What is type safety?

In response to my previous post defining memory safety (for C), one commenter suggested it would be nice to have a post explaining type safety. Type safety is pretty well understood, but it's still not something you can easily pin down. In particular, when someone says, "Java is a type-safe language," what do they mean, exactly? Are all type-safe languages "the same" in some way? What is type safety getting you, for particular languages, and in general?

In fact, what type safety means depends on language type system's definition. In the simplest case, type safety ensures that program behaviors are well defined. More generally, as I discuss in this post, a language's type system can be a powerful tool for reasoning about the correctness and security of its programs, and as such the development of novel type systems is a rich area of research.

Basic Type Safety

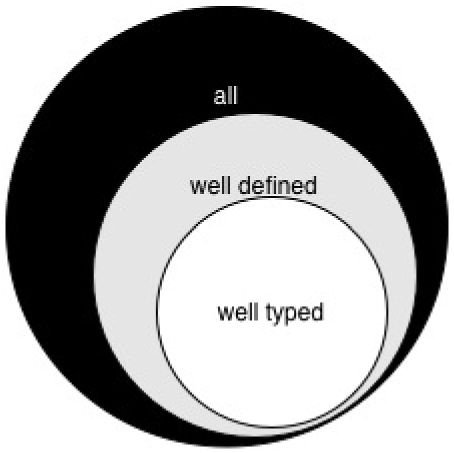
An intuitive notion of type safety is pithily summarized by the phrase, "***Well typed*** *programs* ***cannot go wrong***." This phrase was coined by Robin Milner in his 1978 paper, A Theory of Type Polymorphism in Programming. Let's deconstruct this phrase and define its parts, considering the second part first.

Going wrong

Programming languages are defined by their *syntax* — what programs you're allowed to write down — and *semantics* — what those programs mean. A problem all languages face is that there are many programs that are syntactically valid but are semantically problematic. A classic English example is Chomsky's "Colorless green ideals sleep furiously"—perfectly syntactically correct, but meaningless. A example in the OCaml programming language is **1 + "foo"**; according to the language's semantics, such a program has no meaning. Another example is **{ char buf[4]; buf[4] = 'x' }** in C: the write to index 4 is outside the declared bounds of the buffer, and the language specification deems this action to be undefined, i.e., meaningless. If we were to run such meaningless programs, we could say that they would *go wrong*.

Well typed ⟹ Cannot go wrong

In a type-safe language, the language's *type system* is a particular way of ensuring that only the "right" (non-wrong) programs go through. In particular, we say a program (or program phase) is *well typed* if the type system deems it acceptable, and type safety ensures that well typed programs never go wrong: they will have a (well defined) meaning. The following picture visualizes what's going on.



In a type safe language, well typed programs are a subset of the well defined programs, which are themselves a subset of all possible (syntactically valid) programs.

Which languages are type safe?

Let’s now consider whether some popular languages are type safe. We will see that for different languages, type safety might mean different things.
C and C++: *not* type safe. C’s standard type system does not rule out programs that the standard (and common practice) considers meaningless, e.g., programs that write off the end of a buffer. [1] So, for C, well typed programs can go wrong. C++ is (morally) a superset of C, and so it inherits C’s lack of type safety.
Java, C#: type safe (probably). While it is quite difficult to ascertain whether a full-blown language implementation is type safe (e.g., an early version of Java generics was buggy), formalizations of smaller, hopefully-representative languages (like Featherweight Java) are type safe. [2] Interestingly, type safety hinges on the fact that behaviors that C’s semantics deem as undefined, these languages give meaning to. Most notably, a program in C that accesses an array out of bounds has no meaning, but in Java and C# it does: this program will throw an *ArrayBoundsException*.
Python, Ruby: **type safe** (arguably). Python or Ruby are often referred to as *dynamically typed languages*, which throw exceptions to signal type errors occurring during execution: Just like Java throws an *ArrayBoundsException* on an array overflow at run-time, Ruby will throw an exception if you try to add an integer and a string. In both cases, this behavior is prescribed by the language semantics, and therefore the programs are well defined. In fact, the language semantics gives meaning to all programs, so the *well defined* and *all* circles of our diagram coincide. As such, we can think of these languages as type safe according to the *null* type system, [3] which accepts all programs, none of which go wrong (making all three circles coincide). Hence: type safety.
This conclusion might seem strange. In Java, if program **o.m()** is deemed well typed, then type safety ensures that **o** is an object that has a no-argument method **m**, and so the call will always succeed. In Ruby, the same program **o.m()** is *always* deemed well typed by Ruby’s (null) type system, but when we run it, we have no guarantee that **o** defines the method **m**, and as such either the call will succeed, or it will result in an exception.
In short, type safety does not mean one thing. What it ensures depends on the language semantics, which implicitly defines wrong behavior. In Java, calling a nonexistent method is wrong. In Ruby, it is not: doing so will simply produce an exception. [4]

Beyond the Basics

Generic type safety is useful: without it, we have no guarantee that the programs we run are properly defined, in which case they could do essentially anything. C/C++’s allowance of undefined behavior is the source of a myriad security exploits, from stack smashing to format string attacks.
Such exploits are not possible in type safe languages.
On the other hand, as our discussion above about Ruby and Java illustrates, not all type systems are created equal: some can ensure properties that others cannot. As such, we should not just ask whether a language is type safe; we should ask what type safety actually buys you. We’ll finish out this post with a few examples of what type systems can do, drawing from a rich landscape of ongoing research.

Narrowing the gap

The *well typed* and *well defined* circles in our diagram do not match up; in the gap between them are programs that are well defined but the type system nevertheless rejects. As an example, most type systems will reject the following program:

```
if (p) x = 5;
  else x = "hello";
if (p) return x+5;
```

```
    else return strlen(x);
```

This program will always return an integer, but the type system may reject it because the variable **x** is used as both an **int** and a **string**. Drawing an analogy with static analysis, as discussed previously in the context of the Heartbleed bug, type systems are *sound* but *incomplete*. This incompleteness is a source of frustration for programmers, (and may be one reason driving them to use dynamic languages like Python and Ruby). One remedy is to design type systems that narrow the gap, accepting more programs.

As an example if this process in action, Java's type system was extended in version 1.5 with the notion of *generics*. Whereas in Java 1.4 you might need to use a cast to convince the type system to accept a program, in Java 1.5 this cast might not be needed. As another example, consider the lambda calculus, the basis of functional programming languages. The simple type system for the lambda calculus accepts strictly fewer programs than Milner's polymorphic type system, which accepts fewer programs than a type system that supports Rank-2 (or higher) polymorphism. Designing type systems that are expressive (more complete) *and* usable is a rich area of research.

Enforcing invariants

Typical languages have types like **int** and **string**. Type safety will ensure that a program expression that claims to be an **int** actually is: it will evaluate to -1, 2, 47, etc. at run-time. But we don't need to stop with **int**: A type system can support far richer types and thereby express more interesting properties about program expressions.

For example, there has been much recent interest in the research community in *refinement types*, which refine the set of a type's possible values using logical formulae. The type **{v: int | 0 <= v}** refines the type **int** with formula **0 <= v**, and in effect defines the type of non-negative integers. Refinement types allow programmers to express data structure invariants in the types of those data structures, and due to type safety be assured that those invariants will always hold. Refinement type systems have been developed for Haskell and F# (called Liquid Haskell and F7, respectively), among other languages.

As another example, we can use a type system to ensure data race freedom by enforcing the invariant that a shared variable is only ever accessed by a thread that holds the lock that guards that variable. The type of a shared variable describes the locks that protect it. Types for safe locking was first proposed by Abadi and Flanagan, and implementations have been developed for Java and C (Locksmith).

There are many other examples, with type systems designed to restrict the use of tainted data, to prevent the release of private information, to ensure object usage follows a strict protocol called *type state* (e.g., so that an object is not used after it is freed), and more.

Type abstraction and information hiding

Many programming languages aim to allow programmers to enforce *data abstraction* (sometimes called *information hiding*). These languages permit writing abstractions, like classes or modules or functions, that keep their internals hidden from the client code that uses them. Doing so leads to more robust and maintainable programs, because the internals can be changed without affecting the client code.

Type systems can play a crucial role in enforcing abstraction. For example, with the aid of a well designed type system, we can prove representation independence, which says that programs only depend on the behavior of an abstraction, not on the way it is implemented. Type-enforced abstraction can also be an enabler of free theorems, as Wadler elegantly showed. The late John Reynolds did groundbreaking work on types and abstraction, most notably described in his 1983 paper, Types, Abstraction, and Parametric Polymorphism. In this paper he famously states "*type structure is a syntactic discipline for maintaining levels of abstraction*," positing that types have a *fundamental* role in building maintainable systems.

Parting thought

Type safety is an important property. At the least, a type safe language guarantees its programs are well defined. This guarantee is necessary for reasoning about what programs might do, which is particularly important when security is a concern. But type systems can do more, forming a foundation for reasoning about programs, ensuring that they enforce invariants, and maintain abstractions. As software is becoming ever more pervasive and more complicated, type systems are an important tool for ensuring the systems we rely on are trustworthy and secure.

If you are interested in learning more about type systems, I'd recommend starting with Benjamin Pierce's books, Types and Programming Languages, and Advanced Topics in Types and Programming Languages.

Thanks to Matthew Hammer, Jeff Foster, and David Van Horn for comments and suggestions on drafts of this post.

Notes:

1. C is also not memory safe; in effect, the undefined behaviors that memory safety rules out are a subset of the undefined behaviors ruled out by type safety.
2. While full-scale languages rarely receive the same attention as smaller *core calculi* that represent them, in 1997, Standard ML (SML) was formally specified — both its semantics and type system — and proven to be type safe. Later work mechanized the metatheory of SML, adding further assurance to this result. This was landmark work to prove that a full-scale language was indeed type safe.
3. Sometimes languages like Python and Ruby are characterized as *uni-typed*, meaning that for the purposes of type checking, all objects in the language have one type that accepts all operations, but these operations may fail at run-time. Programmers may not think of the languages this way, though.
4. You could imagine building an alternative type system to rule out some of Ruby's run-time errors; indeed that's what the Diamondback Ruby project tries to do.