Types and Types Systems

nprenet@bsu.edu

CS431, Spring 2022

1. Types

1.1 Definitions

A **type is a set**: it represents the range of values that a variable can assume during the execution of a program.
E.g. a variable x of type *Boolean* is supposed to assume boolean values during every run of a program. If x has type *Boolean*, then the boolean expression $not(x)$ has a sensible meaning in every run of the program.

- All variables that belong to the set are encoded the same way by the machine. E.g. an `int` (C, C++, Go) is encoded as a 64-bit integer, a `char` as a 16-bit value, etc.
- Types come with collections of specific **operators** that the language supports for them: for instance, a `String` type in Java may be the operand of a concatenation (`+`); a numerical type may be involved in arithmetical operations (`+`, `-`, `*`, `/`)...

A language where variables can be given types is called a **typed language**, to be contrasted with **untyped languages**, where there is no type, or a universal type that contains all values: in such languages, operations may be applied to inappropriate arguments. E.g. assembly language.[1]

A **type system** is the component of a typed language that keeps track of the types of variables and, more broadly, of all expressions in the language: program sources that do not comply with the type system are discarded before they are run (the compilation fails, that is).

Typed languages may be

- **explicitly typed** if types are part of the syntax E.g. `int i = 2;` (C), `String[] words;` (Java)
- **implicitly typed** otherwise; among those
  - languages where the type of variable is checked **statically** (i.e. at compile time), based on the compiler's type inference system. E.g. ML, Haskell, that support writing program fragments with no type information.
  - languages where the type of variables is checked **dynamically**, at runtime, because a variable may have any type. E.g. Scheme[2], Python, Javascript.

[1]The pure λ-calculus is an extreme case of an untyped language where no fault ever occurs: the only operation is function application, and since all values are functions, that operation never fails.

[2]Among type theoricians, Scheme has sometimes been referred to as an untyped language (f.i. Cardelli, 1997), but that is misleading, and probably based on the outdated assumption that all typed languages are statically checked. In Scheme, as well as in the more popular dynamically checked languages that came on its heels (Python, Javascript, Ruby, ... to name a few), variables do have a type, even if it may not be known until runtime. E.g. As an illustration, consider this Scheme expression:

```
> (let ((x 2))
      (+ x "he"))
Exception in +: "he" is not a number
```

1.2 A Menagerie of Types

Primitive vs. Constructed types

Definition. [Built-in vs. user-defined] Any type that a program can use but cannot define for itself is a **primitive type** in the language: it is directly supported by the compiler. Any type that a program can define for itself (using the primitive types) is a **constructed type** (or derived type).

In some languages, this distinction largely overlaps with another one, that opposes **scalar** types, that stand for low-level, fixed-size data types in the machine, and **aggregate** types (or compound types), that are built from smaller components. E.g. In C/C++, the primitive types `bool`, `char`, `int` are all scalar types. The user may construct aggregate types by declaring arrays of `int`, `char`, structures, vectors...

However, many recent languages (Scheme, Python, Javascript, Go) have **built-in constructors for aggregate data** such as strings and sequences, objects, and functions (see below).[3]

Language	Primitive types	Constructed types
C/C++	`bool`, `char`, `int`, `double`	arrays, functions, pointers, structures, unions
Go	`bool`, `int`, `float`, `complex`, `rune`, `string`	arrays, slices, maps, structs
C#	`byte`, `int`, `char`, `decimal`, `object`, ...	arrays, classes,
Haskell	`Int`, `Integer`, `Char`, `double`	Bool, lists, custom types
Python	`int`, `bool`, `float`, `complex`, (`str`, `list`, `dict`, `set`, `tuple`: type constructors)	types constructed with `str`, `list`, `dict`, `set`, `map`...
Javascript	`string`, `number`, `boolean`, `null`, `undefined`, `object`	types constructed with String(), Number(), Boolean(), Function(), Array(), ...

(see Classroom exercise #1)

Because types are sets, we can categorize constructed types by describing them in terms of set operations.

Constructed types: sum types

A sum type is the sum or union of two or more sets: $A \cup B \cup \ldots$. Such a type may be

- an **enumeration**, that lists all inhabitants of the type. E.g. in Haskell (see Classroom exercise #3)

  ```
  data Weekday = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday
  ```

  The type is actually the union of the singleton sets $\{Sunday\}, \{Monday\}, \ldots$ In Haskell, the data constructors hide the implementation and create values that stand for themselves, as in `data Bool = True | False`. In Go, the enumeration is just a subset of integer constants:

  ```
  type Weekday

  const {
          Sunday Weekday = iota // where iota is 0 by default
          Monday                // iota + 1
          Tuesday               // iota + 2
          ...                   // ...
          Saturday
  }
  ```

  Java and C# implement enumerations as classes, where each member of an `Enum` type is an instance of its class.

[3]From the language user's point of view, they represent aggregate data; from the language implementer's perspective, they actually are scalar data, i.e. pointer types. E.g. a `str` variable in Python is an address.

- a **union**, i.e. a type that stores only one value, that may belong to different types. E.g. This union type in C stores one value, that may be either an array of `int`, or a `float`:

  ```
  union array_or_fractional {
          int numbers[SIZE];
          float fractional ;
  } aof ;
  ```

  In memory, variable `aof` is allocated enough space to store the largest type (the `int[]` type). During a read operation, C programmers must take great care of reading the type that has been written last, lest they get corrupted data.

  Exercise: In Haskell, how would you create a sum type that can be either a list of `Int` (`[Int]`) or a `Double`? (See also Classroom exercise #4.b)

  Answer: The following would *not* compile, because the RHS of a Haskell type constructor is made of *data* constructors, not types:

  ```
  data ArrayOrFractional = [Int] | Double -- does not compile!
  ```

  However, the code below is correct:

  ```
  data ArrayOrFractional = Array [Integer] | Fract Double
   --         [a]                [b]               [c]

  Prelude> x = Array [1,2,3]
  Prelude> :type x
  x :: ArrayOrFractional
  Prelude> y = Fract 1.2
  Prelude> :type y
  y :: ArrayOrFractional
  ```

  Expressions `[b]` and `[c]` do not define types: they are both *data* constructors, that take infinitely many possible values as parameters (respectively, `Integer` and `Double` values). The type `ArrayOrFractional` is the set of all inhabitants that can be created through either constructor.

Constructed types: product types

A product type is the Cartesian product of two or more sets: $A \times B \times \ldots$, i.e. the set of all tuples t, where the first element of t is an element of A, the second element of t an element of B, ... (see Classroom exercise #4.a)

- **Tuples** are sequences that store a fixed number of values, that may belong to different types. E.g. In Haskell, a tuple may created with the `(,)` constructor:

  ```
  :type (1, 2.3, "ABC")
  (1,2.3,"ABC") :: (Fractional b, Num a) => (a, b, [Char])
  ```

  In the code above, the type is implicit. For ease of use, we could create a type alias as follows:

  ```
  type MyPrettyTuple = (Integer, Double, String)
  ```

  ... or a full-fledged, named custom type (see Classroom exercise #5):

  ```
  data MyPrettyTuple = MyPrettyTuple (Integer, Double, String)

  Prelude> myTpl = MyPrettyTuple (1, 2.3, "ABC")
  ```

  Lisp, and its most popular dialect, Scheme, has a built-in constructor for creating so-called list types: implemented as linked lists made of primitive pairs, they allow for creating sequences that store heterogeneous values (integers, strings, functions, or other lists). Both Python and Javascript implement similar constructors.

In Python, two primitive constructors allow for creating tuple-like types: `list` and `tuple`. Both kind of objects may store values belonging to different types and support all read operations you would expect to be defined on arrays (subscript, slices, ...). However, list types are mutable, when tuples are not.

```
>>> my_pretty_tuple = (1, 2.3, "ABC")
>>> del my_pretty_tuple[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
```

In Javascript, similar types can be constructed with the built-in `Array()` constructor, i.e. a function.

- **Tuples with named components** allow for referring to their members. In C, we use structures:

```
struct idc {
 int i;
 double d;
 char * s; // pointer to char ~ string address
}
...
idc x;
x.i = 1;
x.d = 2.3;
x.s = "ABC";
```

In this C structure, all members are stored contiguously in memory, in the same order they have been declared: the `int` member first, followed by the `double` value and the `char` pointer (the string itself is allocated on the heap). Javascript objects also belong to the named tuple category:

```
var myRecord = {
        i: 1,
        d: 2.3,
        s: "ABC"
}
> myTuple.i // 1
> myTuple.d // 2.3
```

In Haskell:

```
data MyRecord = MyRecord { int :: Integer, db :: Double, st :: String }
Prelude> MyRecord 1 2.3 "ABC"
MyRecord {int = 1, db = 2.3, st = "ABC"}
```

In Java, C#, and all OO languages, all **classes are tuples**!

- **Lists and arrays** are a particular kind of Cartesian product: $A^* = A \times A \times \cdots \times A$.
  - when the size is fixed, the type A^n is a particular kind of tuple, where all elements in the tuple have the same type
  - types of variable size A^*, where the sequence may be updated

  The concept is simple but leads to an amazing variety of types:

  - Haskell lists, that are immutable (e.g. `[Char]`) (like every value in Haskell)
  - C/C++, Java/C# fixed-size, mutable arrays (e.g. `int[]`), whose elements may be accessed through subscript operators (e.g. `myArray[2]`) and updated.
  - C++/Java/C# containers, i.e. classes of object that store objects of the type (or type class), and support dynamic operations: insertion and deletion of elements.
  - In many languages (Haskell, C/C++, Java, ...), plain strings are just a particular kind of array; in Lisp/Scheme and Javascript, they are supported as primitive values.

3. Types and safety

Safe vs. Unsafe

The fundamental purpose of a type system is to prevent the occurence of **execution errors** during the running of a program. There are two kinds of execution errors:

- **trapped errors** have obvious symptoms: they cause the computation to stop immediately. E.g. Calling a function on an object that does not support it; accessing an illegal address; divide by zero.
- **untrapped errors** go unnoticed and later cause arbitrary behavior. E.g. accessing past the end of an array in absence of runtime bounds checks; branching to the wrong address; reading truncated data.

A program fragment is **safe if it does not cause untrapped errors** to occur. A language is safe if all fragments in it are safe.

However, the purpose of a type system is also to rule out large classes of trapped errors.

Well-behaved vs. Ill-behaved program

A language is **well-behaved** if it forbids

- all untrapped errors, to make the language safe (minimal requirement—see definition above)
- a subset of trapped errors, as designated by the language designers

A language where all (legal) fragments have good behavior is called **strongly checked**.

- **statically typed languages** may enforce good behavior (including safety) by checking types at compile time, thus preventing ill-behaved programms from running. E.g. Pascal, ML, Haskell, Java, C#
- other languages may enforce safety by **dynamic typechecking** at runtime: for instance by checking all array bounds, all division operations, but also by ensuring that a variable may be involved only in those operations that are supported by the type it contains at that moment. E.g. Lisp/Scheme, Python, Javascript, Ruby.
- Few languages with static typechecking can dispense with runtime checks: array bounds must be tested dynamically for instance; in Java and C#, **type introspection** allows for the runtime interpreter to obtain information about a given object (`instanceof`)

It is tempting to conflate safety with static type checking, but this is not correct.

- some statically checked languages are **weakly checked**[4], because their set of forbidden errors does not include all untrapped errors.

  E.g. C is the prime example of a language where some unsafe operations are not detected at compile time: the language has many unsafe features (such as pointer arithmetic and casting) that the programmer should only use with great care. This was a deliberate choice by the language designers[5], because of **performance considerations**: safety has a cost, even when checks are done at compile time. However, since unsafe features make the system vulnerable to attacks,[6] the resulting **lack of security** may have a greater cost in code maintenance.

[4]Or weakly typed, in the older literature.

[5]The C language was born from Dennie Ritchie's and Ken Thompson's work on the Unix system at Bell Labs between 1970 and 1972. The first version of the system was written in assembler, but this proved awkward and time consuming. Thompson wanted the advantages of a high-level implementation language, without the PL/I performance issues that he had seen on Multics—the ambitious, and ultimately failed precedecessor to Unix. Thompson created the language B by simplifying the research BCPL so its interpreter would fit the PDP-7 8k memory. Like BCPL, B had only one type (a machine word), and the ability to decompose an array reference into pointer-plus-offset references. However, an untyped language proved unsuitable to the PDP-11: introduced in 1970, this architecture provided hardware support for datatypes of different sizes: while Thompson was busy reimplementing Unix on the PDP-11 in assembler, Dennis Ritchie created the "New B", which solved both issues of datatypes and performance. It was compiled, not interpreted, and rose to fame as "C"—see Peter Van Der Linden, *Expert C Programming*, 1994.

[6]Typical: buffer overflow or underflow, reading too much, or not enough, of a memory location.

- interestingly, **dynamically checked languages (Scheme, Python) are, by necessity, completely safe**: once facilities are in place for enforcing the safety of *most* operations at runtime, there is little additional cost to checking *all* operations.

	Statically checked	Dynamically checked
Safe	ML, Haskell, Java, etc.	Lisp, Scheme, Javascript, Ruby, Python, ...
Unsafe	C, C++, (Go)[7]	

Examples and exercises

See Exercise "Unsafe C" in this week's worksheet, and key.

[7]Regarding pointers, Go strikes a compromise: subroutines may be passed a pointer to a variable *and* may update the variable (which is not allowed in modern languages using references only). However pointer arithmetic, a major source of bugs in C programs, is not part of the language.