion to the premises. Sometimes, the same property can be specified in different directions. For example, we can define a predicate init which verifies that all variables are properly initialized and which works from the conclusion to the premises with the following schema.

$$\frac{}{\mathsf{init}(\mathsf{nop})} \qquad \frac{\mathsf{init}(s_1) \quad \mathsf{init}(s_2)}{\mathsf{init}(\mathsf{seq}(s_1, s_2))}$$

$$\frac{\mathsf{init}(s) \quad \neg\mathsf{live}(s, x)}{\mathsf{init}(\mathsf{decl}(x, \tau, s))} \qquad \text{(other rules omitted)}$$

The omitted rules just verify each substatement so that all declarations in the program are checked in the end.

6 From Judgments to Functions

We now focus on the special case that the inference rules are to be read bottom-up. Starting with the judgments we ultimately want to verify, consider $\mathsf{init}(s)$. When we start this, s is known and we are trying to determine if there is a deduction of $\mathsf{init}(s)$ given the rules we have put down. If there is such a deduction, we succeed. If not, we issue an error message. We can model this as a function returning a boolean, or a function returning no interesting value but raising an exception in case there the property is violated.

$$\mathsf{init} : \mathsf{stm} \to \mathsf{bool}$$

Now each of the rules becomes a case in the function definition.

$$\begin{array}{lcl} \mathsf{init}(\mathsf{nop}) & = & \top \\ \mathsf{init}(\mathsf{seq}(s_1, s_2)) & = & \mathsf{init}(s_1) \wedge \mathsf{init}(s_2) \\ \mathsf{init}(\mathsf{decl}(x, \tau, s)) & = & \mathsf{init}(s) \wedge \neg\mathsf{live}(s, x) \\ \ldots & & \end{array}$$

Here we assume a boolean constant $\top$ (for *true*) and boolean operators conjunction $A \wedge B$ and negation $\neg A$ in the functional language; later we might use disjunction $A \vee B$ and falsehood $\bot$. When we call $\mathsf{live}(s, x)$ we assume that it is a similar function.

$$\mathsf{live} : \mathsf{stm} \times \mathsf{var} \to \mathsf{bool}$$

This function is now a transcription of the rules for the live judgment. In this process we sometimes have to combine multiple rules into a single case of the function definition (as, for example, for $\mathsf{seq}(s_1, s_2)$.

$$
\begin{array}{lcl}
\mathsf{live}(\mathsf{nop}, x) & = & \bot \\
\mathsf{live}(\mathsf{seq}(s_1, s_2), x) & = & \mathsf{live}(s_1, x) \vee (\neg \mathsf{def}(s_1, x) \wedge \mathsf{live}(s_2, x)) \\
\mathsf{live}(\mathsf{decl}(y, \tau, s), x) & = & y \neq x \wedge \mathsf{live}(x, s) \\
\ldots
\end{array}
$$

We still have to write functions for predicates $\mathsf{def}(s, x)$ and $\mathsf{use}(e, x)$, but these are a straightforward exercise now.

$$
\begin{array}{lcl}
\mathsf{def} & : & \mathsf{stm} \times \mathsf{var} \to \mathsf{bool} \\
\mathsf{use} & : & \mathsf{exp} \times \mathsf{var} \to \mathsf{bool}
\end{array}
$$

The whole translation was relatively straightforward, primarily because the rules were well-designed, and because we always had enough information to just write a boolean function.

7 Maintaining Set of Variables

What we have done above is a perfectly adequate implementation of initialization checking. But we might also try to rewrite it in order limit the number of traversals of the statements. For example, in

$$
\mathsf{live}(\mathsf{seq}(s_1, s_2), x) \quad = \quad \mathsf{live}(s_1, x) \vee (\neg \mathsf{def}(s_1, x) \wedge \mathsf{live}(s_2, x))
$$

we may traverse s_1 twice: once to check if x is live in s_1, and once to see if x is defined in s_1. In general, we might traverse statements multiple times, namely for each variable declaration in whose scope it lies. This in itself is not a performance bug, but let's see how one might change it.

One way this can often be done is to notice that for any statement s, there could be multiple variables x such that $\mathsf{live}(s, x)$ or $\mathsf{def}(s, x)$ holds. We can try to combine these into a *set*. We denote a set of variables with δ and define the following two judgments:

- $\mathsf{init}(\delta, s, \delta')$: assuming all the variables in δ are defined when s is reached, no uninitialized variable will be referenced and after its execution all the variables in δ' will be defined.
- $\mathsf{use}(\delta, e)$: e will only reference variables defined in δ.

As a common convention, we isolate assumptions on the left-hand side of a turnstile symbols are write these:

- $\delta \vdash s \Rightarrow \delta'$ for $\mathsf{init}(\delta, s, \delta')$.
- $\delta \vdash e$ for $\mathsf{use}(\delta, e)$.

From the previous rules we develop the following:

$$\frac{}{\delta \vdash \mathsf{nop} \Rightarrow \delta} \qquad \frac{\delta \vdash s_1 \Rightarrow \delta_1 \quad \delta_1 \vdash s_2 \Rightarrow \delta_2}{\delta \vdash \mathsf{seq}(s_1, s_2) \Rightarrow \delta_2}$$

$$\frac{\delta \vdash e}{\delta \vdash \mathsf{assign}(x, e) \Rightarrow \delta \cup \{x\}} \qquad \frac{\delta \vdash e \quad \delta \vdash s_1 \Rightarrow \delta_1 \quad \delta \vdash s_2 \Rightarrow \delta_2}{\delta \vdash \mathsf{if}(e, s_1, s_2) \Rightarrow \delta_1 \cap \delta_2}$$

$$\frac{\delta \vdash e \quad \delta \vdash s \Rightarrow \delta'}{\delta \vdash \mathsf{while}(e, s) \Rightarrow \delta} \qquad \frac{\delta \vdash s \Rightarrow \delta'}{\delta \vdash \mathsf{decl}(y, \tau, s) \Rightarrow \delta' - \{y\}}$$

$$\frac{\delta \vdash e}{\delta \vdash \mathsf{return}(e) \Rightarrow \{x \mid x \text{ in scope}\}}$$

It is worth reading these rules carefully to make sure you understand them. The last one is somewhat problematic, since we don't have enough information to know which declarations we are in the scope of. We should generalize our judgment to $\gamma \; ; \; \delta \vdash s \longrightarrow \delta'$, where γ is the set of all variables currently in scope. Then the last rule might become

$$\frac{\delta \vdash e}{\gamma \; ; \; \delta \vdash \mathsf{return}(e) \Rightarrow \gamma}$$

and we would systematically add γ to all other judgments. We again leave this as an exercise.

In these judgments we have traded the complexity of traversing statements multiple times with the complexity of maintaining variables sets.

8 Modes of Judgments

If we consider the judgment $\delta \vdash e$ there is nothing new to consider: we would translate this to a function

$$\mathsf{use} : \mathsf{set\,var} \times \mathsf{exp} \to \mathsf{bool}$$

On the other hand, it does not work to translate $\delta \vdash s \Rightarrow \delta'$ as

$$\mathsf{init} : \mathsf{set\,var} \times \mathsf{stm} \times \mathsf{set\,var} \to \mathsf{bool}$$

This is because, in general, we do not know δ' before we start out. We need to *compute* it as part of building the deduction! So we need to implement

$$\mathsf{init} : \mathsf{set\,var} \times \mathsf{stm} \to \mathsf{bool} \times \mathsf{set\,var}$$

In order to handle $\mathsf{return}(e)$, we probably should also pass in a second set of declared variables or a context. We could also avoid returning a boolean by just returning an optional set of defined variables, or raise an exception in case we disover a variable that is used but not defined.

Examining the rules shows that we will need to be able to add variables to and remove variables from sets, as well as compute intersections. Otherwise, the code should be relatively straightforward.

Before we actually start this coding, we should go over the inference rules to make sure we always have enough information to compute the output δ' given the inputs δ and s. This is the purpose of *mode checking*. Let's go over one example:

$$\frac{\delta \vdash s_1 \Rightarrow \delta_1 \quad \delta_1 \vdash s_2 \Rightarrow \delta_2}{\delta \vdash \mathsf{seq}(s_1, s_2) \Rightarrow \delta_2}$$

Initially, we know the input δ and $s = \mathsf{seq}(s_1, s_2)$. This means we also know s_1 and s_2. We cannot yet compute δ_2, since the required input δ_1 in the second premise is unknown. But we can compute δ_1 from the first premise since we know δ and s_1 and this point. This gives us δ_1 and we can now compute δ_2 from the second premise and return it in the conclusion.

$$\mathsf{init}(\delta, \mathsf{seq}(s_1, s_2)) \quad = \quad \textbf{let } \delta_1 = \mathsf{init}(\delta, s_1) \textbf{ in } \mathsf{init}(\delta_1, s_2)$$

9 Typing Judgments[1]

Arguably the most important judgment on programs is whether they are well-typed. The *context* (or *type environment*) Γ assigns types to variables. This is often defined explicitly with

$$\Gamma ::= \cdot \mid \Gamma, x{:}\tau$$

where we write $\Gamma(x) = \tau$ when $x{:}\tau$ in Γ. The typing judgment for expressions

$$\Gamma \vdash e : \tau$$

verifies that the expression e is well-typed with type τ, assuming the variables are typed as prescribed by Γ. Most of the rules are straightforward; we show a couple.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \;\&\&\; e_2 : \mathsf{bool}}$$

[1] Not covered in lecture

Typing for statements is slightly more complex. Statements are executed for their effects, but statements in the body of a functions also ultimately return a value. We write

$$\Gamma \vdash s : [\tau]$$

to express that s is well-typed in context Γ. If s returns (using a $\mathsf{return}(e)$ statement), then e must be of type τ. We use this to check that no matter how a function returns, the returned value is always of the correct type.

$$\frac{\Gamma(x) = \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash \mathsf{assign}(x, e) : [\tau]} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{if}(e, s_1, s_2) : [\tau]}$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash s : [\tau]}{\Gamma \vdash \mathsf{while(e, s)} : [\tau]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{return}(e) : [\tau]}$$

$$\frac{}{\Gamma \vdash \mathsf{nop} : [\tau]} \qquad \frac{\Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \mathsf{seq}(s_1, s_2) : [\tau]}$$

$$\frac{\Gamma, x{:}\tau' \vdash s : [\tau]}{\Gamma \vdash \mathsf{decl}(x, \tau', s) : [\tau]}$$

In the last rule for declarations, we might prohibit shadowing of variables by requiring that $x \notin \mathsf{dom}(\Gamma)$. Alternatively, we could stipulate that the rightmost occurrence of x in Γ is the one considered when calculating $\Gamma(x)$. It is also possible that we already know that no conflict can occur, since shadowing may have been ruled out during elaboration already.

10 Modes for Typing

When implementing type-checking, we need to decide on a *mode* for the judgment. Clearly, we want the context Γ and the expression e or statement s to be known, but what about the type?

We first look at expression typing, $\Gamma \vdash e : \tau$. Can we always know τ? Perhaps in our small language fragment from this lecture, but not in L3. For example, if we check an expression $e_1 == e_2 : \mathsf{bool}$, we may know it is of type bool but we do not know the types of e_1 or e_2 (they could be bool or int). Similarly, if we have an expression used as a statement, we do not know the type of expression. Therefore, we should implement a function that takes the context Γ and e as input and *synthesizes* a type τ such that $\Gamma \vdash e : \tau$ (if such a type exists, and fails otherwise). The resulting type τ can be then be compared to a given type if that is known. Of course, you should go through the rules and verify that one can indeed always synthesize a type.

For the typing of statements $\Gamma \vdash s : [\tau]$ the situation is slightly different. Because τ is the return type of the function in which s occurs, we will know τ instead of having to synthesize is. We say we *check* the statement s against the return type τ.

Therefore, if we assume that functions raise an exception if an expression or statement is not well-typed, we might have functions such as

$$\begin{array}{lcl} \mathsf{syn\_exp} & : & \mathsf{ctx} \times \mathsf{exp} \to \mathsf{tp} \\ \mathsf{chk\_stm} & : & \mathsf{ctx} \times \mathsf{stm} \times \mathsf{tp} \to \mathsf{unit} \end{array}$$

For convenience, we might also write a function

$$\mathsf{chk\_exp} \quad : \quad \mathsf{ctx} \times \mathsf{exp} \times \mathsf{tp} \to \mathsf{unit}$$

where $\mathsf{chk\_exp}(e, \tau)$ would simply synthesize a type τ' for e and compare it to τ.

Questions

1. Write out the rules for proper returns: along each control flow path starting at the beginning of a function, there must be a return statement. Clearly, this is a *must*-property.
2. Write some cases in the functions for type-checking.

References