Programming language design and analysis

Types

Marius Minea

6 October 2014

Types: what and why ?

A simple definition (first programming course):
"Type = set of values together with some operations on that set"

A trivial error (at the ML prompt):

```
# (+) 3 (fun x -> x);;
Error: This expression should not be a function,
the expected type is int
```

⇒ Some (syntactically correct) programs do not make sense

Types: a filter for bad programs

Generalizing:
"A *type* is any property of a program that we can establish without executing the program" Krishnamurthi, PLAI book

Type system
a mechanism for distinguishing good programs from bad (informally)

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

Can we always type things ?

```
(+) 1 (if unknown then 3 else (fun x -> x))
```

would run OK if `unknown` is `true`

would give an error otherwise

$\Rightarrow$ can't (always) decide

Type systems are always prey to the Halting Problem.
$\Rightarrow$ a type system for a general-purpose language must always either over- or under-approximate:

either must accept programs that will error when executed

or must reject programs that might have run without error

Krishnamurthi, PLAI

Types as a means to organize

[Cardelli and Wegner,
On Understanding Types, Data Abstraction and Polymorphism]

The following are *untyped* universes

bit-strings in computer memory
  everything represented as bit-strings $\Rightarrow$ untyped/only one type

S-expressions in (pure) Lisp
  no distinction between program and data
  but: some structure (more than bit-strings)

λ-expressions in the λ-calculus
  everything is a function (numbers, booleans, if-then-else)

Sets in set theory
  everything is an element or a set (can encode mathematics...)

Even simple things can be classified

Bitstrings can represent *operations* or *characters*, *integers*, ...

Some S-expressions are *lists*, others are LISP *programs*

Some $\lambda-$ expressions (functions) represent *booleans*, or *integers*

Some sets may denote *ordered* pairs, leading to *functions*
$\Rightarrow$ Can think of *untyped* universes as *typed*
But this is an illusion unless there is some means to enforce it

Typing may be:
explicit (types part of syntax, e.g. all variables typed)
implicit (can be reconstructed: type inference)

Types as protective mechanism

Types avoid problems related to exposing internal representation

Types impose constraints which help to enforce correctness

Types avoid logical inconsistencies ("set of all sets")

Types prevent inconsistent interactions between objects

"A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use." Cardelli & Wegner
"Violating the type system involves removing the protective set of clothing and operating directly on the naked representation."

Types, execution errors and safety

A program might have: [Cardelli, Type Systems]

- *trapped errors*: cause computation to stop
- *untrapped errors*: may go unnoticed

A program (fragment) is *safe* if it does not cause untrapped errors.

A *safe* language: all program fragments are safe.

But, we want more...

- no untrapped errors
- no trapped errors that we consider *forbidden errors*
- programmer must avoid other trapped errors

Static and Strong Typing

Static Typing

type of every expression can be determined by static analysis
- at compile-time, e.g, ML, Java, Pascal (partly unsafe)
- well-typed programs are well-behaved (conservatively)

Strong Typing

Languages in which all expressions are type-consistent
- although type itself may be statically unknown
- can be done by introducing some run-time type checking

Weak Typing (weak checking)
- some unsafe operations detected
  - Pascal: untagged variants and function parameters unsafe
  - Modula-3: separates safe/unsafe modules

Kinds of Polymorphism

Strachey (1967) defines:
– *parametric polymorphism*: function works uniformly on a range of types (with some common structure)
– *ad-hoc polymorphism*: function works on several different types (may not have common structure), may behave in unrelated ways

Refined classification [Cardelli and Wegner]:

- Polymorphism
  - universal
    - parametric
    - inclusion
  - ad-hoc
    - overloading
    - coercion

4 kinds of polymorphism, revisited

- Coercion:
  a single abstraction serves several types through implicit type conversion
- Overloading:
  a single identifier denotes several abstractions
- Parametric:
  an abstraction operates uniformly across different types
- Inclusion:
  an abstraction operates through an inclusion relation

[Wm. Paul Rogers, Reveal the magic behind subtype polymorphism, JavaWorld, 2001]

Ad-hoc polymorphism

Overloading

different functions with same name; context used to make decision
could view as syntactic abbreviation handled by preprocessing
e.g. multiple methods with same name, if signatures are distinct

Coercion

semantic operation, converts a type to that expected by a function
(otherwise type error would occur)
can be done statically or dynamically

Distinction blurred at times. Discuss:

$3 + 4$ $3.0 + 4$ $3 + 4.0$ $3.0 + 4.0$

Universal polymorphism

Parametric polymorphism

Use of a single abstraction across different types

e.g. list abstraction `'a list`

Inclusion polymorphism

subtyping and inheritance

Duck Polymorphism

"when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

a form of dynamic typing

- concerned with just the aspects of an object that are used, rather than the type of the object itself (entire interface).

Offers more freedom (polymorphism without inheritance)

Does not define an explicit interface

Can result in semantic unintended behavior