COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 11: Formalizing Typed Programs

Finley McIlwaine

University of Wyoming

fmcilwai@uwyo.edu

March 1, 2022

Review

Last time we started looking at how we can formalize type systems. We did this using logical notation like the following:

$$\Gamma \vdash_e e : T$$

The above statements are called *judgements*, and they give us information about the type of the expressions e. The above says the expressions e has type T in the environment Γ.

The symbol $\vdash_e$ is the typing relation for expressions. Formally, it is a 3-place relation of environments, expressions, and types. If a triple (Γ, e, T) is in $\vdash_e$, then we can say e is well-typed and it has type T in environment Γ.

Review

Our typing relation needed to include some expressions that contained other expressions. In other words, we wanted the following to be true:

$$x : int, y : int \vdash_e x \leq y : bool$$

We used *inference rule* notation to represent those cases. For example, the following rule that adds well-typed less-than-or-equal-to expressions to the typing relation:

$$\frac{\Gamma \vdash_e e_1 : int \qquad \Gamma \vdash_e e_2 : int}{\Gamma \vdash_e e_1 \leq e_2 : bool}$$

Inference rules can be read similarly to logical implications. The above says the expression $e_1 \leq e_2$ has type *bool* in environment Γ if and only if e_1 and e_2 both have type *int* in environment Γ.

Well-Typed Statements

Last time, I mentioned that we'll need typing relations for each non-terminal in our grammar. Now that we have the typing relation for expressions figured out, let's develop the relation for *statements*.

Well-Typed Statements

Let's see what our type checker does when it type checks the statement

```
if true || false then print 1; else print 2;
```

Well-Typed Statements

Let's see what our type checker does when it type checks the statement

`if true || false then print 1; else print 2;`

```
@Override
public Void visit(SIf p, Void arg) {
    ImpType expType = p.exp_.accept(this, arg);
    if (expType.equals(ImpType.T_BOOL)) {
        p.stm_1.accept(this, arg);
        p.stm_2.accept(this, arg);
        return null;
    }
    throw new ImpTypeError(" ... ");
}
```

Checks that the expression has type bool, then type checks both statements!

Well-Typed Statements

So we expect our typing relation for statements to somehow restrict what types of expressions can occur where, if necessary. This makes the relation for statements depend on the relation for expressions!

Also notice that we don't get an explicit result from the visit methods for statements...

Well-Typed Statements

Let's see what our type checker does when it type checks the statement `print 1;`

Well-Typed Statements

Let's see what our type checker does when it type checks the statement `print 1;`

```
@Override
public Void visit(SPrint p, Void arg) {
    p.exp_.accept(this, arg);
    return null;
}
```

Just checks that the expression *has* a type, but doesn't care what type it has.

Well-Typed Statements

We're seeing here how type checking relies on type inference. Whenever an expression occurs in a statement, we infer its type and then check that the type fits the place it occurs in the statement. We've already seen this in the way we type check expressions.

Well-Typed Statements

What does our type checker do when it type checks the statement `x := 5`?

Well-Typed Statements

What does our type checker do when it type checks the statement `x := 5`?

```
@Override
public Void visit(SInit p, Void arg) {
    environment.get(0).put(p.ident_, p.exp_.accept(this, arg));
    return null;
}
```

It sets the type of `x` to the type of the expression in the local level of the environment! This hints at how our simple language is handling variable scopes, and it means we need to start building "scoping" into our environments in our specification.

It also means we need to somehow represent the fact that statements may update the environment.

Well-Typed Statements

So here's everything we've learned about the statement typing relation by looking at our implementation:

- Must only allow certain types of expressions in certain places, any type of expressions in others.
- Must depend on the expression typing relation.
- Must somehow represent the updating of environments.

Well-Typed Statements

The statement typing relation is similar to the expression typing relation, except it doesn't relate expressions and environments to types, it relates statements and environments to *new environments*!

Well-Typed Statements

With that, we have discovered that the typing relation for statements is a 3-place relation of environments, statements, and environment resulting from executing that statement in that environment.

If a triple is in the relation, which we'll denote as $\vdash_s$, we write:

$$\Gamma \vdash_s stm \Rightarrow \Gamma'$$

Where Γ is the environment (just a map from identifier to type), *stm* is the expression, and Γ' is the environment resulting from running *stm* in Γ.

Environments Vs. Contexts

We now need to introduce some notation that will allow us to talk about the scopes of variables in the environment. So let's think about how we might want to do this.

In general, every variable binding exists in the scope (block) it occurs in and every block that occurs *after* that binding until it is redeclared.

Variables are "shadowed" if a binding occurs in a scope beneath the scope the original binding occured in.

Environments Vs. Contexts

Here are some example statements, annotated with the environment that exists at each step:

```
x := 5;              -- [[(x,int)]]
if true then {       -- [[],[(x,int)]]
    x = 10;          -- [[],[(x,int)]]
    y := true;       -- [[(y,bool)],[(x,int)]]
    x := false;      -- [[(y,bool),(x,bool)],[(x,int)]]
} else {             -- [[],[(x,int)]]
    x := true;       -- [[(x,bool)],[(x,int)]]
    y := 4;          -- [[(x,bool),(y,int)],[(x,int)]]
}                    -- [[(x,int)]]
y := 10;             -- [[(x,int),(y,int)]]
```

Shadowing occurs at the third statement of the if block and first statement of the else block.

Environments Vs. Contexts

Obviously, if we encounter a variable during type checking, we need to be able to look up its type. And its type comes from the "closest" binding in the environment.

So environments consist of a list-like structure of maps from identifiers to types, where each map is preceded by inner scopes, and succeeded by outer scopes. We refer to the maps as *contexts*, which consist of the local bindings (maps) at each scope (block).

There's many ways to implement this, but the simplest is as lists, where the first element is the most nested context, and the last element is the global context.

Environments Vs. Contexts

This is how it's implemented in the type checker:

```
...
private List<HashMap<String, ImpType>> environment;
...
```

Using lists makes the variable lookup and declaration more straightforward as we will see.

Environments Vs. Contexts

We will still write environments as Γ whenever we refer to an arbitrary environment. If we need to refer to individual contexts, we will also write them as Γ, but separated with a $\triangleright$ character: $\Gamma \triangleright \Gamma'$. In this example, Γ' is the "lowest" context.

Environments Vs. Contexts

Since our typing relation for expressions doesn't much rely on variable scoping, the typing rules stay the same. For example, the typing rule for variable expressions:

$$\frac{id : T \in \Gamma}{\Gamma \vdash_e id : T}$$

So we're abusing set membership notation a little bit here, and saying that $id : T \in \Gamma$ means the closest binding of id in Γ is T.

Type Checking Statements

For statements, the scope of variables does start to matter. For example, the rules for type checking variable initializations and assignments:

$$\frac{\Gamma \vdash_e e : T}{\Gamma \vdash_s id := e \Rightarrow \Gamma, id : T} \qquad \frac{\Gamma \vdash_e e : T \quad id : T \in \Gamma}{\Gamma \vdash_s id = e \Rightarrow \Gamma}$$

The meta-expression $\Gamma, id : T$ means to map id to type T in the local context of environment Γ.

Type Checking Statements

The rule vs. the implementation for initialization:

$$\frac{\Gamma \vdash_e e : T}{\Gamma \vdash_s id := e \Rightarrow \Gamma, id : T}$$

```
@Override
public Void visit(SInit p, Void arg) {
    environment.get(0).put(p.ident_, p.exp_.accept(this, arg));
    return null;
}
```

Type Checking Statements

The rule vs. the implementation for assignment:

$$\frac{\Gamma \vdash_e e : T \qquad id : T \in \Gamma}{\Gamma \vdash_s id = e \Rightarrow \Gamma}$$

```
@Override
public Void visit(SAss p, Void arg) {
    ImpType expType = p.exp_.accept(this, arg);
    for (HashMap<String, ImpType> ctx : environment) {
        if (ctx.containsKey(p.ident_)) {
            if (ctx.get(p.ident_).equals(expType))
                ctx.put(p.ident_, p.exp_.accept(this, arg));
            else
                throw new ImpTypeError("type mismatch");
            return null;
        }
    }
    throw new ImpTypeError("variable not initialized");
}
```

Type Checking Statements

How would we type check a block statement?

Type Checking Statements

How would we type check a block statement?

Here's what the type checker we wrote does:

```
@Override
public Void visit(SBlock p, Void arg) {
    environment.add(0, new HashMap<>());
    for (Stm stm : p.liststm_) {
        stm.accept(this, arg);
    }
    environment.remove(0);
    return null;
}
```

What would the inference rule for block statements look like?

Type Checking Statements

A block statement is only well-typed in an environment Γ if it's list of sub-statements is well-typed in $\Gamma \triangleright \varnothing$, where $\varnothing$ is the empty context.

$$\frac{\Gamma \triangleright \varnothing \vdash_s s_0 \ldots s_n \Rightarrow \Gamma'}{\Gamma \vdash_s \{s_0 \ldots s_n\} \Rightarrow \Gamma}$$

Type Checking Statements

A block statement is only well-typed in an environment Γ if it's list of sub-statements is well-typed in $\Gamma \triangleright \varnothing$, where $\varnothing$ is the empty context.

$$\frac{\Gamma \triangleright \varnothing \vdash_s s_0 \ldots s_n \Rightarrow \Gamma'}{\Gamma \vdash_s \{s_0 \ldots s_n\} \Rightarrow \Gamma}$$

However, with the rules we have so far, we don't know how to type check lists of statements!

Type Checking Statements

A block statement is only well-typed in an environment Γ if it's list of sub-statements is well-typed in $\Gamma \triangleright \varnothing$, where $\varnothing$ is the empty context.

$$\frac{\Gamma \triangleright \varnothing \vdash_s s_0 \ldots s_n \Rightarrow \Gamma'}{\Gamma \vdash_s \{s_0 \ldots s_n\} \Rightarrow \Gamma}$$

However, with the rules we have so far, we don't know how to type check lists of statements! We do this by showing how to type check non-empty and empty lists of statements:

$$\frac{\Gamma \vdash_s s_0 \Rightarrow \Gamma' \qquad \Gamma' \vdash_s s_1 \ldots s_n \Rightarrow \Gamma''}{\Gamma \vdash_s s_0 \ldots s_n \Rightarrow \Gamma''}$$

$$\frac{}{\Gamma \vdash_s \oslash \Rightarrow \Gamma}$$

Where $\oslash$ is the empty list of statements.

Type Checking Statements

We'll finish this lecture by writing the rest of the inference rules for statements and programs, and then finishing our implementation of the Haskell type checker, if we have time.

The End