CPSC 311: Types **(DRAFT)**
("lec-types")

Joshua Dunfield
University of British Columbia

October 6, 2016

> "You must always ask yourself: What kind of an animal is it? Is it a function? Is it a set?"
>
> —Prof. Maria Balogh

1 Topics discussed

- why use types?
  - stop bad things from happening
  - make sure that good things will happen
- classifying errors
- catching errors statically vs. dynamically
- typed vs. untyped languages; safe vs. unsafe languages:
- catching bugs with Haskell's type checker; not catching bugs with Haskell's type checker
- refined type systems
- object-oriented languages
- disadvantages of typed languages

2 "Static" vs. "dynamic"

A language consists of syntax and semantics. Semantics consists of dynamic semantics (perhaps defined using a big-step semantics $e \Downarrow v$ or a small-step semantics $e1 \longrightarrow e2$) and static semantics.

What makes something static or dynamic?

A simplistic model—which mostly made sense in the 1960s—is that "static" means "at compile time", and "dynamic" means "at run time". If our language is implemented by an interpreter, rather than a compiler, this model becomes:

- "static" means "before running the program", and
- "dynamic" means "while running the program".

So in our Fun implementations, "static" would mean "before starting `interp`", and "dynamic" would mean "while running `interp`".

Under this model, we can say that, in Fun:

- syntax checking is static (because it happens inside `parse`, which runs before `interp`); and
- arithmetic is dynamic because numbers are added and subtracted inside `interp`, but not inside `parse`).

Besides arithmetic, many other operations are dynamic, such as function calls and Pair-case.

Most language implementations, especially highly engineered compilers, muddy this model. For example, any "serious" C compiler, given code that looks like

```
i += (15 * 1024);
```

will generate the same machine code as it would for

```
i += 15360;
```

The compiler knows that the result of `15 * 1024` cannot change, so it does the multiplication *at compile time*.

Similarly, I would expect that good C compiler would[1] generate the same code from

```
i += 1024;
i += 1024;
```

as from

```
i += 2048;
```

The compiler knows that adding 1024 twice is the same as adding 2048 once.

```
i += 1024;
if (i == 0) {
  i += 1024;
} else {
  i += 1024;
}
```

I would also expect a good compiler to optimize the above code: the compiler would analyze the "then" and "else" branches, see that they are doing the same thing, and generate code that doesn't test `i` at all. This is a form of *static analysis*.

3 Prevention

The most important kind of static semantics is *typing*, sometimes known as *static typing*. We'll see later that typing can be defined through rules.

What's the point of typing? I know two good answers to this question. I like one of these answers better, but first I'll give the more popular answer.

Safety: Typing stops bad things from happening, by telling you that they could happen, and not letting you run your program.

[1] Subject to certain conditions; if I remember correctly, if `i` is declared to be `volatile`, the additions must be done separately.

3.1 Errors: a renewable resource

What kinds of bad things can happen in programs? As you know, there are many such animals.

- **Syntax errors**: missing ")", missing semicolon, missing keywords, extra keywords, "illegal string literals", ...
- **Scope errors**: "unbound identifier", "unknown variable", "duplicate definition for identifier", ...

Anything that doesn't match the language's BNF is a syntax error. A program with a scope error matches the BNF, so it's (usually?) not considered a syntax error.

Syntax and scope errors are pretty universal in programming languages. Other kinds of errors depend on the language; a language without arrays, for example, won't have array bounds errors.

- **Agreement errors and "mismatches"**: The terminology, and the specific error messages, for these errors depend very much on the language; here are some examples:

```
- 3 + "a";
stdIn:1.1-1.8 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * string
  in expression:
    3 + "a"
```

```
> (+ 3 "a")
+: contract violation
  expected: number?
  given: "a"
  argument position: 2nd
  other arguments...:
   3
```

```
> ((lambda (x) x) 'a 'b)
⊗ #<procedure>: arity mismatch;
 the expected number of arguments does not match the given number
  expected: 1
  given: 2
  arguments...:
   'a
   'b
```

These are sometimes called *type errors*, but I would like to use that term in a more specific way, so I'll try to avoid using it for now.

- **Array bounds error**: trying to access an element outside an array.
- **Not returning anything**: writing a procedure that's meant to return a value, but doesn't (happens to me in Python).

Each language gets to decide what counts as an error. For example, writing `"a" + "b"` is not an error in Python, but `(+ "a" "b")` is an error in Racket. Writing `(+ 1 0.5)` is not an error in Racket, but `1 + 0.5` is an error in SML (it doesn't let you mix integers and floats).

Division by zero is almost always an error—but if you really wanted to, and if you have no respect for algebra, you could define a language in which division by zero returned zero.

Integer overflow is an error in many languages, but not all. Some earlier lecture notes discussed C's overflow behaviour; to summarize, overflow on C's `unsigned int` is supposed to "wrap around", but C doesn't specify what should happen if you overflow on a (signed) `int`. We could reasonably say that a C program that overflows a signed `int` has a bug, since the definition of C doesn't specify what that program will do, but C does not specify that it is an error.

An error is a failure that is somehow *caught* and reported, though not necessarily reported in a clear or helpful way. Thus, in Python, not `return`ing from a function is not an error: the function will return `None`. This is well-defined, but not what I wanted to do. In C, not `return`ing returns an unspecified value (I think?); again, not an "error".

3.2 Warnings

Language *implementations* often try to help programmers by giving "warnings" for code that is likely to be wrong, but doesn't do anything the language actually forbids. For example, `gcc` has many kinds of warnings, some of which can be extremely useful. To me, many of these warnings are only to compensate for C's design flaws, but some warnings are useful even in languages I like better. OCaml can warn you when you use OCaml's `let` (like Racket's `let`) to bind an identifier that you never use, which catches quite a few bugs.

3.3 When are errors caught?

With some idea of *what* an error is—something that is caught, or reported—we can ask: *when* are errors reported?

As usual, it depends on the language, but we can make some generalizations that are (almost) always true:

- **Syntax errors** are caught during parsing.
- **Scope errors** are often caught during parsing, but not always; Racket doesn't catch them until you either run your program or click "Check Syntax" (though DrRacket's "Check Syntax" checks for a few kinds of errors that aren't usually considered syntax errors).

Beyond syntax and scope errors, it depends entirely on the language.

- **Agreement errors and mismatches**: Caught at run time in Racket and Python; caught at compile time in C, SML, OCaml, Haskell. (Java catches many of these at compile time, but not all.)
- **Array bounds error**: Caught at run time in Racket, Python, Pascal, Java, SML, OCaml, Haskell; (mostly) caught at compile time in some "cutting-edge" (last 20 years) research languages.

  C's behaviour is almost entirely undefined; a program reading or writing outside an array may crash ("segmentation fault", "bus error", etc.) or continue unpredictably (or *too* predictably, as with countless viruses that exploit "buffer overruns").

- **Not returning anything**: Caught at run time (sometimes?) in Racket; caught at compile time in Java, SML, OCaml, Haskell. (Not an error in Python, C, C++.)

3.4 Types: raising errors earlier than run time

The most popular purpose of a *type system* is to prevent "agreement errors and mismatches", such as applying a list to a function (rather than applying a function to a list) or using + to add things that can't be added. We can specify a type system with rules (in fact, this is much closer to Gentzen's motivation—formal proofs—than using his notation to specify dynamic semantics), which guide the language implementor at compile time, rather than run time.

It doesn't make sense to talk about "compile time" unless there's a compiler. So, to cover both compilers and interpreters, we'll say that types catch errors *statically*, and that a type system is part of the *static semantics* of a language.

The part of a language implementation that checks the abstract syntax of a program, to see whether it violates the type system, is called a *type checker*.

Often, the type checker is checking expressions (or statements, etc.) against type declarations written by programmers. But some languages don't require programmers to write types (or to write only a few types). In these languages (e.g. Haskell), the type checker is sometimes called a *type inferencer*, because it *infers* types the programmer didn't write. The line between "checking" and "inference" is fuzzy, so I use "type checker" for all typed languages, even languages that infer types.

In an interpreter for a typed language, types are checked after parsing (so the type checker can work with abstract syntax instead of concrete syntax), but *before* running the program. In a compiler for a typed language, types are checked before the compiler generates machine code.

3.5 What about C?

The C language (and C++) don't fit neatly into the "typed"/"untyped" space. If you've written many C programs, you know that C compilers like to complain about type mismatches—so C must be typed, right?

Without getting mired in terminological disputes, that question has two reasonable answers:

- C is not typed, because a program that passes the type checker may still do (clearly) bad things, such as segfault; and this is unlike Java, SML, and Haskell.
- C is typed (C compilers complain about type errors!), but not *safe*, because (as an example) C never checks for array bounds errors. (Sometimes, C is called "weakly typed".)

If we like the second answer better, we can classify languages along two dimensions: typed vs. untyped, and safe vs. unsafe. Then we would say that

- C is typed but unsafe;
- Java is typed and safe;
- SML, OCaml, and Haskell are typed and safe;
- Racket, JavaScript, and Python are untyped but safe;
- assembly language is untyped and unsafe.

These distinctions are explained pretty well by Luca Cardelli, in the first few pages of a paper (`http://www.lucacardelli.name/Papers/TypeSystems.pdf`). I encourage you to read it (the first few pages, not necessarily the whole paper!), though I can't promise that my terminology will always be the same as his.

Most "scripting languages" are untyped and safe.

Question: What if we implement a safe language using an unsafe language? For example, `bash`, which is safe, is written in C, which is unsafe. What if the `bash` interpreter segfaults?

I wouldn't say this makes `bash` unsafe. Rather, the *implementation* of `bash` has a bug. I haven't read the definition of `bash`, but unless it says that some behaviour is unspecified, any interpreter that segfaults is not consistent with the definition.

This goes for compilers as well: a Racket compiler might have a bug that causes it to generate machine code that segfaults, even though Racket is safe, so programs should never segfault.

(A more subtle case is when a language's implementation *is* consistent with the language's definition, but the language's definition is wrong. For example, a compiler that follows an "obvious" definition of type polymorphism may generate unsafe code. I mentioned *determinism* in previous lectures as an example of a good property of a language's definition; an arguably more important property is *type safety*, which we'll cover later.)

4 Good things aren't happening, and I don't like that either

I said that safety—stopping errors from happening, or rather, reporting errors statically (before you run the program) rather than dynamically (while the program is being run, either by you, or by an unhappy user), is the most popular reason to use a typed language.

Another reason, which I think is more interesting, is to ensure that things you *want* to have happen actually will happen.

When you write a helper function, you (should) write a comment with a "signature" that describes what kind of animals the function expects as input, and what kind of animal it produces as output.

;; match-length : string string → natural
;; interleave : (listof any?) (listof any?) → (listof any?)
;; contains-sequence : (list-of symbol?) (list-of symbol?) → boolean?
;; truth-or-lie? : Bool-expr → boolean?

Unfortunately, in Racket, these signatures are merely comments. Racket is not typed, so it doesn't check whether any of these signatures match the function you actually wrote.

You can see some inconsistency in our style, in fact: is it "`listof`" or "`list-of`"? Or maybe we should put a question mark after `Bool-expr`. After all, `Bool-expr?` is an actual Racket/PLAI function, as is `boolean?`. But `natural` isn't (even with a question mark). These signatures are not part of the Racket language, so they are useful documentation, but Racket doesn't check whether that documentation is accurate.

In contrast, in a typed language, the signatures you give actually matter to the language. If we write `interleave` in Haskell:

```
interleave :: [Int] -> [Int] -> [Int]
interleave [] list2 = list2
```

```
interleave list1 [] = list1
interleave (first1:rest1) (first2:rest2) = first1:first2:(interleave rest1 rest2)
               -- ^ Haskell ":" is like Racket "cons"
main =
   do
      putStrLn (show (interleave [1, 3, 5, 7] [2, 4, 6, 8, 9]))
```

the *type annotation* (or *type declaration*, or *type signature*) on the first line guarantees that the Haskell type checker will make sure—assuming `interleave` is passed two lists of integers—that every clause of the definition will evaluate to a list of integers. It will also check that, when we call `interleave` on the last line, that we are passing lists of integers. All of this happens after parsing, *not* when we run the program.

When the Haskell program *is* run, it doesn't have to check whether the second argument of `cons` (spelled "`:`" in Haskell) is a list: it *will* be a list, because the type checker accepted the program. Here, we are still in the category of "stop bad things from happening".

Quite a few mistakes will be caught by Haskell's typechecker. For example, if we wrote

```
interleave :: [Int] -> [Int] -> [Int]
interleave [] list2 = list2
interleave list1 [] = list1
interleave (first1:rest1) (first2:rest2) = first1:first2:(interleave rest2)
```

Haskell will complain, because we applied `interleave` to one argument rather than two. (Actually, it will complain because `(interleave rest2)` returns a function of one argument, and a function of one argument is not a list.)

But many other mistakes will not be caught. We might forget to cons `first2`:

```
interleave :: [Int] -> [Int] -> [Int]
interleave [] list2 = list2
interleave list1 [] = list1
interleave (first1:rest1) (first2:rest2) = first1:(interleave rest1 rest2)
```

This will not be caught: the type annotation demands that `interleave` return something of type `[Int]`, a list of integers, and `first1:(interleave rest1 rest2)` has that type.

All mainstream typed languages (including Haskell and SML) are limited in what their type systems can check. The specific limits vary from language to language. Haskell (or rather its most popular compiler, GHC) has developed a rather powerful, but complicated, type system; SML has a simpler type system than Haskell.

What I said above—that Haskell will check "that every clause of the definition will evaluate to a list of integers"—is not entirely accurate. We will make this kind of statement accurate, and more precise, over the next few weeks. This should also illuminate the line between "preventing bad things" and "ensuring good things".

4.1 Refined type systems

While popular typed languages are limited in what their type systems can check, many experimental typed languages push these limits—sometimes amazingly far. What if we could write, in the type

annotation for `interleave`, that the length of the list it returns is equal to the sum of the lengths of its arguments? This is within the power of modern Haskell, and of several recent experimental languages.

What if we could write that `interleave` should return a list whose elements are a *permutation* of the elements in its arguments? This is (I believe) beyond Haskell, at least for now.

5 Object-oriented languages

Like functional languages, some object-oriented languages are typed, and some are not. The one you're probably most familiar with, Java, is typed. Java's type checker catches many mistakes, but it catches fewer mistakes than Haskell or SML programmers might like. We will explore this in future lectures, but a short, vague explanation is that object-oriented languages assume an "open world": given a particular class, say `DoorLock`, we can declare subclasses of `DoorLock` representing new kinds of locks. We don't know in advance how many subclasses of `DoorLock` will be created. Back in the 1990s, Java was motivated by the desire to send Java programs over the Internet, with the expectation that a program written by the original `DoorLock` author might interact with subclasses of `DoorLock` written by other people around the world.

In contrast, the **define-type** of PLAI, the `datatype` declaration of SML, and the `data` declaration of Haskell define a "closed world": a PLAI program cannot add variants to a **define-type**. This is what allows PLAI to statically check whether you have missed a branch in a **type-case**. In Java, we cannot enumerate all possible subclasses, because our compiled Java program might load new ones!

6 Typed programs run faster

Another advantage of typing, which slipped my mind until just now, is that an implementation of a typed language can safely omit some of the checks that would otherwise be required. (Cardelli calls this *economy of execution*.) For example, when you evaluate (Add *e*1 *e*2), you have to check that the values that *e*1 and *e*2 evaluate to are Nums. This remains true even if you use `Num-n` to access the `n` field of the `Num` variant: Racket/PLAI will do this check "under the hood".

In an interpreter, the cost of such checks is almost certainly far outweighed by the difference in cost between interpreted code and compiled code. So this point in types' favour is usually raised in the setting of compiled code. This was a powerful argument for typed languages until the 1990s; modern hardware architectures have drastically reduced the cost of such checks.

7 Disadvantages of typed languages

Catching errors statically seems better than catching them dynamically. If your program has an error, you probably want to find out sooner rather than later. But what if the "error" wouldn't have actually happened? Consider the program in Section 8.2. Java rejects this program because the `else` branch doesn't have a `return`. However, we would expect that this won't matter, because the test in the `if` statement will always be true and the `then` branch, which *does* have a `return`, will be executed.

Whether this expectation is true is another question. Mathematical properties of the reals rarely hold for floating-point numbers. What if `x` is $+\infty$? If you use floating-point arithmetic for anything

important, you probably need to become horribly familiar with the IEEE 754 standard. Wikipedia has the following hint of the horrors lurking within:

> `https://en.wikipedia.org/wiki/IEEE_floating_point`
>
> - Two infinities: $+\infty$ and $-\infty$.
> - Two kinds of NaN: a quiet NaN (qNaN) and a signaling NaN (sNaN). A NaN may carry a *payload* that is intended for diagnostic information indicating the source of the NaN. The sign of a NaN has no meaning, but it may be predictable in some circumstances.

I believe that equality is one of IEEE 754's unpredictable operations, so I wouldn't expect changing the condition to `x == x` to necessarily solve this issue.

But we could replace the floating-point arithmetic with something more reliable, like integer arithmetic; the test `x == x` *will* always succeed if `x` is an integer. Then, Java is complaining about an "error" that is *guaranteed* not to happen.

When such issues are raised with advocates of typing (and in this situation they will often be called advocates of *static typing*, because their opponents claim to support "types", as long as the "types" are only checked dynamically), they might respond like this:

- (Appeal to dogma) You shouldn't write an else branch without a `return` anyway. Types are like logic! Types are part of the fabric of the universe! And *of course* a function should always return something. (Unless it runs forever. Or raises an exception. Both of which are not terribly logical...)[2]
- (Appeal to courtesy) You shouldn't do that, because someone else (such as yourself, a year later) should be able to read your function and understand immediately that it will return something. If they need to reason about the `if` condition to understand why the function will return something, that's not immediate.
- (Appeal to simplicity) Maybe the language should let you do that, but it would make the type system harder to understand, and the type checker harder to implement.

[2] To be fair, many of the people who are most dogmatic about this are also interested in advanced type systems where you can prove that your program won't run forever.

8 noreturn examples

8.1 noreturn.c

```
#include <stdio.h>

int f (int x)
{
  x * 2;
}

int main (int argc, char **argv)
{
  printf ("noreturn returned: %d\n", f(5));
}
```

With `gcc`: no warnings.
With `gcc -Wall`:

```
noreturn.c: In function 'f':
noreturn.c:5: warning: statement with no effect
noreturn.c:6: warning: control reaches end of non-void function
```

8.2 noreturn.java

```
public class noreturn {

  public static int f (double x) {
    if (x == (x + 0.1 - 0.1)) {
      return 0;
    } else {
      // return 1;     // even when there is a return in the then-branch,
                       // Java rejects this program because
                       // there's no return in the else-branch.
    }
  }

  public static void main(String[] args) {
    System.out.println(f (5.0));
  }

}
```