Chapter 28

Type Soundness

We would like a guarantee that a program that passes a type checker will never exhibit certain kinds of errors when it runs. In particular, we would like to know that the type system did indeed abstract over values: that running the type checker *correctly predicted* (up to the limits of the abstraction) what the program would do. We call this property of a type system *type soundness*:[1]

> For all programs p, if the type of p is τ, then p will evaluate to a value that has type τ.

Note that the statement of type soundness *connects types with execution*. This tells the user that the type system is not some airy abstraction: what it predicts has bearing on practice, namely on the program's behavior when it eventually executes.

We have to be more careful about how we define type soundness. For instance, we say above (emphasis added) "*p will* evaluate to a value such that ...". But what if the program doesn't terminate? So we must recast this statement to say

> For all programs p, if the type of p is τ and p evaluates to v, then $v : \tau$.[2]

Actually, this isn't quite true either. What if the program executes an expression like (*first empty*)? There are a few options open to the language designer:

- Return a value such as -1. We hope you cringe at this idea! It means a program that fails to properly check for return values at every single place will potentially produce nonsensical results. (Such errors are common in C programs, where operators like `malloc` and `fopen` return special values but programmers routinely forget to check them. Indeed, many of these errors lead to expensive, frustrating and threatening security violations.)

- Diverge, i.e., go into an infinite loop. This approach is used by theoreticians (study the statement of type soundness carefully and you can see why), but as software engineers we should soundly (ahem) reject this.

[1] The term "soundness" comes from mathematical logic.

[2] We could write this more explicitly as: 'For all programs p, if the type checker assigns p the type τ, and if the semantics says that p evaluates to a value v, then the type checker will also assign v the type τ."

- Raise an exception. This is the preferred modern solution.

Raising exceptions means the program does not terminate with a value, nor does it not terminate. We must therefore refine this statement still further:

> For all programs p, if the type of p is τ, p will, if it terminates, either evaluate to a value v such that $v : \tau$, or raise one of a well-defined set of exceptions.

The exceptions are a bit of a cop-out, because we can move arbitrarily many errors into that space. In Scheme, for instance, the trivial type checker rejects no programs, and all errors fall under the exceptions. In contrast, researchers work on very sophisticated languages where some traditional actions that would raise an exception (such as violating array bounds) instead become type errors. This last phrase of the type soundness statement therefore leaves lots of room for type system design.

As software engineers, we should care deeply about type soundness. To paraphrase Robin Milner, who first proved a modern language's soundness (specifically, for ML),

Well-typed programs do not go wrong.

That is, a program that passes the type checker (and is thus "well-typed") absolutely cannot exhibit certain classes of mistakes. [3]

Why is type soundness not obvious? Consider the following simple program (the details of the numbers aren't relevant):

```
{if0 {+ 1 2}
     {{fun {x : number} : number {+ 1 x}} 7}
     {{fun {x : number} : number {+ 1 {+ 2 x}}} 1}}
```

During execution, the program will explore only one branch of the conditional:

$$\dfrac{\dfrac{\texttt{1},\emptyset \Rightarrow \texttt{1} \quad \texttt{2},\emptyset \Rightarrow \texttt{2}}{\texttt{\{+ 1 2\}},\emptyset \Rightarrow \texttt{3}} \quad \dfrac{\cdots}{\texttt{\{\{fun ...\} 1\}},\emptyset \Rightarrow \texttt{4}}}{\texttt{\{if0 \{+ 1 2\} \{\{fun ...\} 7\} \{\{fun ...\} 1\}\}},\emptyset \Rightarrow \texttt{4}}$$

but the type checker must explore both:

$$\dfrac{\dfrac{\emptyset \vdash \texttt{1} : \texttt{number} \quad \emptyset \vdash \texttt{2} : \texttt{number}}{\emptyset \vdash \texttt{\{+ 1 2\}} : \texttt{number}} \quad \dfrac{\cdots}{\emptyset \vdash \texttt{\{\{fun ...\} 7\}} : \texttt{number}} \quad \dfrac{\cdots}{\emptyset \vdash \texttt{\{\{fun ...\} 1\}} : \texttt{number}}}{\emptyset \vdash \texttt{\{if0 \{+ 1 2\} \{\{fun ...\} 7\} \{\{fun ...\} 1\}\}} : \texttt{number}}$$

Furthermore, even for each expression, the proof trees in the semantics and the type world will be quite different (imagine if one of them contains recursion: the evaluator must iterate as many times as necessary to produce a value, while the type checker examines each expression only once). As a result, it is *far from*

[3] The term "wrong" here is misleading. It refers to a particular kind of value, representing an erroneous configuration, in the semantics Milner was using; in that context, this slogan is tongue-in-cheek. Taken out of context it is misleading, because a well-typed program can still go wrong in the sense of producing erroneous output.

obvious that the two systems will have any relationship in their answers. This is why a theorem is not only necessary, but sometimes also difficult to prove.

Type soundness is, then, really a claim that the type system and run-time system (as represented by the semantics) are in sync. The type system erects certain abstractions, and the theorem states that the run-time system mirrors those abstractions. Most modern languages, like ML and Java, have this flavor.

In contrast, C and C++ *do not have sound type systems*. That is, the type system may define certain abstractions, but the run-time system does not honor and protect these. (In C++ it largely does for object types, but not for types inherited from C.) This is a particularly insidious kind of language, because the static type system lulls the programmer into thinking it will detect certain kinds of errors, but it fails to deliver on that promise during execution.

Actually, the reality of C is much more complex: C has *two different type systems*. There is one type system (with types such as `int`, `double` and even function types) at the level of the program, and a different type system, defined *solely* by lengths of bitstrings, at the level of execution. This is a kind of "bait-and-switch" operation on the part of the language. As a result, it isn't even meaningful to talk about soundness for C, because the static types and dynamic type representations simply don't agree. Instead, the C run-time system simply interprets bit sequences according to specified static types. (Procedures like `printf` are notorious for this: if you ask to print using the specifier `%s`, `printf` will simply print a sequence of characters until it hits a null-teriminator: never mind that the value you were pointing to was actually a double! This is of course why C is very powerful at low-level programming tasks, but how often do you actually need such power?)

To summarize all this, we introduce the notion of *type safety*:

> Type *safety* is the property that no primitive operation ever applies to values of the wrong type.

By primitive operation we mean not only addition and so forth, but also procedure application. *A safe language honors the abstraction boundaries it erects*. Since abstractions are crucial for designing and maintaining large systems, safety is a key software engineering attribute in a language. (Even most C++ libraries are safe, but the problem is you have to be sure no legacy C library isn't performing unsafe operations, too.) Using this concept, we can construct the following table:

	statically checked	not statically checked
type safe	ML, Java	Scheme
type unsafe	C, C++	assembly

The important thing to remember is, due to the Halting Problem, some checks simply can never be performed statically; something must always be deferred to execution time. The trade-off in type design is to maximize the number of these decisions statically without overly restricting the power of the programmer. The designers of different languages have divergent views on the powers a programmer should have.

So what is "strong typing"? This appears to be a meaningless phrase, and people often use it in a nonsensical fashion. To some it seems to mean "The language has a type checker". To others it means "The language is sound" (that is, the type checker and run-time system are related). To most, it seems to just mean, "A language like Pascal, C or Java, related in a way I can't quite make precise". If someone uses this phrase, be sure to ask them to define it for you. (For amusement, watch them squirm.)