Expressions and Statements

Kyle Dewey

1 Introduction

The previous handout considered a language which consisted only of expressions, no statements. In this handout, we use fundamentally the same language, but use statements instead of expressions for `let` and `assign`.

1.1 Language Used

We'll first define the abstract syntax for our language, shown below:

$$x \in \mathit{Variable} \qquad i \in \mathit{Integer}$$

$$\begin{aligned}
\tau \in \mathit{Type} &::= \textsf{int} \mid \textsf{bool} \\
e \in \mathit{Exp} &::= x \mid i \mid \textsf{true} \mid \textsf{false} \mid e_1 \mathbin{\&\&} e_2 \mid e_1 + e_2 \mid e_1 < e_2 \\
s \in \mathit{Stmt} &::= \textsf{let}\ x : \tau = e \mid x = e \\
p \in \mathit{Program} &::= s \mid s\ p
\end{aligned}$$

Compared to the last handout, there are two notable changes:

- `let` and assignment have been made into statements.
- A program is now explicitly defined as either a statement, or a statement followed by another program. Phrased more simply, a program consists of one or more statements.

2 Type System

Now that we have the syntax defined, we can define the language's type system. Formerly, this entailed the definition of a single set of rules which operated over the language's expressions. However, since we now have both statements and expressions, we need to define *two* sets of rules: one for expressions and one for statements. Both sets of rules will need type environments (expressions have variable access, and all statements currently involve variables), so we'll define that first:

$$\Gamma \in \mathit{TypeEnv} = \mathit{Variable} \rightarrow \mathit{Type}$$

Note this is the same type environment definition from last time; this is still a mapping of variables to types. From here, we define the rules for expressions, statements, and programs in the subsections below.

2.1 Rules for Expressions

The rules for expressions are identical to what we had before, though without `let` and assignment:

$$\frac{}{\Gamma \vdash i : \textsf{int}}\ (\textsc{integer}) \qquad \frac{}{\Gamma \vdash \textsf{true} : \textsf{bool}}\ (\textsc{true}) \qquad \frac{}{\Gamma \vdash \textsf{false} : \textsf{bool}}\ (\textsc{false})$$

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \quad \Gamma \vdash e_2 : \textsf{bool}}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 : \textsf{bool}}\ (\textsc{and}) \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash e_1 + e_2 : \textsf{int}}\ (\textsc{plus}) \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash e_1 < e_2 : \textsf{bool}}\ (\textsc{less-than})$$

$$\frac{x \in \texttt{dom}(\Gamma) \quad \tau = \Gamma[x]}{\Gamma \vdash x : \tau}\ (\textsc{var})$$

2.2 Rules for Statements

Statements are a bit trickier. By their nature, statements don't give back values, so it doesn't make sense for them to have a return type. However, statements *do* have an effect: they add and interact with variables in scope. With this in mind, while statements don't have corresponding types, they instead produce new type environments. If we were to implement this in Java, when typechecking statements, we'd have a type signature something like the following:

```
public Map<Variable, Type> typecheckStatement(Stmt s, Map<Variable, Type> env)
  throws IllTypedException
```

In formal notation, we'll have our statements give back type environments instead of types. To help keep things unambiguous, we'll use $\vdash_s$ to indicate the typechecking of statements, whereas just $\vdash$ refers to typechecking expressions. With this in mind, the typing rules for statements are below:

$$\frac{\Gamma \vdash_s e : \tau}{\Gamma \vdash_s \textbf{let } x : \tau = e : \Gamma[x \mapsto \tau]} \text{ (LET)} \quad \frac{x \in \texttt{dom}(\Gamma) \quad \tau = \Gamma[x] \quad \Gamma \vdash_s e : \tau}{\Gamma \vdash_s x = e : \Gamma} \text{ (ASSIGN)}$$

As shown, `let` adds a new variable to the type environment, making sure that the type of the variable (τ) is the same as the type of the expression (e). Assignment doesn't change the input type environment (Γ) at all. Instead, it makes sure the variable being assigned to (x) is in scope, and makes sure the type of e is the same type as the variable in scope (τ).

2.3 Rules for Programs

We can now define the typechecking rules for whole programs, where a program is one or more statements. The idea is that we can chain along the type environment as we process statements. As such, typechecking whole programs is similar to typechecking statements: we take an input type environment, and produce an output type environment. We'll use $\vdash_p$ to help keep things unambiguous. This is shown below:

$$\frac{\Gamma_1 \vdash_s s : \Gamma_2}{\Gamma_1 \vdash_p s : \Gamma_2} \text{ (PROG-ONE)} \quad \frac{\Gamma_1 \vdash_s s : \Gamma_2 \quad \Gamma_2 \vdash_p p : \Gamma_3}{\Gamma_1 \vdash_p s\ p : \Gamma_3} \text{ (PROG-MULTI)}$$

Note that the premise of PROG-ONE uses the typechecking rules for statements instead of programs.