Today's Lecture Notes for cs173

sk and dbtucker

October 6, 2000

Semantics

We have been writing interpreters in Scheme in order to understand various features of programming languages. What if we want to explain our interpreter to someone else? If that person doesn't know Scheme, we can't communicate how our interpreter works. It would be convenient to have some common language for explaining interpreters. We already have one—math!

Let's try some simple examples. If our program is a number n, it just evaluates to some mathematical representation of n:

$$n \Rightarrow \ulcorner n \urcorner$$

How about addition?

$$(le + re) \Rightarrow \ulcorner le + re \urcorner$$

This definition doesn't work, since le and re might have to be evaluated themselves. If they evaluate to values, then we can compute the result:

$$\frac{le \Rightarrow lv \quad re \Rightarrow rv}{(le + re) \Rightarrow \ulcorner lv + rv \urcorner}$$

Now let's do functions. They evaluate to closures, which we represent as tuples in math:

$$(\mathsf{fun}\,(i)\,e) \Rightarrow \langle i, e, ? \rangle$$

We have a problem—the closure needs an environment. We can represent the environment as a function $\mathcal{E}$ from variables to values. We need to rewrite our rules to use the environment:

$$n; \mathcal{E} \Rightarrow \ulcorner n \urcorner$$

$$\frac{le; \mathcal{E} \Rightarrow lv \quad re; \mathcal{E} \Rightarrow rv}{(le + re); \mathcal{E} \Rightarrow \ulcorner lv + rv \urcorner}$$

$$(\mathsf{fun}\,(i)\,e); \mathcal{E} \Rightarrow \langle i, e, \mathcal{E} \rangle$$

Now that we have an environment, we can evaluate variables:

$$var; \mathcal{E} \Rightarrow \mathcal{E}(var)$$

The only rule that remains is application. This one is a bit tricky. We need to evaluate the function expression and argument expression, then evaluate the body of the function in an extended environment (i.e. the closure environment plus the binding of the function's argument):

$$\frac{fe; \mathcal{E} \Rightarrow \langle i, e, \mathcal{E}' \rangle \quad ae; \mathcal{E} \Rightarrow av \quad e; \mathcal{E}'[i \leftarrow av] \Rightarrow appv}{(fe\ (ae)) \Rightarrow appv}$$

That's it. Now we have completely described our interpreter using math.

Let's see what happens when we evaluate $((\mathsf{fun}\ (x)\ (x + x))\ ((3 + 4)))$:

$$\frac{(\mathsf{fun}\ (x)\ (x+x)); \{\} \Rightarrow \langle x, (x+x), \{\} \rangle \quad \frac{3; \{\} \Rightarrow \ulcorner 3 \urcorner \quad 4; \{\} \Rightarrow \ulcorner 4 \urcorner}{(3+4); \{\} \Rightarrow \ulcorner 7 \urcorner} \quad \frac{x; \{x \mapsto 7\} \Rightarrow \ulcorner 7 \urcorner \quad x; \{x \mapsto 7\} \Rightarrow \ulcorner 7 \urcorner}{(x+x); \{x \mapsto 7\} \Rightarrow \ulcorner 14 \urcorner}}{((\mathsf{fun}\ (x)\ (x+x))\ ((3+4))); \{\} \Rightarrow \ulcorner 14 \urcorner}$$

Notice that the computation is represented as a tree in our mathematical system.

In the following sections, we'll also use conditional expressions. Here are the rules:

$$\mathsf{true}; \mathcal{E} \Rightarrow \#t$$

$$\mathsf{false}; \mathcal{E} \Rightarrow \#f$$

$$\frac{ce; \mathcal{E} \Rightarrow \#t \quad te; \mathcal{E} \Rightarrow tv}{(\mathsf{if}\ ce\ te\ fe); \mathcal{E} \Rightarrow tv}$$

$$\frac{ce; \mathcal{E} \Rightarrow \#f \quad fe; \mathcal{E} \Rightarrow fv}{(\mathsf{if}\ ce\ te\ fe); \mathcal{E} \Rightarrow fv}$$

Types

Our mathematical definition doesn't say anything about the result of type mismatches—say, when we try to apply a number to a number. Real programming languages prevent these errors via type checking. Some languages, such as Scheme, check types dynamically (i.e. at run time). Others, such as ML, check types statically (i.e. at compile time). We will develop a static type checker for our language. A type checker is a function which takes a program (a parse tree) and returns true only if the program will not produce any type errors when executed.

We want to write a function that assigns a type to an expression (and each of its subexpressions). The first question is, what is a type? We know that 5 has type num. The expression $(\mathsf{fun}\ (x)\ (x + 1))$ must be some function type. Here's a simple definition: a *type* is a partitioning of the universe of values (based on what operations are legal).

Let's write a function τ which takes an expression and returns its type, where type is either num, bool, or $\mathsf{function}$. For values, this is easy:

$$\tau(n) = \mathsf{num}$$

$$\tau(\mathsf{true}) = \mathsf{bool}$$

$$\tau(\mathsf{false}) = \mathsf{bool}$$

$$\tau(\mathsf{fun}\,(i)\,e) = \mathsf{function}$$

But what about $\tau(\mathsf{var})$? We need an environment where we can look up var's type. We'll carry around a *type environment*, denoted Γ, which is a function from identifiers to types (as opposed to our regular environment, which maps identifiers to values). In addition, we'll use some funny notation:

$$\Gamma \vdash n : \mathsf{num}$$

Read the above as "Gamma proves that n has type num." Let's make our other rules look like this:

$$\Gamma \vdash \mathsf{true} : \mathsf{bool}$$

$$\Gamma \vdash \mathsf{false} : \mathsf{bool}$$

We can only evaluate addition expressions where the two subexpressions have type num; if so, the result of the addition also has type num:

$$\frac{\Gamma \vdash le : \mathsf{num} \quad \Gamma \vdash re : \mathsf{num}}{\Gamma \vdash (le + re) : \mathsf{num}}$$

How about the conditional expression?

$$\frac{\Gamma \vdash ce : \mathsf{bool} \quad \Gamma \vdash te : ? \quad \Gamma \vdash fe : ?}{\Gamma \vdash (\mathsf{if}\ ce\ te\ fe) : ?}$$

We don't know what types the branches have, but for now we'll assume that they have to be the same. We use a type variable t to enforce this constraint:

$$\frac{\Gamma \vdash ce : \mathsf{bool} \quad \Gamma \vdash te : t \quad \Gamma \vdash fe : t}{\Gamma \vdash (\mathsf{if}\ ce\ te\ fe) : t}$$

This rule says that the two branches have the same type (t), and the conditional expression as a whole has this type.

Next up is function application:

$$\frac{\Gamma \vdash fe : \mathsf{function} \quad \Gamma \vdash ae : at}{\Gamma \vdash fe\,(ae) : ?}$$

Now we have a problem. We don't know what the argument and return types of the function are, so we can't type check this expression. We need to modify our definition of types to be:

$$\begin{array}{rcl} type & = & \mathsf{num} \\ & | & \mathsf{bool} \\ & | & type \rightarrow type \end{array}$$

where $at \to rt$ is the type of a function whose argument has type at and whose return value has type rt.

The correct way to type check application is:

$$\frac{\Gamma \vdash fe : at \to rt \quad \Gamma \vdash ae : at}{\Gamma \vdash fe\,(ae) : rt}$$

The final case is typing a function. We need to know the type of the argument in order to type the body:

$$\frac{\Gamma[i \leftarrow at] \vdash e : rt}{\Gamma \vdash (\mathsf{fun}\,(i)\,e) : at \to rt}$$

This is missing one detail—in order to know that the argument has type at, we need to annotate the argument in the function:

$$\frac{\Gamma[i \leftarrow at] \vdash e : rt}{\Gamma \vdash (\mathsf{fun}\,(i : at)\,e) : at \to rt}$$

Note that we have now changed the grammar of our language—every function now must specify the type of its argument.