INTRODUCTION TO TYPE SYSTEMS
A major concern of Software Engineers should be the correctness of the software that they produce.

However, correctness can mean many different things:

- Software Testing
- Model Checking
- Static Analysis
- Algebraic Specification
- Formal Methods
- Runtime Monitoring
- Theorem Provers
- Hoare Logics
- Denotational Semantics
- Formal Methods
- Runtime Monitoring

For a lot of Software Engineers the issue of correctness stops here. For a lot of Software Engineers these techniques are considered too heavyweight in practice.

We all use these already! Lightweight formal methods.
WHAT IS A TYPE SYSTEM?

“A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying (program) phrases according to the kinds of values they compute”

Benjamin Pierce

So what are types?

They are the particular classifications of program behaviours that one makes. For example, in simple type systems for C functions we classify programs according to the number of and primitive type of the arguments along with the type of the result of the function.

Types are **abstract** descriptions of programs

We can study the correctness properties of interest by abstracting away all of the low-level details

Types are **precise** descriptions of program behaviours

We can use mathematical tools to formalise and check these interesting properties
WHAT DO TYPES DO FOR US?

- We mentioned above that Types Systems are concerned with correctness and that we use type systems to guarantee the absence of certain behaviours: e.g.
  - application of an arithmetic expression to the wrong kind of data
  - existence of a method or field when invoked on a particular object
  - array lookups of the correct dimension

- Type Systems can also be used to enforce higher-level modularity properties
  - maintain integrity of data abstractions
  - check for violation of information hiding

- In doing this, Type Systems enforce disciplined programming
  - type systems form the backbone of module based languages for large-scale composition
  - types are the interfaces between the modules
  - encourages abstract design
  - types also are a form of documentation
Most programming languages use at least some notion of types for programs or the data that programs manipulate. But there are several different approaches to using types.

- **Strongly Typed**
  - Languages prevent programs from accessing private data, from corrupting memory, from crashing the machine etc

- **Weakly Typed**
  - Languages do not always prevent errors but use types for other compilation purposes such as memory layout.

- **Static Typing**
  - Refers to the point at which type checking occurs - compile time.

- **Dynamic Typing**
  - Refers to type checking that is delayed until run time.

Strong / Weak typing are opposite approaches.
Static / Dynamic typing are opposite approaches.
The Strong/Weak and Static/Dynamic approaches are independent.
E.g., we can say a language has Strong Static Typing.
Liskov and Zilles described “Strong” typing by the requirement that “whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function.”

This ‘definition’ generalises to the same requirement on any consumer of data.

Let’s consider a language to have “Weak” typing otherwise.

One implication of Strong Typing is that intended types must be declared or inferred with functions/methods.

This can make languages verbose - Java suffers from this

An implication of Weak Typing is that data of the ‘wrong’ type may be passed to a function - and the function is free to choose how to behave in that case.

Typically, weakly typed languages attempt to implicitly coerce data to be of the required type. This can go disastrously wrong : Software Engineers favourite example is Ariane 5.

“The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. “ (from the Ariane 501 Inquiry Board Report)
STATIC VS DYNAMIC TYPING

• Statically typed languages necessarily use an *approximation* of the run time types of values.
  • Why? Because statically determining control flow is undecidable (in sufficiently rich languages).
  • Compile time checking can avoid costly run time errors though.
  • Where types are used for memory layout, static typing is appropriate (e.g. C)

• Dynamically typed languages check the types of data at point of use in run time.
  • Exact types can be used. This implies no false negative type errors.
  • It is quite common to allow variables to change the type of data they store, or objects to dynamically grow new methods. Some programmers find this convenient.
  • Very common in scripting languages and web programming.
  • Should not be used for Critical Systems as errors may be detected too late.
## Languages and Their Types

<table>
<thead>
<tr>
<th></th>
<th>Weak</th>
<th>Strong</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dynamic</strong></td>
<td>PHP, Perl, Javascript, Lua</td>
<td>Python, Lisp, Scheme</td>
</tr>
<tr>
<td><strong>Static</strong></td>
<td>C, C++</td>
<td>Ada, OCaml, Haskell, Java, C#</td>
</tr>
</tbody>
</table>

Visual Basic (.NET) ?!??!!?

Future programming languages are likely to feature controllable Dynamic / Statically typed regions of code - these are likely to be able to interact in safe ways.
WHEN DO WE CHECK TYPES?

• Type checking is performed as part of compilation for strongly typed languages
  • in the compiler front end - type checking is done on abstract syntax tree of the program and is sometimes referred to as “semantic” analysis
  • Most strongly typed languages insist that all program variables are manifestly typed, some (e.g. OCaml, Haskell) allow type inference so that the compiler automatically works out suitable types.
• So this means that there is a part of the implementation of the compiler that guarantees a certain amount of correctness with respect to programs that pass its type checker:
  • This is some sort of algorithm coded up in a high-level language, or even in C.

• How do we know we can trust this type checking algorithm? Do we really know that it gives us what is called **type safety**?

Type safety, for strong statically typed languages, can be loosely described with the quote:
“‘Well-typed programs never go wrong’”  (Robin Milner)
WHAT DOES IT TAKE TO PROVE TYPE SAFETY?

• You need to know what “well-typed” means - precisely
• You need to know what “programs” means - precisely
• You need to know what “never go” means - precisely
• You need to know what “wrong” means - precisely

“Well-typed programs never go wrong”

This is the where the mathematics comes in!

• We give an inductively defined **typing relation** - a bit like a proof in propositional logic
• We give an inductively defined **reduction relation** - i.e. how the program runs
• We give a description of the **error** states of the program - these vary widely
• We use the mathematical descriptions to try prove that type safety holds for a given language

This is of course all very language dependent - the larger the language the more work there is
WHAT WILL WE DO IN THE NEXT FEW WEEKS

• We'll focus on small example languages to give you an introduction to the techniques.
• We'll learn how to define operational semantics for a language
• We'll learn how to design and define a type system for a language
• We'll learn how to relate the two mathematically

How does the mathematics correspond to the code in the compiler that does the type checking?

• We'll learn how to write an interpreter in Haskell that implements our operational semantics
• We'll learn how to write a type checker in Haskell that implements our type system
• We'll learn how to do this for a variety of language features
NEXT STEPS

Step 0 : Functional Programming
- Haskell functional language
- Types of functional programs
- Implementation of programming languages using Haskell
  
  We have already started this in Semester 1

Step 1 : Type Systems
- Simple Types
- Structured Types
- Reference Types
- Subtyping
- Object Types
- Polymorphism

Step 2 : Modelling Programming Languages
- Syntax and Operational Semantics
- Denotational Semantics
- (Simply Typed) Lambda-calculus
- Type Safety

Step 3 : Other Observations
(time permitting)
- The Curry Howard Isomorphism
- Contextual Equivalence
NEXT LECTURE: TYPE DERIVATION RULES