Type Inference, Part I
Lecture Notes for cs173, Fall 2001

sk and rob

November 2, 2001
updated November 6, 2001

Recall our typing rules from previous lectures. These, when arranged in a tree structure, give us a proof of the type of a Rip program. The typing rules do not, however, give us an algorithm for generating the type of an expression. When our language requires procedures be annotated with types, it turns out that an algorithm follows pretty directly from the typing rules. We would like, however, to be able to type check without any programmer annotations—that is, we want an algorithm that generates all types automatically.

Consider the simple Rip program \( f + 12 \). The proof that this program has type \( \text{num} \) is quite simple:

\[
\frac{\Gamma \vdash 1 : \text{num} \quad \Gamma \vdash 2 : \text{num}}{\Gamma \vdash \{+ 1 2\} : \text{num}}
\]

It seems reasonable that an algorithm for type generation would move “bottom-up” in order to generate the type of \( \{+ 1 2\} \). Note that here we are using the computer science sense of the word “bottom-up.” Since computer scientists insist that trees grow downward, any time an algorithm works from the leaves of the tree to the root, the algorithm is a “bottom-up” algorithm. Type judgments come from the world of logic. Logicians have a more sensible view of nature; trees grow upward, and leaves are at the top. Hence, the bottom-up algorithm we speak of goes from the top of the proof tree down.

With respect to the previous proof tree, our mythical algorithm would start at the bottom (proving that \( 1: \text{num} \) and \( 2: \text{num} \)) and move upward to conclude that \( \{+ 1 2\}: \text{num} \).

Now consider the program \( \text{proc} \{x\} \{\text{first} \ x\} \). We won’t use type annotations now, so it’s not clear how to even write the proof tree. Let’s instead write out the skeleton of the proof tree:

---

1If you’re having trouble with this bottom top nonsense, try standing on your head when you look at proof trees.
Let’s “guess” types for $x$ until we find a proof tree that satisfies our typing rules. Assume we guess that $x$ has type $\text{list}(\text{bool})$:

$$
\Gamma \vdash x : \text{list}(\text{bool})
$$

This is a valid proof tree for the program. Now let’s say you hand this program along with its type $(\text{list}(\text{bool}) \rightarrow \text{bool})$ to your friend. Your friend then uses your program inside her own program:

```plaintext
(\text{proc } y)
  (\text{let } (f (\text{proc } x \{\text{first } x\}))
    (+ (f y) 1))
```

If your friend respects the type that you handed her, she will not be able to run this new program. In the new program, the type of the procedure bound to $f$ is $(\text{list}(\text{num}) \rightarrow \text{num})$ because the result of the application of $f$ is added to 1.

What went wrong? Instead of guessing a type for $x$, we would really like to assert that $x$ is of type $\text{list}(\alpha)$, where $\alpha$ is a type variable. The type of the program $\{\text{proc } x \{\text{first } x\}\}$ becomes $\text{list}(\alpha) \rightarrow \alpha$. By using a type variable, when we learn more information about the complete program, we can constrain the types that $\alpha$ can be. In this case, the expression $+ \{f y\} 1$ constrains $\alpha$ to $\text{num}$.

We would like a generic technique for generating the constraints on the types of Rip expressions. We’ll describe the function $[[\bullet]]$, where $\bullet$ represents the argument to the function. The function consumes a Rip expression and returns constraints which need to be satisfied in order for the given expression to type check. This function works “bottom-up”—it is called recursively on subexpressions in order to generate types.

Our language needs to be extended with a grammar for type variables:

$$
\text{Types ::= num | list(Types) | Types \rightarrow Types | TVar}
$$

$$
\text{TVar ::= } \alpha \mid \beta \mid ...
$$

We define $[[\bullet]]$ as follows:

$$
n \mapsto [[n]] = \text{num}
$$

$$
\{m + n\} \mapsto [[\{m + n\}]] = [[m]] = [[n]] = \text{num}
$$

$$
\{\text{cons } e_1 \ e_2\} \mapsto [[\{\text{cons } e_1 \ e_2\}]] = [[e_2]] = \text{list}([[e_1]])
$$

$$
\{\text{first } e\} \mapsto [[e]] = \text{list}([[\{\text{first } e\}]])
$$
\{\text{rest } e\} \mapsto [[\{\text{rest } e\}]] = [[e]] = \text{list}(\alpha), \text{where } \alpha \text{ is a fresh type variable}

\text{empty} \mapsto [[\text{empty}]] = \text{list}(\alpha), \text{where } \alpha \text{ is a fresh type variable}

\{\text{proc } \{x\} \ e\} \mapsto [[\{\text{proc } \{x\} \ e\}]] = ([[x]] \to [[e]])

\{e1 \ e2\} \mapsto [[e1]] = ([[e2]] \to [[\{e1 \ e2\}]]