Today's Lecture Notes for cs173

sk and dbtucker

October 11, 2000

Topic: Types

Types

Why do we want well-typed programs anyway? Robin Milner, who won a Turing Award in part for his work in type theory, proved a theorem regarding the safety of well-typed programs. Milner's theorem is often summarized as:

> Well-typed programs do not go wrong.

In other words, if a program type-checks, there will be no run-time errors.

Pairs

We can add pairs to our language:

$$\begin{array}{rcl} L & ::= & \ldots \\ & | & (\mathsf{pair}\ \langle L \rangle\ \langle L \rangle) \end{array}$$

If v_1 and v_2 are values, then $(\mathsf{pair}\ v_1\ v_2)$ is also a value in our language. The evaluation rule for pairs is:

$$\frac{e_1; \mathcal{E} \Rightarrow v_1 \quad e_2; \mathcal{E} \Rightarrow v_2}{(\mathsf{pair}\ e_1\ e_2); \mathcal{E} \Rightarrow (\mathsf{pair}\ v_1\ v_2)}$$

We also want to type-check pairs, so we extend our type grammar to include pair types:

$$\begin{array}{rcl} \tau & ::= & \mathsf{num} \\ & | & \mathsf{bool} \\ & | & \tau \rightarrow \tau \\ & | & \langle \tau, \tau \rangle \end{array}$$

The type grammar defines the set of all possible types, including num, bool, $\mathsf{bool} \rightarrow \mathsf{num}$, $(\mathsf{num} \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool}$, $\langle \mathsf{num}, \mathsf{bool} \rangle$, $\langle \langle \mathsf{num}, \mathsf{num} \rangle, \mathsf{bool} \rightarrow \mathsf{bool} \rangle$, and many, many more.

To create pair types, we first examine the types of the subexpressions:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (\mathsf{pair}\ e_1\ e_2) : \langle t_1, t_2 \rangle}$$

In other words, if the first item of the pair has type t_1, and the second item has type t_2, then the pair itself has type $\langle t_1, t_2 \rangle$.

We also need primitives to extract the components of pairs. Assume we have a primitive function π_1 that returns the first component, and a primitive π_2 that returns the second:

$$\pi_1 : \langle t_1, t_2 \rangle \to t_1$$

$$\pi_2 : \langle t_1, t_2 \rangle \to t_2$$

The evaluation rule for π_1 is:

$$\frac{e; \mathcal{E} \Rightarrow (\mathsf{pair}\ v_1\ v_2)}{\pi_1(e); \mathcal{E} \Rightarrow v_1}$$

Lists

Now we want to develop a representation for lists. We can represent lists using nested pairs; for example, the list of numbers 1, 2, and 3 would be:

$$\langle 1, \langle 2, \langle 3, 4 \rangle \rangle \rangle$$

Can we write a function first that takes a list and returns the first element? Here's one attempt:

$$\mathsf{first} \quad \equiv \quad (\mathsf{fun}\ (x)\ \pi_1(x))$$

This attempt doesn't even match the syntax of our typed language, since the function's argument lacks a type annotation. Here's another attempt:

$$\mathsf{first} \quad \equiv \quad (\mathsf{fun}\ (x : \langle \mathsf{num}, \langle \mathsf{num}, \langle \mathsf{num}, \mathsf{num} \rangle \rangle \rangle)\ \pi_1(x))$$

This function, when applied to the list $\langle 1, \langle 2, \langle 3, 4 \rangle \rangle \rangle$, will correctly return the first element (1).

However, what if we want the first element of $\langle 1, \langle 2, 3 \rangle \rangle$? The type annotation on the argument of first requires that the list have length four, so we have to write an entirely new function to deal with lists of length three:

$$\mathsf{first}' \quad \equiv \quad (\mathsf{fun}\ (x : \langle \mathsf{num}, \langle \mathsf{num}, \mathsf{num} \rangle \rangle)\ \pi_1(x))$$

This solution is not general—we want a single function first that can operate on lists of *any* length.

To overcome this problem, we will add Scheme-like lists to our language; for example:

$$(\mathsf{cons}\ 1\ (\mathsf{cons}\ 2\ (\mathsf{cons}\ 3\ \mathsf{empty})))$$

is the list of numbers 1, 2, and 3. We extend the language grammar to include the list constructors cons and empty, as well as the primitives first and rest:

$$
\begin{array}{lcl}
L & ::= & \ldots \\
 & | & \mathsf{empty} \\
 & | & (\mathsf{cons}\ \langle L \rangle\ \langle L \rangle) \\
 & | & (\mathsf{first}\ \langle L \rangle) \\
 & | & (\mathsf{rest}\ \langle L \rangle)
\end{array}
$$

Additionally, we'll add a list type to our type grammar:

$$
\begin{array}{lcl}
\tau & ::= & \ldots \\
 & | & \mathsf{list}(\tau)
\end{array}
$$

Note that in order to be able to do type checking, we require that all elements of a list have the same type.

We can write a rule to type-check cons—we just make sure that the tail is a list, and that type of the head matches that of the tail's elements. More precisely:

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : \mathsf{list}(t)}{\Gamma \vdash (\mathsf{cons}\ e_1\ e_2) : \mathsf{list}(t)}$$

The primitive functions first and rest have the following types:

$$\mathsf{first} : \mathsf{list}(\tau) \rightarrow \tau$$

$$\mathsf{rest} : \mathsf{list}(\tau) \rightarrow \mathsf{list}(\tau)$$

Now what is the type of empty? We know it's a list of something, but is it a list(num), or list(bool), or list(bool → num), or what? In truth, we want it to have type list(α) where α can match any type—but, we don't have any types that look like this (remember the type grammar).

We're stuck. For now, we'll have to make empty some particular kind of list, say a list of numbers. We'll call the type nlist:

$$\mathsf{empty} : \mathsf{nlist}$$

Now we have to rewrite the type rule for cons, since we only have number lists:

$$\frac{\Gamma \vdash e_1 : \mathsf{num} \quad \Gamma \vdash e_2 : \mathsf{nlist}}{\Gamma \vdash (\mathsf{cons}\ e_1\ e_2) : \mathsf{nlist}}$$

The types of first and rest also change:

$$\mathsf{first} : \mathsf{nlist} \rightarrow \mathsf{num}$$

$$\mathsf{rest} : \mathsf{nlist} \rightarrow \mathsf{nlist}$$

Termination

Let's write an infinite loop in our typed language. Before we had types, we could write the following expression:

$$((\mathsf{fun}\,(g)\ (g\ (g)))\ \ ((\mathsf{fun}\,(f)\ (f\ (f)))))$$

We can reduce this expression by substituting the argument $(\mathsf{fun}\,(f)\ (f\ (f)))$ for the parameter g in the body $(g\ (g))$, which gives us:

$$((\mathsf{fun}\,(f)\ (f\ (f)))\ \ ((\mathsf{fun}\,(f)\ (f\ (f)))))$$

Note that this expression is the same as the first one (other than the renaming of g to f). In fact, if we reduce this expression again, we get:

$$((\mathsf{fun}\,(f)\ (f\ (f)))\ \ ((\mathsf{fun}\,(f)\ (f\ (f)))))$$

Since this expression just reduces to itself, it will never finish computing to a value (i.e., it's an infinite loop, it diveges, it runs forever).

In the typed version of our language, however, we have to assign a type t to the function's argument:

$$(\mathsf{fun}\,(f : t)\ (f\ (f)))$$

Look at the first f in the body. It is used in the function position of an application, so its type t, must be $? \to ?$. The argument to this function is the second f in the body. Since f's type is t, this means the argument position of t must be t itself, that is,

$$t \quad = \quad t \to ?$$

However, we don't have any types in our set that satisfy this equality.

Looking at this equality a bit more closely, we see that substituting $(t \to ?)$ for the t in the right-hand side, we get:

$$t \quad = \quad (t \to ?) \to ?$$

This type would work if the recursion only went two levels deep. If we substitute again, we get:

$$t \quad = \quad ((t \to ?) \to ?) \to ?$$

This type would work for three levels of recursion. We can continue on in this fashion; however, we won't be able to write down a type for an unbounded number of recursions.

In fact, we can't write down an infinite computation as a well-typed program in our language (proof not given). This property is called *strong normalization*, and can be useful in applications which require that computations terminate. Some examples are:

1. languages for real-time systems
2. program linkers

3. packet filters in network stacks
4. programs in routers
5. configuration files for applications