Types

Lecture Notes for cs173, Fall 2001

sk and rob

October 22, 2001

Scheme has a nice feature: When a primitive operation is used incorrectly (e.g. (+ 2 (lambda () 3))), the evaluator signals a (run-time) error. In a language like C (which does not perform the run-time checks that Scheme does), a primitive operation applied incorrectly might cause your program to seg fault at some (seemingly random) point in your program’s evaluation. Or worse, the program might not crash at all, and you, as the programmer or program user, will have no idea that something went wrong.

Now imagine that the program you are writing launches a rocket into space. Suddenly, Scheme’s run-time checks are no longer acceptable. It would be horrible if just as the rocket was about to exit the earth’s atmosphere, a run-time error occurred, causing our program (and rocket) to crash.

We would like to have some sort of guarantee about how our program will evaluate. The Halting Problem dictates that we can’t expect to get the result of a program without running it, but there is something useful that we can statically guarantee: A program (that passes a certain “check”) will never wrongly apply a primitive operation.

In the context of Rip programs, passing this check would mean that a program will never try to apply a number and never try to add a procedure. We can accomplish this by assigning types to Rip expressions. A type is simply a name given to a set of values. We will define the type num, for example, to represent the entire set of Rip numbers.

We will now write a series of typing rules, analogous to the evaluation rules from last class. Let’s first consider Rip numbers:

\[ n : \text{num}, \text{where } n \text{ is a Rip number} \]

The \( : \) means “is of type.” We read the whole rule as, “the Rip number \( n \) is of type num.” That was easy. Let’s write the rule for Rip add expressions:

\[ \text{LHS : num, RHS : num, } \{ + \text{ LHS RHS} \} : \text{num} \]

where LHS and RHS are Rip expressions

---

1We are using slightly different notation from what was presented in class. This notation (namely, the “:\”) is common in the programming languages community. The notation used in class was equivalent, and used just to make the transition from evaluation rules to typing rules easier.
This typing rule uses a judgment, just as in our evaluation rules from last time. We read this rule as, “if LHS is of type num and RHS is of type num, then \{ + LHS RHS \} is of type num.”

Now comes the first tricky rule—the rule for variables. Recall our evaluation rule for variables:

\[ X; \xi \Rightarrow \xi(X) \]

By keeping track of the environment, we were able to lookup the value of the variable \( X \). Analogously, we will introduce a type environment \( \Gamma \). \( \Gamma \) is a function with contract

\[ \Gamma : \text{var-name} \rightarrow \text{type} \]

We need to rewrite the typing rules we have so far, this time keeping track of \( \Gamma \). We introduce the turn-stile symbol \( \vdash \), which we read “proves”:

\[ \Gamma \vdash n : \text{num} \]

We read this rule, “Gamma proves that \( n \) is of type \( \text{num} \).” It is conventional to put the environment to the left of the turn-stile, rather than adjacent to the expression. This tradition originates from mathematical logic, which contributes much of the foundation for type systems (and indeed most of programming languages). The rules for addition and variables are

\[
\begin{align*}
\Gamma \vdash \text{LHS : num} & \quad \Gamma \vdash \text{RHS : num} \\
\Gamma \vdash \{ + \ \text{LHS} \ \text{RHS} \} : \text{num} \\
\Gamma \vdash \text{X : } \Gamma(\text{X})
\end{align*}
\]

What about Rip procedures? They are just values, so just as we reduced all numeric values to one type, it seems we should reduce all procedural values to a single type also:

\[ \Gamma \vdash \{ \text{proc} \ \{ X \} \ \text{BODY} \} : \text{procedure} \]

Now that we have (an attempt at) a rule for procedure typing, we try to write the procedure application rule:

\[
\begin{align*}
\Gamma \vdash \text{FEXP : procedure} & \quad \Gamma \vdash \text{ARGEXP : } \text{t} \\
\Gamma \vdash \{ \text{FEXP} \ \text{ARGEXP} \} : ?
\end{align*}
\]

There is a problem here. By simply assigning the type \( \text{procedure} \) to a procedure, we’ve lost important information about the procedure. We need to know the type of thing that the procedure consumes, and the type of thing that it returns. We could have our rule for application handle this, but then we would be performing this check every time we used the procedure. Worse, we may not even know which procedure is being used in a particular location without running the program (which we’re trying to avoid).
Therefore, it seems best to build this into our rule for procedures.

We will express the type of a procedure with an \( \rightarrow \), just as we do for contracts (which, we now know, are just unchecked type declarations). For example, consider the Rip procedure \( \{ \text{proc } \{ x \} \{ + \ x \ 2 \} \} \). This is a procedure that consumes a number and returns a number, so its type is \( \text{num} \rightarrow \text{num} \). The Rip procedure

\[
(\{ \text{proc } \{ f \} \\
\quad \{ + \ {f \ 1} \\
\quad \{ f \ 2 \} \})
\]

consumes a procedure \( f \). The procedure \( f \) is applied to 1 and 2, and therefore it must consume numbers. Because the return values of the two \( f \) invocations are added together, \( f \) must return a number. We conclude that \( f \) is of type \( \text{num} \rightarrow \text{num} \). Therefore, the whole procedure has type \( (\text{num} \rightarrow \text{num}) \rightarrow \text{num} \).

We can express what a type is using a data definition:

A type is

1. \( \text{num} \)
2. \( \text{type} \rightarrow \text{type} \)

We would like our type judgments to be able to prove Rip procedures are of a certain type. But consider the procedure \( \{ \text{proc } \{ y \} \ y \} \). The type of this procedure could be \( \text{num} \rightarrow \text{num} \), or it could be \( (\text{num} \rightarrow \text{num}) \rightarrow (\text{num} \rightarrow \text{num}) \). In fact, this procedure could be any of an infinite number of types. We will avoid this problem by forcing the programmer to annotate procedures with type information. We will show how the last few Rip procedures can be annotated with types:

\( \{ \text{proc } \{ y \} \ y \} \) might be annotated \( \{ \text{proc } \{ y : \text{num} \} : \text{num} \ y \} \), and the procedure \( \{ \text{proc } \{ f \} \{ + \ {f \ 1} \ {f \ 2} \} \} \) becomes

\[
\{ \text{proc } \{ f : (\text{num} \rightarrow \text{num}) \} : \text{num} \{ + \ {f \ 1} \ {f \ 2} \} \}
\]

We can now write a correct type judgement for procedures:

\[
\Gamma \vdash \{ \text{proc } \{ X : t1 \}; t2 \ \text{BODY} \} : t1 \rightarrow t2
\]

The rule for procedure application is now easy:

\[
\Gamma \vdash \text{FEXP} : t1 \rightarrow t2 \quad \Gamma \vdash \text{ARGEXP} : t1 \quad \Gamma \vdash \{ \text{FEXP ARGEXP} \} : t2
\]

Let’s use our typing rules to construct a proof that the type of
\{(\text{proc } \text{y} : \text{num} : \text{num} \\
\hspace{1em} (+ \ y \ 5)) \\
\hspace{1em} (+ \ 3 \ 4)\}\}

is \text{num}. The proof looks very similar to the proof from last lecture:

\[
\begin{array}{ccc}
\{y \mapsto \text{num}\} \vdash y : \text{num} & \{y \mapsto \text{num}\} \vdash 5 : \text{num} & \{\} \vdash 3 : \text{num} \\
\hline
\{y \mapsto \text{num}\} \vdash (+\ y\ 5) : \text{num} & \{\} \vdash (+\ 3\ 4) : \text{num} \\
\hline
\{\} \vdash \{\text{proc } \{y : \text{num} : \text{num} (+\ y\ 5)\} : \text{num} \mapsto \text{num} \} \vdash (+\ 3\ 4) : \text{num}
\end{array}
\]