Types

Principles of Programming Languages

Mark Armstrong

Fall 2020

1 Preamble

1.1 Notable references

- Robert W. Sebesta, "Concepts of Programming Languages" (10th edition)
  - Chapter 6, Data Types
- Benjamin Pierce, "Types and Programming Languages"
  - Chapter 11, Simple Extensions
    - Unit, Tuples, Sums, Variants, Lists.
  - Chapter 13, References
    - Reference types.
  - Chapter 20, Recursive types

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.

The discussion in this section is fairly "encyclopedic"; the next section on the typed λ-calculus give some insight about how some of these types can be incorporated into a programming system.

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

"Weakly typed" simply means not strongly typed.

3.1.1 So what does "strongly typed" mean?

We will take it to mean "type clashes are restricted".

- A *type clash* being an instance where one type is expected but a different, incompatible type is provided.
  - Such as added a string to an integer.
- 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?

3.1.2 Exercise: Examples of implicit type casting

What are the results of these two Javascript output statements?

```
console.log(1 + 2 + "3")
console.log("1" + 2 + 3)
```

How about the following C++ code? (This is arguably not quite an example of implicit type casting.)

```
#include<iostream>

int main() {
  std::cout << ("Hello" + 1);
}
```

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.
  - We will likely later study the *Hindley-Milner* type system for the λ-calculus, which is used for many functional languages and supports parametric polymorphism.

3.2.1 Implicit typing by name

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*.

4.1 Ubiquitous basic types

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**
  - Which, of course, can be stored in a bit, but will usually be stored in at least a word (4 bits) or perhaps even a byte (8 bits) for convenience of memory access.

4.2 Singleton and empty types

- **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`.

        - But this value still implicitely exists; a function of type `void` can certainly return, implying it returns 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 (or should be) **nothing** in the empty type.
        - This might be violated by using error values.
        - For instance, in Haskell, the term referred to as `bottom` or $\perp$ —which refers to any computation which never completes— is a member of all types, even the empty type.
    - No function with return type `Empty` should ever be able return.

4.2.1 Singleton and empty types in Haskell

```haskell
data Unit = Unit -- This type Unit has a single constructor,
 ↪  also called Unit.
data Empty -- Empty has no constructors.

-- Previous versions of Haskell required a workaround to write
 ↪  Empty,
-- as a constructor list was mandatory;
-- this alternate definition has a constructor also called
 ↪  EmptyAlt,
-- but using this constructor requires an argument of type
 ↪  EmptyAlt
-- to already exist, so should never actually be usable.
data EmptyAlt = EmptyAlt EmptyAlt


-- The bottom can be defined by unending recursion.
bottom = bottom

x = Unit

-- We can cheat and get Empty/EmptyAlt "values" by using
 ↪  bottom.
```

```
y = bottom :: Empty
z = bottom :: EmptyAlt

-- Check the types of these values
:t x   -- reports x :: Unit
:t y   -- reports y :: Empty
:t z   -- reports z :: EmptyAlt
```

4.3 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.

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.

4.4 Atom or symbol types

Many languages include a type of *atoms* or *symbols*, which are essentially interned strings.

- Strings which are immutable, and of which there is only one copy in memory.

Specifically, decendents of Lisp and Prolog tend to have a symbol type. For instance, in Ruby:

```
# A symbol in Ruby begins with a :
x = :hello
```

```ruby
# We can intern strings using an intern method
y = "hello".intern

# The equal? method checks if two values are the same object.
# These all evaluate to true, since there's only one copy of
 ↪  the symbol hello.
puts x.equal?(y)
puts :hello.equal?(x)
puts y.equal?(:hello)

# But this to false, because these are two copies of the same
 ↪  string!
puts "hello".equal?("hello")
```

4.5 Ordinal types

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

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

For instance, Pascal supports both enumerations and subranges. (This example based on one from the Free Pascal and Lazarus Wiki.)

```
type
  // An enumeration type; unless specified,
  // the first element is implicitely assigned the ordinal
   ↪  value 0,
  // the second ordinal value 1, etc.
  DaysOfWeek = (Sunday, Monday, Tuesday, Wednesday,
                Thursday, Friday, Saturday);

  // A subrange type.
  // In this case, DaysOfWorkWeek contains the ordinals
   ↪  1,2,3,4 and 5.
  DaysOfWorkWeek = Monday..Friday;
```

4.6 Less common numeric 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.
    * For instance, to store a decimal digit directly (without conversion to binary) takes at least 4 bits.
    * There are 10 possibilities, too many for 3 bits (which can only have 8 different states.)
    * But 4 bits could represent 16 states (6 more than needed.)
    * For memory access reasons, some such implementations even use 8 bits (one byte) per decimal digit.

4.6.1 Complex numbers in C#

(This example code taken from the .NET documentation.)

```
using System;
using System.Numerics;

public class Example
{
   public static void Main()
   {
      // Create a complex number by calling its class
      ↪ constructor.
      Complex c1 = new Complex(12, 6);
      Console.WriteLine(c1);

      // Assign a Double to a complex number.
      Complex c2 = 3.14;
      Console.WriteLine(c2);

      // Cast a Decimal to a complex number.
      Complex c3 = (Complex) 12.3m;
      Console.WriteLine(c3);
```

```
      // Assign the return value of a method to a Complex
       ↪ variable.
      Complex c4 = Complex.Pow(Complex.One, -1);
      Console.WriteLine(c4);

      // Assign the value returned by an operator to a Complex
       ↪ variable.
      Complex c5 = Complex.One + Complex.One;
      Console.WriteLine(c5);

      // Instantiate a complex number from its polar
       ↪ coordinates.
      Complex c6 = Complex.FromPolarCoordinates(10, .524);
      Console.WriteLine(c6);
   }
}
// The example displays the following output:
//       (12, 6)
//       (3.14, 0)
//       (12.3, 0)
//       (1, 0)
//       (2, 0)
//       (8.65824721882145, 5.00347430269914)
```

4.6.2 Decimal numbers in C#

(This example code taken from the .NET documentation.)

```
// Keeping my fortune in Decimals to avoid the round-off
 ↪ errors.
class PiggyBank {
    protected decimal MyFortune;

    public void AddPenny() {
        MyFortune = Decimal.Add(MyFortune, .01m);
    }

    public decimal Capacity {
        get {
            return Decimal.MaxValue;
```

```
        }
    }

    public decimal Dollars {
        get {
            return Decimal.Floor(MyFortune);
        }
    }

    public decimal Cents {
        get {
            return Decimal.Subtract(MyFortune,
             ↪  Decimal.Floor(MyFortune));
        }
    }

    public override string ToString() {
        return MyFortune.ToString("C")+" in piggy bank";
    }
}
```

5 Structured data

Of course, we rarely want to deal only with "atomic" data. More commonly, we are interested in *collections* of data, or *alternatives* between different data types; these are captured in the notion of *structured* data.

Specifically, we consider

- products,
- sequences,
  - including arrays and lists
- "sets"
  - (really associative arrays (maps/hashes/tables))
- and unions/variants.

5.1 Homogeneous or heterogeneous

An important design decision for any structured type is whether it is *homogeneous* or *heterogeneous*.

- "Heterogeneous" structures store elements of differing types.
- "Homogeneous" store only elements of the same type.

5.2 Product types (tuples)

A *heterogeneous* collection of a *fixed* number of elements.

- Implemented by, for instance,
  - `struct`'s or records,
    - both of which have labelled fields,
  - tuples,
    - which are often be implemented as records with specially named fields, such as "`first`" or "`_1`", and
  - classes,
    - which have *methods* as well as fields.
- In lower level languages, programmers may be concerned with the alignment/packing of the data.
  - See this overview of structure packing in C, and the sort of memory inefficiency that can result from poorly ordering the fields of a `struct`.

5.3 Array types

Arrays are an abstraction of finite sequences of adjacent memory cells.

- Programmers are guaranteed (or required to work around) certain properties.
  - `O(1)` access/update time for any element.
    - Implying elements are stored in adjacent memory cells (a *contiguous block* of memory cells.)
    - Also implying each element is of a known, usually constant, size.

        - Heterogeneous arrays can store *references* to elements of differing types instead of elements of those types themselve to maintain this property.
      - To perform a lookup, just look `element_space * index` bits past the start of the array.
  - `O(n)` insert time (if inserting is possible.)
    - The (possibly `n`-many) elements after the insertion point must be shifted.
  - It may be computationally costly or impossible to modify length.

We can classify arrays by where and how their memory is allocated.

5.4 Classification of arrays based on memory allocation: data segment and stack

- Static arrays
  - (Memory) allocation is static, and presumably in the data segment.
  - Subscript ranges are statically bound.
- Fixed stack-dynamic arrays
  - Allocation is dynamic and on the stack, done when the declaration of the array is reached.
  - Subscript ranges are statically bound.
- Stack-dynamic arrays
  - Allocation is dynamic and on the stack, done when the declaration of the array is reached.
  - Subscript ranges are dynamically bound when the declaration is reached.
  - After declaration, subscript range and storage remain fixed.

5.5 Classification of arrays based on memory allocation: heap

- Fixed heap-dynamic arrays
  - Allocation is dynamic, done when the user program requests.

  - Subscript ranges are dynamically bound at allocation time. After allocation, subscript range and storage remain fixed.

- Heap-dynamic arrays
  - Allocation is dynamic, done when the user program requests.
  - Subscript ranges are dynamically bound at allocation time.
  - Subscript range and storage can change throughout runtime!
  - These are commonly implemented as *array-lists*, discussed shortly.

5.6 List types

Lists are simply an abstract notion of sequences.

- May be implemented by arrays or by structures such as linked lists.
- Often we do not `O(1)` access time for lists.
- But we do have better flexibility;
  - appending or prepending an element into a linked list only requires changing a pointer,
  - and inserting into a list only requires changing `O(n)` pointers.

Persistance of data is more feasible using linked lists; parts of the list can be reused.

- For instance, two immutable lists `[1,2,3,4]` `[5,6,3,4]` can share the storage for the `3` and `4` elements.

Lazily (non-strictly) constructed lists may even be "infinite". For instance, the infinite list of 1's in Haskell:

```
ones = 1 :: ones
```

5.7 Array lists

An *array list* type provides the flexibility of a list, but the performance of an array (except for the occasional reallocation.)

- Array lists are stored in a continguous block of memory cells.
- And the block is just reallocated when the array grows too large.

  - Reallocating memory is costly, and it requires copying the elements to the new memory, and so there is a performance dip when this is needed.
  - Generally, *twice as much* memory as is needed is allocated.
    * So as the size of the list grows, so does the extra memory allocated at each reallocation.
    * This mitigates the cost of reallocation; it becomes less frequent as time goes on.

5.8 "Sets"

It is notoriously difficult to represent unordered collections such as sets and bags on computers.

- Computers are extremely ordered machines; how do we store unordered collections in ordered memory?
- When available, "set types" are usually implemented using *trees* or *associative arrays* (discussed next.)

5.9 Associative array (map, hash, table) types

Associative arrays, also called *hashes*, *maps* or sometimes *tables*, are sets of key/value pairs.

- Abstracts away the ordering of the sequence.
  - (Though we could order the keys, and so impose an order on the collection.)
- The programmer can imagine they are lists of key/value pairs.
  - And they may even be implemented that way.
    * But in practical languages, they are more often implemented by sorting elements into "buckets" by a hashing function on the keys.
    * The implementations get quite complex.

5.10 Unions, variants

Whereas an element of a product type contains

- a collection of elements of some types,

a *union* or variant type contains

- one element of a selection of types.

Unions can be *tagged* or *untagged*.

- With an untagged union, the runtime does not keep track of the underlying type of the element.
- Whereas a tagged union uses a *tag* on the value to identify the underlying type of the element.

Note that union types are unnecessary in dynamically type checked language.

- We can think of every variable/argument/procedure's type as being a union of all possible types.

5.11 Untagged unions

As we've said, with an untagged union, the runtime does not keep track of the underlying type of elements.

- So it's type is dynamic! (Amongst the types involved in the union.)
- Languages that provide untagged unions (today, mainly C and C++) do not even dynamically check the type.
- Accessing it as the wrong type simply treats the bits as if they were of that type; it is not a cast!
- This is *very* unsafe; it allows for type clashes.

5.11.1 An example usage of untagged unions in C++

We can see the danger of untagged unions in a short C++ example; we are legally allowed to interpret the bits of an integer as if they were the bits of a floating point, which has a very different arrangement in memory!

```cpp
#include<iostream>

union foo {
  int a;
  float b;
};

int main() {
  foo x;

  x.a = 1; // Set x as an integer.

  std::cout << x.b; // Treat x as a float, even though it's an
   ↪  int right now.
                    // The typechecker does not complain!

  // Outputs 1.4013e-45 during my testing;
  // the result of interpreting the bits of x as a float.
}
```

For a more practical usage of untagged unions, see the example implementation of a language of integer expressions in C++, provided in homework 6.

5.12 Tagged unions and pattern matching

Tagged unions are also known as

- *sum* and *either* types, or
- as *variant* types when the labels are chosen by the programmer.

The introduction of tags or labels makes *pattern matching* a viable and useful control structure.

Given a variant type such as

```haskell
data Foo = A Int | B Float
```

We can *match* on the tag, using placeholder variables for the values of the underlying types.

```haskell
bar :: Foo -> Int
bar (A i) = i   -- i is a placeholder for the integer value.
bar (B _) = 0   -- An example of a nameless placeholder, _.
```

Many languages also allow the use of constants instead of placeholder variables, to create more specific cases.

5.13 Recursive unions

If a union type is allowed to be *recursive* (meaning that one of the underlying types of the union is allowed to be the type being defined as the union), then union and product types together allow for the definition of *algebraic datatypes* (discussed later.)

A well-known example is to define lists as a recursive union of products.

```
data List = Cons (Int, List) | Empty
```

(Using a type parameter and currying, we get this more flexible definition more in the style of most functional languages.)

```
data List A = Cons A (List A) | Empty
```

6 References

Pointer and reference types capture the notion of a memory address.

- Not just alternate namings! They have very different properties.
- Pointers are a numerical representation, and can be manipulated as numbers to access *adjacent* memory locations.
  - Pointer arithmetic.
  - Note: adding `1` to a pointer does not shift by `1` bit; it shifts by the size of the type being pointed to.
- A reference is a more abstract notion; it cannot (or at least should not) be manipulated as a pointer can.

6.1 Referencing/dereferencing

In a language with reference types, we typically have

- a *dereference* operator to access the value stored at that reference
  - (written `*` in C and many C-like languages, and
  - written `!` in ML and languages descended from it),

and we often have

- a *reference* operator to obtain a reference to a stored value
  - (written `&` in C and many C-like languages.)

6.2 Garbage collection and dangling references

When discussing memory binding, we previously discussed the concept of *garbage collection* and mentioned another related problem:

- *dangling* or *wild* references.

A dangling or wild reference is a reference value which refers to a memory location that has already been deallocated.

- The contents of deallocated memory are usually not specified by the language, so accessing it can result in undefined behaviour.
- This *should* only occur if the language allows explicit deallocation of memory (instead of only using implicit deallocation through garbage collection.)

There are mechanisms to prevent against the use of dangling references.

6.3 Preventing the use of dangling references; tombstones

With the *tombstone* approach, every reference is in fact a reference to a cell of memory called a *tombstone*, which then refers to the value being stored in memory.

- So there is an additional level of indirection.
- When memory is deallocated, the tombstone is left, but its reference is set to a null pointer.
  - So it can no longer be used; accessing a "nulled" tombstone will result in an error.
- This is costly both in time (two dereferences are needed for each access) and in space (the tombstone must remain indefinitely.)

6.4 Preventing the use of dangling references; locks and keys

With the *lock-and-keys* approach, a reference value is in fact a pair of a special *key* value and the reference itself.

- And referenced memory cells have allocated with them a special *lock* value which matches those keys.
- When memory is freed, the lock value for that cell is wiped.
- So the keys of any remaining references are no longer valid.
- This is also costly in time
  - (a comparison must be made between the lock and the key at each access)
- and in space
  - (the lock and key values take space on every reference and every memory cell)
- but not so much as tombstones.

7 Further advanced topics

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