Languages and Compilers
An Introduction to Types

Adam Sampson
School of Design and Informatics
Abertay University

Overview

- What's a type?
- Types of type system
- What kinds of types do we usually have – and what are some of the concerns when designing a type system?

The idea of types

- As programmers, we're used to thinking about the **types** of things:
  - “the variable `count` contains an `int`”
  - “the expression `true` has type `bool`”
  - “the expression `true && false` has type `bool`”
  - “the expression `count == 0` has type `bool`”
- A **type** is a collection of possible values
  - `bool` has the values `true` and `false`
  - `unsigned short` has the values:
    0, 1, 2, 3 … 65535

Type systems

- When we're defining a programming language, one of the things we need to define is the **type system**
- This is part of the semantics of the language
  - What types exist, how they work, how you define new ones
- … and also part of the syntax of the language
  - Most languages need some way of naming types, through a **type expression**, e.g.
    `int[]  Maybe String   int * string`
    `const std::vector<bool>&`

Typechecking

- During semantic analysis, the compiler will check that the types in the program are **consistent**
  - e.g. `int x = 3;` is consistent; `int x = “Hello!”;` isn't
- This is the job of the **type checker** – which follows **typing rules** in order to decide whether the program is typed correctly
- Typing rules define the type system (what valid types you can construct) in sort-of the same way that BNF defines the syntax (what valid sentences you can construct)...

Types of type system

- Languages fall roughly into three categories – we discussed this a bit in Lecture 1...

- Typeless languages
- Dynamically-typed languages
- Statically-typed languages

- `#include <lunch>`

Typeless languages

- A **typeless language** has no idea of the types of values or variables
  - e.g. assembler/machine code, BCPL, B, MCPL
- The programmer's mental model is basically the same as the underlying hardware:
  - Memory consists of fixed-size cells (e.g. 32-bit)
  - Whenever you do an operation, you must specify what type to treat them as...
  - … so you have an “integer add” operator, and a “floating-point add” operator, and and and...
- Surprisingly, it is actually possible to program this way... but type errors just mean runtime crashes

Dynamically-typed languages

- A **dynamically-typed language** assigns types to values, but not to variables
  - e.g. Smalltalk, Python, Ruby, PHP, APL, Erlang...
  - In Python, `n = 3; n = “Hello”` is fine
  - … so the type of value stored in a variable can change while the program's running...
  - … and the language must decide what operation to perform based on the current type (**dynamic dispatch**)
- This limits the optimisations that the compiler can perform, and makes it much harder for the compiler to detect type errors

Statically-typed languages

- A **statically-typed language** fixes the types of all values and variables at compile-time
  - e.g. C, C++, Java, C#, ML, Haskell, Go, Rust...
- … which means either the programmer has to specify them (C, Java) or the compiler has to do a lot of extra work to **infer** them (ML, etc.)...
- … but the semantic analysis becomes much more powerful: you can detect most/all type errors at compile time, and generate better code

Why have both?

- Dynamically-typed languages are typically seen as “more expressive”
  - Shorter code for the same problem because you don't need to specify types...
  - … which means you're trading off correctness (and performance) for succinctness
- Ideally we'd have a static type system, but with sufficiently clever inference that the compiler can always figure out what type you mean
  - … and type inference is steadily getting better – type systems are a cutting-edge CS research topic

Primitive types

- Your type system usually starts with the **primitive types** built into the language
- The simplest kinds of single values
- e.g. `int`, `float`, `bool`
- The possible values of these are obvious – and usually correspond to the machine in some way

Type conversions

- If I use an `int` in a context where a `float` is expected, what happens?
- Does it get automatically converted?
  - (an **implicit type conversion**)
  - You need to be very careful with this to avoid losing precision – some languages only allow safe implicit conversions, some don't allow it at all
- Can I *ask* for it to be converted?
  - (an **explicit type conversion**)

Unit/void types

- Most languages have an even simpler primitive type – a placeholder for nothing
  - e.g. when you want a function to return nothing
- e.g. `void`, `unit`
- There is **only one** possible value for this – if the language lets you talk about the value at all!
  - e.g. `()`
  - Not the same thing as `nullptr` or `None`, which are sentinel values of other types

User-defined types

- A type system really becomes useful when we can define our own types for particular problems
- The simplest approach tends to look like this:
  - `typedef float length;`
  - `using length = float;`
  - `type length = float`
- Now I can declare a variable containing a length, rather than a float...

Aliasing vs. new types

- When you define a new type like that:
```
using length = float;
using weight = float;
```
- … does the language treat it as a **new type** or just **another name for the existing type**?
- i.e. if I have a function expecting a `length`, can I pass it a value of type `weight`?
  - This is probably an error... but different languages have different semantics here, and some provide both options (e.g. Haskell `type` vs. `newtype`)
  - Sometimes (especially in C++) you really do just want an easier-to-spell name for an existing type!

Product types

- A **product type** combines two or more values (of existing types) together
  - e.g. a **tuple type** `(Int, String)` or `Int * String`
  - or a structure with named fields:

```
struct {
  int age;
  string name;
}
```

- It's a product type because the possible values are the product of the possible values of the types being combined: (0, “Adam”), (1, “Adam”)...
  (0, “Fred”), (1, “Fred”)...

Classes

- A class is a kind of product type – a collection of related data items
- From a type perspective, what makes classes special is how **inheritance** works:
  if I have a function expecting an `Animal`, I can pass it a `Dog`
- However, it's possible to get the same effect in different ways (e.g. interfaces/traits) – so don't assume that you have inheritance in all languages

Literals vs. type expressions

- There's nothing that says a language's type expressions have to look like its literals
  - Although modern language designs usually try to do this as far as possible...

- ML: int * int * int (1, 2, 3)
- Haskell: (Int, Int, Int) (1, 2, 3)
- C++: struct { int a, b, c; } { 1, 2, 3 }

Sum types

- A **sum type** also combines existing types, but holds only one of them at a time
  - ```
    type ContactDetail =
        Address of string
        | Phone of int ;
    ```
  - A `ContactDetail` can be an `Address` with a `string` inside it, or a `Phone` with an `int` inside it
  - (please do not really store phone numbers as `int`s)
- It's a sum type because the possible values (in this case) are all the possible `string`s, **plus** all the possible `int`s
    ```

Sum types

- You will also hear these called **variant** types or **union** types
- Some languages (e.g. C/C++) leave it up to the programmer to remember which type they've stored: **untagged union** types
  - … which is inherently unsafe
- Enumerated types are a restricted subset of this: data Day = Monday | Tuesday | Wednesday …
- Knowing the possible values means the compiler can force you to deal with all of them!
  - e.g. in a `switch` or pattern match

Option types

- An **option type** represents a value that may or may not be present
- In languages with (tagged) sum types, this can be defined as a sum type
  - Haskell: `data Maybe t = Just t | Nothing`
  - … so `openFile` can return `Maybe File`
- Again: not the same as `nullptr`/`null` which are just sentinel values in other types
  - Use an option type, and the language can force you to deal with the special cases correctly

Function types

- We can write a type expression describing a function as a mapping from its inputs to its outputs
  - `sqrt :: Float → Float`
  - `(+) :: Int → Int`
  - `int (*atoi)(const char *);`

Higher-order functions

- This becomes more interesting in languages where we can treat functions as values, e.g. write a function that takes another function as an argument
  - `map :: (Int → Int) → [Int] → [Int]`
  - (takes a function from integers to integers,
    and a list of integers, and applies it to each integer)

Array/sequence types

- Many languages provide some special representation for arrays, associative arrays, or other collection types – a kind of product type
- Is the size of the array part of the type or not?
  - C++: `int[10], int[]`
  - Java: `int[]`
  - ML: `int array`
- Knowing the size at compile time permits more accurate semantic analysis – e.g. efficient array bounds checks

Any questions?

- Constructing a type system – like constructing a grammar
- I have not talked about:
  - generic types
  - type inference
  - Next week!
- Solutions for Practical 2 are on Blackboard now

- **Next lecture**: lexical analysis