COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 9: Formalizing Typed Expressions

Finley McIlwaine

University of Wyoming

fmcilwai@uwyo.edu

February 17, 2022

Review

Last time we examined some of the main ideas behind type checking. We looked at untyped vs. dynamically typed vs. statically typed and how each of those systems can change the way programs are ran.

More importantly, though, we got a first look at how type checkers are implemented, and saw that their structure is very similar to that of an intepreter in that they can just traverse a abstract syntax tree.

Specification vs. Implementation

Type checkers *enforce* a type system. This is conceptually similar to how programs can *implement* algorithms. Algorithms and type systems can be specified in many ways, one of those ways being code!

Another way that algorithms are specified is as abstract, language agnostic descriptions of the operations required to complete the task. Many important algorithms are specified in this manner.

Specification vs. Implementation

Type systems and programming language semantics are no different. It's possible to specify them simply through implementation in code as type checkers or interpreters, but it's often more useful to use abstract, language agnostic devices.

- Mathematical specifications allow us to formally prove properties of the systems.
- Easier to translate into implementations in multiple languages.

Specifying a Type System

We now wish to provide such an abstract and formal specification of the static type system that we implemented in our type checker last time.

Recall that we implemented untyped, dynamically typed, and statically typed systems. We're only going to worry about specifying static type systems since the language we're going to implement in this course is statically typed.

Specifying a Type System

Let's start by remembering exactly what our type checker did.

- Given a program, determined whether or not that program was *well typed*.
- What is a *well typed* program? One that does not contain type errors!
- What is a type error? It depends on the language but in the case of our simple imperatice language, it is when some expression does not have the *expected* type.

A key idea here is that our type checker takes a program, and determines if it well typed. In contrast, we will specify our type system by effectively *defining the set of well typed programs*. The type checker can then be seen as an algorithm for determining membership of this set.

Type Checking Expressions

For every non-terminal in our language (programs, statements, expressions) we will define a *typing relation* that encodes the typing rules for that non-terminal. Recall that a relation is a set of tuples of a specific size.

Let's think about what we expect the typing relation for expressions to be.

Type Checking Expressions

What does our type checker do when it type checks the expression `true`?

Type Checking Expressions

What does our type checker do when it type checks the expression `true`?

```
@Override
public ImpType visit(ETru p, Void arg) {
    return ImpType.T_BOOL;
}
```

It returns the boolean type! So it maps the expression `true` to the boolean type. Based on this, we would expect the typing relation for expressions to somehow relate expressions to their types.

Type Checking Expressions

What does our type checker do when it type checks the expression `x`?

Type Checking Expressions

What does our type checker do when it type checks the expression `x`?

```
@Override
public ImpType visit(EVar p, Void arg) {
    for (HashMap<String, ImpType> ctx : environment) {
        if (ctx.containsKey(p.ident_)) {
            return ctx.get(p.ident_);
        }
    }
    throw new RuntimeException(" ... ");
}
```

It looks up the type of `x` in the environment and returns it, or throws an error if it doesn't find it! This means that the typing relation for expressions should somehow include the environment.

Type Checking Expressions

With that, we have discovered that the typing relation for expressions is a 3-place relation (meaning each tuple has three elements). Each tuple contains an environment, an expression, and the type of that expression in that environment.

If a triple (3-tuple) is in the relation, which we'll denote as $\vdash_e$, we could write:

$$(\Gamma, e, T) \in \vdash_e$$

Where Γ is the environment (just a map from identifier to type), e is the expression, and T is the type of e in environment Γ.

Type Checking Expressions

$$(\Gamma, e, T) \in \vdash_e$$

The above notation is classic set membership notation, which should drive home the point that the typing relation for expressions is really only a set of 3-tuples of environments, expressions and types.

However, we will adopt a notation convention from logic that makes these relations a little friendlier to work with. If an expression e has type T in environment Γ, we will write:

$$\Gamma \vdash_e e : T$$

Type Checking Expressions

For example, the following is true:

$$\varnothing \vdash_e \mathit{true} : \mathit{bool}$$

Where $\varnothing$ is the empty environment (empty map). The following is also true:

$$x : \mathit{int} \vdash_e \mathit{true} : \mathit{bool}$$

Obviously *true* : *bool* in any arbitrary environment, so we can simply write:

$$\Gamma \vdash_e \mathit{true} : \mathit{bool}$$

Where Γ represents an arbitrary environment.

Type Checking Expressions

The case of variables is a little more nuanced. Only variables that have a type in the environment should be in the typing relation. We could encode this using a classical logic implication:

$$id : T \in \Gamma \rightarrow \Gamma \vdash_e id : T$$

Where *id* is an arbitrary variable, and we write $x : T \in \Gamma$ to mean variable x has type T in environment Γ.

Instead, we will adopt some more notation from logic called *inference rules*. The above implication can be rewritten as the following inference rule:

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

Type Checking Expressions

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

Inference rules include a set of premises and a conclusion. For the conclusion to be true, the premises must be true. The above inference rule says that in order for *id* to have type T in environment Γ, *id* must be mapped to type T in Γ.

We also typically write typing rules without any premises with the bar still. So the typing rule for the expression *true* is written:

$$\frac{}{\Gamma \vdash_e true : bool}$$

Type Checking Expressions

Inference rules make it very easy to specify the typing rules for other types of expressions. Let's see how our type checker type checked addition expressions:

```
@Override
public ImpType visit(EAdd p, Void arg) {
    ImpType t1 = p.exp_1.accept(this, arg);
    if (!t1.equals(ImpType.T_INT))
        throw new ImpTypeError(" ... ");

    ImpType t2 = p.exp_2.accept(this, arg);
    if (!t2.equals(ImpType.T_INT))
        throw new ImpTypeError(" ... ");

    return ImpType.T_INT;
}
```

Type Checking Expressions

So we need to create an inference rule for typing addition expressions that requires both of the subexpressions to be integers. . .

Type Checking Expressions

So we need to create an inference rule for typing addition expressions that requires both of the subexpressions to be integers. . .

$$\frac{\Gamma \vdash_e e_1 : \mathit{int} \qquad \Gamma \vdash_e e_2 : \mathit{int}}{\Gamma \vdash_e e_1 + e_2 : \mathit{int}}$$

This rule says that in order for an addition expression to have type *int* in environment Γ, both of its subexpressions must have type *int* in the same environment.

Type Checking Expressions

For the rest of this lecture, we will work on writing out the rest of the typing rules for the remaining expressions in our language. We will then implement the type checker that enforces those rules in Haskell.

The End