Types

Principles of Programming Languages

Mark Armstrong

Fall 2020

1 Preamble

1.1 TODO Notable references

:TODO:

1.2 TODO Table of contents

2 Introduction

This section introduces the concepts of *types*, a particularly useful language safety feature.

Common simple types and methods of building new types are discussed, as well as some more advanced topics.

3 Properties of type systems

In the previous notes, we have discussed

- polymorphism and
- static/dynamic typing

which are two important properties of a type system.

Here we discuss some other commonly discussed properties, before discussing in the following sections what is arguably the most important property: what types a language might have.

3.1 "Strong" and "weak" typing

These are comparative terms.

- We'll consider them a subjective criteria.

"Strongly typed"

- Languages are frequently called strongly typed.
  - But less frequently do they state what they mean by that.
  - The term is used inconsistently.
    - "C is a strongly typed, weakly checked language" – Dennis Ritchie, creator of C
- We will take it to mean "type clashes are restricted".
- That definition does not make a good objective property.
  - What does restricted mean?
    - Is it a warning or an error?
    - Does type casting violate this?
  - What qualifies as a type clash?
    - Is implicit type casting allowed?

"Weakly typed" simply means not strongly typed.

3.2 Explicit and implicit typing

Languages may require annotations on variables and functions (*explicit typing*) or allow them to be omitted (*implicit typing*).

- Implicit typing does not weaken the typing system in any way!
  - A very common misconception.
- In general, type inference is an undecidable problem (its not guaranteed that the compiler/interpreter can determine the type).
  - Most languages have relatively simple type systems, and this is not a problem.

Some languages make type annotations a part of the name, or annotate names with sigils to indicate type details.

- In older versions of Fortran, names beginning with `i`, `j` or `k` were for integer variables, and all variables were of floating point.
- In Perl, names beginning with the sigil
  - `$` have scalar type,
  - `@` have array type,
  - `%` have hash type, and
  - `&` have subroutine type.

4 Atomic types

We begin our discussion of what types languages have with what are usually the "simplest" types: *atomic* types.

- Atomic in the sense that they cannot be broken down any further.
- Sometimes called *primitive* or *basic*.

Most languages have at least these atomic types.

- **Integers**; `int`
  - Including possibly signed, unsigned, short, and/or long variants.
- **Floating point** numbers
  - Including possibly single precision and double precision variants.
- **Characters**
  - Sometimes an alternate name for the byte type (8-bit integers).
- **Booleans**
- **Unit** (the *singleton* type)
  - Sometimes called `void`, `nil`-type, `null`-type or `none`-type.
    - In C like languages, you cannot store something of type `void`.
    - Commonly represented as the type of 0-ary tuples, whose only element is `()`.
- **Empty**
  - Unlike a singleton type, which has a single value (called `nil`, `null` or `none`), there is **nothing** in the empty type.

4.1 Implementation of atomic types

When we discussed the pure untyped λ-calculus, we discussed the process of *encoding* the integers and booleans as functions, since they were not included in the language.

- We also mentioned that we can add constants for them to the language, forming an *unpure* untyped λ-calculus.
  - Such added constants are called

This raises a question we can ask about "practical" programming languages as well;

- are the "atomic" ("primitive", "basic") types *truly* atomic (primitive, basic), or are they represented using one of the language's abstractions?
- We have discussed the fact that in Scala and Ruby, which we call "purely object-oriented", even these atomic types are classes!
  - Whereas in Java and C++, they are not; there, they are "primitives" which exist outside the object-oriented abstraction.

4.2 Uncommon basic types

Some languages include these less common basic types.

- **Complex** numbers
  - Especially for scientific computation.
- **Decimal** (representation of) numbers
  - Especially for business (monetary) applications.
  - There are decimal numbers that cannot be properly represented using binary (e.g. `0.3 = 0.010011`, repeating)
  - Not included in all languages because they cannot be efficiently represented.
    - It takes at least 4 bits to represent a single decimal digit, but 4 bits could represent 16 digits, instead of the 10 that are actually possible.

4.3 Ordinal types

Many languages include a means of defining other *finite* types. Instances include

- enumeration types (`enum`'s) and
- subset/subrange types.

5 Sequences

:TODO:

6 Algebraic types

:TODO:

7 References

:TODO:

8 Advanced type systems

:TODO:

9 Further advanced topics

Depending upon time at the end of the course, we may return to discuss more about types.