4 – Introduction to Types

Mark Armstrong, PhD Candidate, McMaster University

Fall, 2019

1 Preamble

1.1 Notable references

- Concepts of Programming Languages, Sebesta, 10th ed.
  - Chapter 6 – Data Types

1.2 Update history

Nov. 12th
- An error was corrected regarding product types; they are *heterogeneous* collections, not *homogeneous*.
- Slight cleanup; added references and table of contents.

Oct. 3rd
- Original version posted

1.3 Table of contents

2 What is a type?

Types are the *most* important abstraction in programming.

- Without types, the programmer deals entirely in bitstrings.
  - I.e., the programmer must keep a mental map of representation of data.

> Types are the central organising principle of the theory of programming languages. Language features are manifestations of type structure. The syntax of a language is governed by the constructs that define its types, and its semantics is determined by the interactions among those constructs. The soundess of a language design — the absence of ill-defined programs — follows naturally.
>
> — Robert Harper, *Practical Foundations for Programming Languages*

2.1 But what are types?

A *type* is simply a collection of values. Recall that in set theory, a set is a collection of values.

- In fact, we will see when we begin using Agda that it uses the word `Set` instead of `Type`.

You have seen previously many methods to construct sets:

- Listing their members (for finite sets).
- Giving the means of constructing elements, for instance by an inductive definition.
- Combining other sets.
  - Unions, $A \cup B$.
    - *Disjoint* or *tagged* unions, $A \uplus B$.
  - Intersections, $A \cap B$.
  - Products, $A \times B$ and A^n.
  - (Finite) sequences, A^*, and infinite sequences A.
  - Functions, $A \to B$.
  - Others:
    - Differences, A/B.
    - Power sets, A.

2.2 The common methods to construct types

That list of methods for constructing sets contains a lot, not all of which are necessarily practical in the context of types in programming languages.

We will see that most languages include some *primitive*, (also called *atomic* or *basic*) types, along with these ways to combine types to form new types:

- Products
- Sequences
- Unions
- Functions

2.3 Exercise: tuples vs lists

What is the difference between

- A^n, the set of n-ary products of A (i.e., $A \times A \times \ldots \times A$), and
- $A*$, the set of sequences of A?

Why would we want both types in our language?

3 Classifications of languages based on types

Before we discuss types themselves further, lets discuss various classifications of languages based on types.

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".
- Not a good objective criteria, by that definition.
  - 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.

3.1.1 Exercise

Try for yourself: pick a language "X", and search for

"is X strongly typed?"

Try to

- find arguments for and against it, and
- evaluate the quality of the arguments.
  - Of course your ability to do this depends on your familiarity with the language.

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.

3.3 Static and dynamic typing

Are types checked before or at runtime?

- It's somewhat natural for interpreted languages to be dynamically typed.
  - Consider the interpreted "scripting" languages; Python, Ruby, Javascript, Lua, Perl, PHP, etc.
  - But it's far from universally true!
    - Haskell has an interpreter, and is definitely statically typed.

"Dynamically typed" is a misnomer.

- It would be better to say "dynamically type checked".

4 Subtyping and polymorphism

Being strict about types (preventing type clashes) introduces a (solveable) problem; it prohibits code reuse!

- Subroutines can only be used on a particular type of arguments.

Subtyping and polymorphism provide solutions to this.

- With **subtyping**, there is a sub/super relation between types.
  - Consider the notion of *subsets*.
  - Subtypes can be used in place of their supertypes.
  - Sub-*classing* is one instance of subtyping.
  - As are subrange or, sometimes, enumeration types.
- With **polymorphism**, a subroutine can have several types.

One notion of polymorphism is **ad hoc** polymorphism, also called **overloading**.

- Not actually a feature of a type system.
- Subroutine names can be reused as long as the types of arguments differ (in some specified way).
- So, the programmer must define the "same" subroutine many times.

4.1 Parametric polymorphism

With parametric polymorphism, subroutines have a *most general* type, based on the *shape* or *form* of their arguments.

- The subroutines behaviour can only be based on the form, not the specific types.
- Commonly used in functional languages.

4.2 Duck typing

More formally called *row polymorphism*.

- Types are not actually checked.
- We only check that an entity has the correct method defined for it.
- "If it walks like a duck, and quacks like a duck, it's a duck!"

5 Primitive types

Most languages include some subset of these *primitive*, (*atomic*, *basic*) 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 nil, null or none type, which have a single value (called `nil`, `null` or `none`), there is **nothing** in the empty type.

5.1 Uncommon basic types

A few languages include these 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)

5.2 Ordinal types

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

- Enumeration types
- Subset/Subrange types

6 Sequence types

Recall from set theory that

- sets are collections of elements where order and multiplicity "don't matter",
- bags are collections of elements where order "doesn't matter" (but multiplicity does), and
- sequences are collections of elements where order and multiplicity both matter.
- (This is part of the *boom* hierarchy).

There are multiple ways sequence types are represented in programming languages.

6.1 Homogeneous or heterogeneous

One design decision for any sequence (or more generally, any collection) type is whether it is *homogeneous* or *heterogeneous*.

- Can it store elements of differing types?
  - "Heterogeneous"
- Or only elements of the same type?
  - "Homogeneous"

6.2 Array types

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

- Programmers are guaranteed certain properties.
  - `O(1)` access time for any element.
  - May be computationally costly or impossible to modify length.
    - There are different classifications of arrays.
    - "Array lists" alleviate this problem.
    - We'll discuss this more later in the course.
  - The programmer may have to handle memory allocation.

6.3 List types

Lists are simply an abstract notion of sequences.

- May be implemented by arrays or by structures such as linked lists.
- Implementation details and properties may not be guaranteed.
  - Often not `O(1)` access time.

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

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

6.4 Associative array (hash, map, table) types

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

- Abstracts away the ordering of the sequence somewhat.
  - Though we could order the keys, and so impose an order on the collection.
- Programmer can imagine they are lists of key/value pairs.

A quick side note about sets and bags

- It is notoriously difficult to represent such unordered collections on computers,
  - Computers are extremely ordered machines.
- When available, "set types" are usually implemented using trees or hash tables.

6.4.1 Exercise: Objects as associative arrays

Consider:

- How can we represent objects as a homogeneous array? (Not classes, objects).

6.5 String types

Strings are simply sequences of characters.

- Often they are built in,
  - but they could be excluded because programmer can implement them using characters and other sequence types.
- Probably they should be built in, or at least part of standard libraries, since they are so commonly used.

7 Product types

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

- Implemented by, for instance:
  - `struct`'s
  - Records
  - Tuples
    - Can be implemented as records with unnamed fields
  - Classes
- In lower level languages, programmers may be concerned with the alignment/packing of the data.

7.1 Exercise: how many elements in a product?

Consider two types A and B.

- How many elements are in the product type $A \times B$ (a product with one element of each).

7.2 Exercise: classes as a product type

Consider:

- How can we represent classes as a record or tuple? (Not objects, classes).

8 Union (sum) types

Whereas an element of a product type contains

- a collection of elements of some types,

a *union* type contains

- one element of a selection of types.

Unions can be *tagged* or *untagged*.

- With an untagged union, we have no idea *which* of the possible types it is storing at any time.

  - So it's type is dynamic! (Amongst the types it can store).
  - This is unsafe; it allows for type clashes!

- Whereas a tagged union keeps a *tag* identifying which type of element it is currently storing.

Tagged unions are also known as

- *sum*, *option* and *either* types.

Note that union types are unnecessary in a dynamically typed language.

9 Pointer and reference types

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

- Not just alternate namings! They have very different properties.
- Addresses are an extremely low level construct.
  - Pointers are fairly low level themselves.
  - References are more abstract.
- Present numerous challenges.
  - Aliasing
    - Especially through pointer arithmetic; given a pointer as an "address", we can access "nearby" memory!
  - *Dangling* or *wild* pointers/references
  - Garbage

10 Where do we go from here?

- We will continue to discuss types; specifically,
  - the concept of an *abstract* datatype,
  - *inductive* (or algebraic, or recursive) datatypes,
    - along with some introductory *type theory*,
  - more details about type implementations,
    - especially arrays.