We’ve developed typing rules which can be used to prove that a Rip program will not produce certain errors at run-time. Specifically, we want to avoid the addition of non-numbers and the application of non-procedures.

Let’s look at a tricky case with types:

```rip
{{proc {x} {x (+ x 1)}}
 <infinite loop>}
```

Let `<infinite loop>` be Rip code that generates an infinite loop. What is the type of the procedure above? Since the same `x` is both added and applied, at least one of these uses of `x` will produce a run-time error, right? Wrong. The infinite loop guarantees that the procedure will never be invoked, so in fact neither of these errors will occur. This program will not produce a run-time error, and so should it pass our type checker?

Infinite loops are hard (impossible, in the general case) to detect. Since we want our type system to be able to prove properties of programs, we will err on the side of conservatism. We will reject this program.

As another tricky example, imagine that we have an `if` construct in Rip. To interpret an `if`, evaluate the first argument, returning the evaluation of the second argument if it is true, and the evaluation of the third argument if it is false.

```rip
{if {magic 25}
  (+ 1 2)
  (1 2))
```

If `{magic 25}` is always true, then this program will never have a type error. Should our checker allow it?

It is reasonable to say that this program should be accepted if it can be proven that the first branch will always be taken. Since the code for `magic` exists somewhere, it may be possible to prove this. But consider a similar example:
Clearly there is absolutely no way of knowing in this example which path will be taken. Hence, we need to be conservative and reject this program, even though it is possible that the false path will never be taken.

Consider Robin Milner’s Theorem, which is often paraphrased as

Well-typed programs do not go wrong.

This paraphrase may sound silly but it conveys a very powerful idea: Type systems allow us to prove that programs don’t contain errors. In the last few examples, we have seen that type issues can be tricky. In general, we will take the conservative path in order to allow us to prove useful things about programs.

In the next few sections, we will add new constructs to Rip, and see how they affect our type system.

1 Booleans in Rip

Let’s add booleans to our language. We start by adding a new type, bool, and add its associated typing rules:

\[
\begin{align*}
\Gamma \vdash true : bool \\
\Gamma \vdash false : bool \\
\Gamma \vdash \text{TEST_EXPR} : ??? \\
\Gamma \vdash \text{THEN_EXPR} : ??? \\
\Gamma \vdash \text{ELSE_EXPR} : ??? \\
\Gamma \vdash \text{if TEST_EXPR THEN_EXPR ELSE_EXPR} : ???
\end{align*}
\]

What should we put in place of the question marks? If we follow the lead of Scheme, we can put any type anywhere we want:

\[
\begin{align*}
\Gamma \vdash \text{TEST_EXPR} : t_1 \\
\Gamma \vdash \text{THEN_EXPR} : t_2 \\
\Gamma \vdash \text{ELSE_EXPR} : t_3 \\
\Gamma \vdash \text{if TEST_EXPR THEN_EXPR ELSE_EXPR} : ???
\end{align*}
\]

Note that \(t_1\), \(t_2\), and \(t_3\) are variables we use in the judgments to stand in the place of actual types. The variable \(t_1\) may be bound to a type as simple as num, or something more complicated, like (num → num) → num.

What should we fill in for the question marks below the line? We only know that the type of the if expression may be \(t_1\) or \(t_2\). That sounds pretty rough; everyone who gets this type then needs to deal with two possibilities for a type. This is messy (actually, it’s a research problem), and so we will make the (very reasonable) assertion that the then and else clause of the if must be of the same type. We further assert that the type of the TEST_EXPR must be bool. Our rule for if is therefore


\[
\Gamma \vdash \text{TEST_EXPR} : \text{bool} \quad \Gamma \vdash \text{THEN_EXPR} : t \quad \Gamma \vdash \text{ELSE_EXPR} : t \quad \Gamma \vdash \{\text{if TEST_EXPR THEN_EXPR ELSE_EXPR}\} : t
\]

2 \hspace{1em} \text{Pairs}

Let’s try to add pairs to Rip. First, let’s give the operational semantics for \text{pair}. We will use mathematical tuples to represent the value of Rip pairs:

\[
E_1;\xi \Rightarrow v_1 \quad E_2;\xi \Rightarrow v_2
\]

\[
\{\text{pair } E_1 \ E_2\};\xi \Rightarrow (v_1,v_2)
\]

We need a way to extract the pieces from a pair. We use \(\pi_1\) and \(\pi_2\) (just like Scheme’s \text{car} and \text{cdr}) for this:

\[
e;\xi \Rightarrow (v_1,v_2)
\]

\[
\{\pi_1 \ E\};\xi \Rightarrow v_1
\]

We add a new pair type to Rip, and our full grammar for types in Rip becomes

An RT is:

1. \text{num}
2. \(\text{RT} \rightarrow \text{RT}\)
3. \(\langle \text{RT}, \text{RT} \rangle\)

Let’s write the typing rule for \text{pair}:

\[
\Gamma \vdash E_1 : t_1 \quad \Gamma \vdash E_2 : t_2
\]

\[
\Gamma \vdash \{\text{pair } E_1 \ E_2\} : \langle t_1,t_2 \rangle
\]

which we can write in a simplified form as

\[
\text{pair} : t_1 \times t_2 \rightarrow \langle t_1,t_2 \rangle
\]

Similarly, the projections’ simplified typing rules are

\[
\pi_i : \langle t_1,t_2 \rangle \rightarrow t_i \quad \text{for } i = 1, 2
\]

Some systems build lists out of pairs. For instance, to represent the equivalent of the Scheme list \(\text{list} \ 1 \ 2 \ 3 \ 4\) in Rip, a programmer might write

\[
\{\text{pair } 1 \ \{\text{pair } 2 \ \{\text{pair } 3 \ 4\}\}\}
\]

Suppose we want to write the procedure \text{third}, which returns the third element of this list. As an example, the third element of the list above is 3. What would the type of \text{third} be?
third : (num, (num, (num, num))) → num

This works if we use the list \{pair 1 \{pair 2 \{pair 3 4\}\}\}, but what if we want to find the third element of the list \{pair 1 \{pair 2 \{pair 3 \{pair 4 5\}\}\}\}? Since the type of this datum is \(\text{num}, \text{num}, \text{num} \langle\text{num}, \text{num}\rangle\), we can’t even feed it to \text{third} because it will not type check. In fact, we can’t write any useful generic list procedures.

This takes us back to what we said at the beginning of cs173: don’t ever mix up the uses of structures and lists. A structure (of which a pair is a special case) should be used for representing a fixed number of heterogeneous data. A list is for representing an arbitrary number of homogeneous data. Never the twain shall meet.

3 Lists

In the spirit of not confusing lists and structs, let’s make \text{nlist} (number lists) a primitive type in Rip. We will also add the following primitives:

\text{cons} : \text{num} × \text{nlist} → \text{nlist}

\text{first} : \text{nlist} → \text{num}

\text{rest} : \text{nlist} → \text{nlist}

We also need to add the value \text{empty} to Rip, which has type \text{nlist}. Note that we can now write a \text{third} procedure which will pass our type checker. It has the same typing rule as \text{first}.

But we still have problems. Consider the Rip program

\{first empty\}

The type of \text{empty} is \text{nlist}, and the type of \text{first} is \text{nlist} → \text{num}. Therefore, our type checker will allow this program, assigning to the program the type \text{num}. Yet the program clearly will not produce a \text{num}—it will produce a run time error! Uh oh.

We’ll deal with this in the next class.