Well-typed programs do not go wrong
—paraphrase of Robin Milner's type soundness theorem

1 Type Semantics Versus Run-Time Semantics

What does a type system do? It statically approximates dynamic behavior. That is, it “runs” the program over a very limited set of values—so limited that programs always terminate, but not so limited that we can’t get any useful information. For instance, a type system treats all functions that consume a number and return a number as one function of type \( \text{num} \rightarrow \text{num} \), irrespective of whether they compute the square or the square root of their argument. On the other hand, a type system distinguishes between all those functions and all functions that consume a number and return another function.

If a type system runs a program over the set of types, and then again over the set of actual values, we need to be pretty careful. We don’t want the two runs to return inconsistent results. (Note: They can’t return the same results, because the value sets are different; what we want is not equality, only consistency.) For example, consider the program \( P \):

\[
\begin{align*}
\text{(if } & \text{<condition>} \text{)} \\
\{ & \text{(proc } x : \text{num} : \text{num }(+ \text{ 1 } x) \text{ 3) ; C1)} \\
\{ & \text{(proc } x : \text{num} : \text{num }(+ \text{ 1 } (+ \text{ 2 } x)) \text{ 3) ) ; C2)}
\end{align*}
\]

The proof that \( P : \text{num} \) contains a judgment tree for each clause of the if. The proof that \( P \) reduces to some value \( v \), depends on the truth of the <condition>; the proof will only contain the computation tree of the clause that is actually evaluated at runtime.

While the value \( v \) certainly will be consistent with the proven type \( \text{num} \), the two systems arrive at their conclusions through differently shaped proofs. If the function were recursive, the divergence would be even greater, because the recursive function would evaluate multiple times in the semantics, while it would be checked only once by the
type checker.

We conclude that the type system isn’t a faithful reproduction of the computation. Is this a problem? It turns out that in order to make our type system useful, we need to relate the run-time semantics with the type semantics. Slightly more precisely, for a given program we would like to verify that the following property is satisfied:

\[
\text{if the program passes the type checker, then the program doesn’t error during execution.}
\]

You should note two things about this conditional:

1. An “error,” as referred to above, is one of a very restricted set of execution errors, namely application of non-functions and addition of non-numbers. A function that \textit{should} square a number, but incorrectly cubes it instead, is outside the purview of our type system.

2. The property above has an implication in one direction, but not the other; it says \textit{if}, not \textit{if and only if}. That reflects the inherent weakness of type systems. Because of the Halting Problem, there will always be programs that don’t error during execution whose correctness we can’t ascertain statically.

We now begin to appreciate the subtlety, utility and overall value of the paraphrase of Milner’s theorem, “Well typed programs do not go wrong.” Let’s try to formulate it more precisely:

\textbf{Version 1 (Type Soundness)} \( \forall P \in RP, \text{ if } \{ \} \vdash P : \tau, \text{ then } P; \{ \} \Rightarrow \tau \)

This can’t be right. The evaluator \( \Rightarrow \) computes values, not types. (There’s a type error in the statement!) What we really mean is

\textbf{Version 2 (Type Soundness)} \( \forall P \in RP, \text{ if } \{ \} \vdash P : \tau, \text{ then } P; \{ \} \Rightarrow v, \text{ where } v : \tau \)

That is, the program evaluates to a value \( v \), and \( v \) (by prior agreement) falls in the set of values whose type is \( \tau \). The implication is, if the program yielded a value, it couldn’t have gone wrong along the way. (The paraphrase of Milner’s Theorem is somewhat weaker than the statement above; the paraphrase only says that a program that passes the type checker won’t manifest an error, not that it will produce a value of the right type. In practice, when trying to actually prove this theorem, the above form proves to be inevitable. This is just a strengthened inductive hypothesis.)

We sometimes also call this property \textit{type soundness}, in tribute to the logical roots of computer science. Type soundness is a result that relates \textit{two} entities: the type system and a corresponding evaluator. It says that a type system is sound with respect to that evaluator if programs typed under the type system will produce a value of the right type when evaluated. All modern languages (should) have sound type systems.
2 Run-Time Errors

This is all fine, but what about the question we left open at the end of the last lecture: What about programs like \{first empty\} or \{/ 1 0\}? The Rip program \{first empty\} has type num, but during run-time will halt with an error. This clearly violates the statement of type soundness because it satisfies the premise but not the consequence.

What should happen when we evaluate \{first empty\}? It currently returns an error from the underlying Scheme system, but it’d be nicer to specify its behavior explicitly rather than rely on the language of the meta-interpreter. We could do several things:

1. Return an error value, say -1. We hope you cringe at this suggestion. This is a terrible idea. It means a program like \{+ 42 \{first empty\}\}, which should halt with an error, will evaluate to 41—and the user is blissfully unaware that the answer is meaningless. Ignorance is bliss.

2. Go into an infinite loop. Notice that this seems to sort of duck the consequent of Milner’s Theorem—or does it? Either way, it’d be a pretty difficult language to use and debug. Not to mention, we wouldn’t be able to distinguish different kinds of errors. A bad idea all round, but we’ll come back to it briefly later.

3. Throw an exception. This is clearly the right thing to do, since it explicitly flags the occurrence of an error, and yet gives the programmer the choice of whether he wants to do anything about it.

We can actually state quite precisely when we’d throw an exception: it’s when a function is partial over its declared type domain. For instance, first and rest are partial over the set of rlist’s; they would throw exceptions when invoked on empty lists. Addition is a total function, so it throws no exceptions. Division is partial (it isn’t defined when the second argument is zero) so it throws an exception when asked to divide by zero. By examining each function, we can arrive at a list of acceptable exceptions.

With this context, we now rewrite the Type Soundness property:

\[
\text{Version 3 (Type Soundness)} \quad \forall P \in RP, \text{ if } \{ \} \vdash P : \tau, \text{ then either } P;\{ \} \Rightarrow v, \text{ where } v : \tau \text{ or evaluating } P \text{ yields one of the pre-declared exceptions.}
\]

Is this a cop-out, or what? Yes and no. It’s definitely a bit unsatisfying, and somewhat reduces the grandeur of the theorem. But it isn’t a cop-out in the sense that it’s another example of bumping up against what isn’t tractable. There do exist type systems that catch some of the errors designated as exceptions above. These consume a lot more computational resource, and/or operate on much more restricted languages. For such a language, Milner’s Theorem moves more properties into the purview of the type system, and fewer into the exceptions clause.
3 An Infinite Loop in Rec-less Rip

Let’s try to write an infinite loop in Rip. Based on the lambda calculus lecture, it should now be pretty clear how we can write one:

\[
\{\text{proc} \ (x) \ (x \ x)\} \\
\{\text{proc} \ (x) \ (x \ x)\}
\]

If you substitute the argument in the body, you’ll quickly see why this results in an infinite loop. Now let’s write this program in typed Rip. We’d have to write

\[
\{\text{proc} \ (x : T1) \ (x \ x)\} \\
\{\text{proc} \ (x : T2) \ (x \ x)\}
\]

T1 and T2 are type variables—not valid Rip code. We need to decide what to write in their place. Let’s consider just the first sub-expression. What is the structure of T1? Clearly, x must be a procedure, because the body applies x to an argument. Therefore,

\[T1 = (T2 -> T3)\]

Because the argument to x is itself x, this means the argument type, T2, must be equal to the type of the procedure itself:

\[T2 = T1\]

This immediately implies

\[T1 = (T1 -> T3)\]

which is an infinite type. We clearly can’t solve this equation in our type language. Hence, we can’t write this expression in typed Rip.

We can, however, turn this problem around in order to obtain a proof that no typed Rip program can enter an infinite loop. We provide some intuition for this proof: Any infinite loop must involve functions, and each function application erases an arrow. Since a type can have only a finite number of arrows, we must eventually exhaust them all.

This isn’t by any stretch a formal proof, but we can formalize this intuition to prove that typed Rip normalizes. That is, every program reduces to a normal form, or a value. In short, no typed Rip program can execute forever.

What good is this property? Actually, many programs do (or, at least, should) exhibit this property. For example,

1. languages for real-time systems
2. program linkers
3. packet filters in network stacks
4. network routers
5. configuration files for applications

When we design such a language, we would do well to consider imposing a type system of the form we've studied so far atop the language. This would guarantee that no program could execute forever. Sometimes these guarantees are merely a convenience, but sometimes, the task may be mission-critical. At those times, it sure would be nice to have a formal proof about behaviors that programs in the language cannot exhibit. In short, this type system has the side-effect of preventing one very common error, namely an accidental infinite loop.

4 Adding Rec to Typed Rip

If we add rec to typed Rip, we will be able to write infinite loops. Let’s write the typing rule for rec:

\[
\Gamma \left[ \text{VAR} \rightarrow \text{T} \right] \vdash \text{T} \quad \Gamma \left[ \text{VAR} \rightarrow \text{T} \right] \vdash \text{BODY} : \tau \\
\Gamma \vdash \{ \text{rec} \left[ \text{VAR} : \text{T} \text{ VAL} \right] \text{BODY} \} : \tau
\]

Note that in order to assign a type to a rec expression, we need to prove VAL is of type \( \text{T} \) in the extended environment in which VAR is mapped to the same \( \text{T} \). This is the reason we are now able to get recursion in typed Rip.

An example of an infinite loop in Rip is

\[
\{ \text{rec} \left[ \text{f} : \text{(num } \rightarrow \text{ num)} \right] \{ \text{proc} \left[ \text{x } : \text{ num} \right] : \text{ num } \{ \text{f x} \} \}}
\{ \text{f 1} \}
\]

Because non-termination is now a possibility in our language, we need to re-write the type soundness property one last time:

**Version 4 (Type Soundness)** \( \forall P \in RP, \text{ if } \{ \} \vdash P : \tau \text{ and } P \text{ terminates, then either } P : \{ \} \Rightarrow v, \text{ where } v : \tau \text{ or evaluating } P \text{ yields one of the pre-declared exceptions.} \)

As mentioned earlier, some programming languages theorists use an infinite loop to represent an error condition. We can now understand the motivation for this: Since we need a disclaimer about program termination in the statement of type soundness anyway, by mapping errors to infinite loops, we wouldn’t need to throw in the clause about exceptions. This is, however, an extremely poor engineering choice. We will adopt the design philosophy that we shouldn’t pervert the programming methodology simply to make the theory a bit cleaner; indeed, we often need to do exactly the opposite. Hence, we will continue to reject this option.