COMPSCI 3MI3 - Principles of Programming Languages

Topic 1 - Introduction

NCC Moore

McMaster University

Fall 2021

Adapted from "Types and Programming Languages" by Benjamin C. Pierce

So What's the Point of this Course?

We are going to study how programming languages can be *constructed* using the formal mathematical tools you learned last year in COMPSCI 2LC3.

- We will use **formal methods** to define and reason about programming languages.
- Using these tools, we will construct a number of small programming languages, and reason about their correctness.
- **Type Theory** will be a major topic in this course, as will **Lambda Calculus**.

Formal Methods

In computer science, we use **formal methods** to reason about the correctness of software (and hardware). Advanced mathematics are employed to attempt to demonstrate the consistency of a program with implicit or explicit specifications or requirements (see COMPSCI 3RA3).

- “full strength” formal methods include:
  - Hoare Logic
  - Algebraic Specification Languages
  - Modal Logic
  - Denotational Semantics
- “lightweight” formal methods include:
  - Model Checkers
  - Satisfiability Modulo Theory solvers
  - Run-Time Monitoring
  - **Type Systems**

Type Systems

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

- **tractable** → In Computer Science, a problem is *intractable* if it can be solved in theory, but not in practice.
    - For example, the **state explosion problem** is a source of intractability.
- **syntactic** → That is, we analysing a program's *grammatical structure*.

Using type systems, we can demonstrate the *absence* of certain bad behaviours, but not their *presence*, so type systems are necessarily conservative.

Static vs Dynamic Checking

There are two main types of type checkers:

- **Static**
  - Analysis is based on information extracted from the program's **abstract syntax** or **parse tree**.
  - Examples include Ada, C, C++, C#, JADE, Java, Fortran, **Haskell**, ML, Pascal, and Scala.
- **Dynamic**
  - Analysis is based on information revealed *at runtime*.
  - Examples include Groovy, JavaScript, Lisp, Lua, Objective-C, PHP, Prolog, **Python**, Ruby, Smalltalk and Tcl

What Sorts of Things Can be Type Checked?

We can check that:

- Arguments to functions or operators are of compatible types.
- A class actually contains a requested member function.
- A function recieves the correct number of arguments.
- Access violation of private members.

We can't check that:

- Specific values are being given to operators, such as dividing by zero.
- The function x returns value y when given input z.

The set of errors which a type system checks for is a *design consideration of the language*!

You're Just Not My Type!

Type systems can figure out a lot on their own, but we (the programmers) can help them out with **type annotations**.

- Most languages minimize the number of annotations required by **inferring** types when they aren't specified (see Python and Haskell).
- Other languages require an explicit declaration for everything (see C), though some degree of type inference is standard in most languages.

In some languages, such as **Agda**, the type system is used to create "proof carrying code". In Agda, you can't get something to compile without having also created a proof of your program's correctness within the type system.

So What's This Good for Again?

"It turns out that a fair amount of careful analysis is required to avoid false and embarassing claims of type soundness for programming languages. As a consequence, the classification, description, and study of type systems has emerged as a formal discipline." – Luca Cardelli, 1996

Detecting Errors

The most obvious benefit of type checking is early error detection.

- In general, errors are best caught at compile-time, rather than at run-time.
- This is what gives richly typed languages (like Haskell) their “it just works” quality.
- You get the most bang for your buck when you start encoding your data structures in your type system.
    - This has an added maintainability benefit. If a data structure has to change, your type system can reliably indicate every place in your program that needs subsequent modification.

Abstraction and Documentation

Rigorous type systems enforce disciplined programming.

- Type checking allows us to guarantee that modularization techniques such as class structures, modules, packages, and functions aren't being misused.
- Enforcing type signatures leads to more abstract software design, which is good for everyone involved!

Typing also makes programs easier to read.

- Type information gives useful hints about behaviour.
- Unlike descriptions found in comments, which can get out-of-date, otherwise it would fail to type-check.

Language Safety

"Informally [...], safe languages can be defined as ones that make it impossible to shoot yourself in the foot while programming." (Pierce, 2002)

- Safety refers to whether or not a language protects its own abstractions.
  - In Haskell, lists can only be accessed in the normal way.
  - In C, pointer manipulation can be used to violate the bounds of arrays to read adjacent data.
- Language safety can be enforced either statically or dynamically, though often a combined approach is used.
  - Haskell, for example, checks array bounds dynamically.

	Statically Checked	Dynamically Checked
Safe	ML, Haskell, Java	Lisp, Scheme, Perl, Postscript
Unsafe	C, C++	

Further Applications

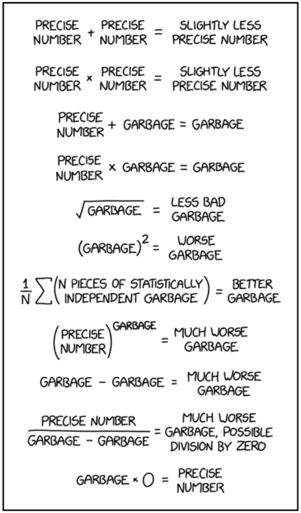
- Frontloading correctness checking as static typechecking reduces the need for runtime checks, thereby improving efficiency.
- In network security, preventing undesirable and unexpected behaviours is important!
- In automated theorem proving, type systems are used to represent logical propositions and proofs.

Wait, I Thought this Class was About Programming Languages...

In this course, we are going to start off with the bare-bones basics of programming language construction and design. The tools we learn will allow us to reason about programming languages, and will structure our later discussions of type theory.

- "In modern languages, the type system is often taken as the foundation of the language's design, and the organizing principle, in light of which every other aspect of the design is considered." (Pierce, 2002)

Last Slide Comic


PRECISE NUMBER + PRECISE NUMBER = SLIGHTLY LESS PRECISE NUMBER
PRECISE NUMBER × PRECISE NUMBER = SLIGHTLY LESS PRECISE NUMBER
PRECISE NUMBER + GARBAGE = GARBAGE
PRECISE NUMBER × GARBAGE = GARBAGE
√GARBAGE = LESS BAD GARBAGE
(GARBAGE)² = WORSE GARBAGE
1/N ∑(N PIECES OF STATISTICALLY INDEPENDENT GARBAGE) = BETTER GARBAGE
(PRECISE NUMBER)^GARBAGE = MUCH WORSE GARBAGE
GARBAGE − GARBAGE = MUCH WORSE GARBAGE
PRECISE NUMBER / (GARBAGE − GARBAGE) = MUCH WORSE GARBAGE, POSSIBLE DIVISION BY ZERO
GARBAGE × 0 = PRECISE NUMBER