

Type Systems

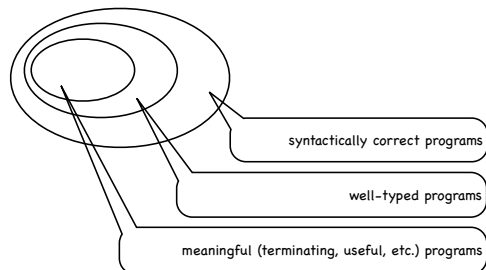
©Andrey Kruglyak, 2010

Example

- ◀ $y = x+3$
- ◀ If x is a numeral, this will work
- ◀ But what if x is a String? or a struct? The program will either crash or calculate a wrong value
- ◀ However, this can be avoided using types

Types

- ◀ We can use types to eliminate such errors at compile time or to detect them at run time
- ◀ The set of well-typed programs is an approximation of the set of correct (meaningful) programs



Well-typed terms and programs

- ◀ A term or a program can be well-typed (= there exists such a type T that the term t is of type T , denoted " $t :: T$ ")
- ◀ Two questions arise:
 - ◀ how do we define types?
 - ◀ how do we determine if a particular term t is well-typed?

How do we define types?

- ◁ Some languages do not have types
- ◁ Some languages only have built-in (primitive) types
- ◁ Some languages define the types automatically, not allowing the programmer to declare types
- ◁ Some languages let/force the programmer to declare certain types but automatically construct other types

Tuesday, April 27, 2010

5

How do we define types?

- ◁ C: primitive types + struct types (declared & implemented) + function types (declared & implemented)
- ◁ Java: primitive types + interface types (declared) + class types (declared & implemented) [+ enum types + generic types]
- ◁ Timber:
 - primitive simple types (`Int`, `Float`, `Char`, `Bool`, `() "unit"`)
 - + primitive type constructors [`a`], `(a,b)`, `Array a`, `a -> b`, `Request a`, `Class a`, `Cmd s a` - these can be automatically instantiated to specific types: `[Int]`, `(Int,Char)`
 - + struct and data types (have to be declared)
`data Day = Mon | Tue | ...`
 - + struct and data type constructors (have to be declared)
`data Maybe a = Just a | Nothing`
these can be automatically instantiated to, e.g. `Maybe Int`

Tuesday, April 27, 2010

6

Typing rules

- ◁ Typing rules are used to calculate the type of a term from its parts...

$$\frac{a :: \text{Int} \quad b :: \text{Int}}{a + b :: \text{Int}} \quad (\text{Sum}) \qquad \frac{a :: T_1 \rightarrow T_2 \quad b :: T_1}{a \ b :: T_2} \quad (\text{App})$$

- ◁ ...or to derive the requirements on the type of the parts from the type of the term, e.g. if `(a+b) :: Int` we can derive from the rules above that it requires that both `a` and `b` are of type `Int`
- ◁ A term is well-typed if its type can be derived using typing rules

Tuesday, April 27, 2010

7

Soundness of a type system

- ◁ A type system is a collection of
 - rules for defining types
 - typing rules
- ◁ Given the syntax of a language, its operational semantics (evaluation rules) and the typing rules, soundness of a type system can be proved by proving two properties:
 - ◁ Progress: a well-typed term is not stuck
For example, `if 4 then 0 else 1` cannot be evaluated further - and it should be excluded by typing rules
 - ◁ Preservation: if a well-typed term takes a step of evaluation, then the resulting term is also well-typed

Tuesday, April 27, 2010

8

The error term

- ◁ But what about partial functions, such as division, where division by zero results in an error? How can x/y be well-typed?
- ◁ One answer is to define the division (or any other similar operation) to return either an integer or a special error value (an exception)
- ◁ Then we can let the error value belong to all types at the same time, but of course we will need to adjust our evaluation and typing rules to allow the exception to propagate to the top of the evaluation tree

Tuesday, April 27, 2010

9

Type systems

- ◁ (Static) explicit typing: types of all variables, functions, etc. are declared by the programmer; the types of all expressions can then be checked statically, at compile time:

```
int x, y;  
Object o;  
y = x+3; => okay  
o = y;    => type error, incompatible types
```

- ◁ (Static) implicit typing: types of all variables, etc. are (automatically) inferred from operations used on variables and these types are checked for consistency (that these requirements agree with each other):

```
y = x + 3; => both x and y must be numerals  
o = y.m;  => y must be a struct with selector m  
=> incompatible types for y: numeral AND struct
```

Tuesday, April 27, 2010

10

Type systems

- ◁ Dynamic typing: instead of checking the types of variables during compilation, each value is tagged at run time with a type and type compatibility is checked before an operation on the value is performed

Tuesday, April 27, 2010

11

What's your preference?

- ◁ Can you give examples of languages from each group?
- ◁ Which approach appeals to you the most?

explicit typing
implicit typing
dynamic typing

Tuesday, April 27, 2010

12

Examples

- ◁ (Static) explicit typing: Java, C, C++, C#
 - the programmer needs to learn to use types
- ◁ (Static) implicit typing: ML, Haskell
 - difficult to implement in some case
- ◁ Dynamic typing: Ruby, Python, JS
 - easy to use and moderately easy to implement
 - still results in run time errors...
 - ...but at least these are detected and reported

Tuesday, April 27, 2010

13

Weak and strong typing

- ◁ Untyped languages: most assembly languages
- ◁ Weakly-typed languages - only a part of the code is type-checked: e.g. C (void* for functions, typecasts), C++
- ◁ Strongly-typed languages - no deviations from types are allowed
 - ◁ Automatic type coercion: if there is a (possibly built-in) function from type A to B, it can be inserted automatically if required by the context. Typical examples: int↔float, subtype→supertype
 - ◁ In Java, types are always checked - but some are checked at run time instead of at compile time

```
class A extends B {...}
int fcn(B b) { A a = (A)b; /* downcast*/ ... }
```

Tuesday, April 27, 2010

14

Timber's type system

- ◁ Timber is a strongly-typed language with static type-checking (all code is checked at compile time)
- ◁ Timber has an advanced type system, but the difficulty for a programmer is mitigated by type inference mechanism (types of variables and expressions are inferred from how they are used)
 - ◁ Explicit type signatures can also be given

Tuesday, April 27, 2010

15

Types in Timber

- ◁ Primitive types:
 - Int (literals: 1, -5)
 - Float (literals: 3.5, 4)
 - Bool (literals: True, False)
 - Char (literals: 'a')
 - () (pronounced "unit")
- ◁ String is a list of Chars (literals: "Hello!")
`type String = [Char]`
- ◁ Function types:
 - Int -> Int {- takes one argument of type Int and returns an Int -}
 - Int -> Int -> String {- takes two arguments of type Int and returns a String -}

Tuesday, April 27, 2010

16

Types in Timber

- ◀ User-defined data types need to be explicitly declared:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

```
data Temp = Fahrenheit Int | Celsius Int
```

```
data Either a b = Left a | Right b
```

```
data Maybe a = Just a | Nothing
```

Here `Day`, `Temp`, `Either`, `Maybe` are types or type constructors, and `Mon`, `Tue`, etc. are data constructors = values (literals) of corresponding type

Note that data constructors can take arguments (e.g. `Celsius 3`)

- ◀ Data types can be parametrized, and such type constructors can be automatically instantiated with any specific type:

`Maybe` is a type constructor, `Maybe Int` is a concrete type

Types in Timber

- ◀ User-defined struct types need to be explicitly declared:

```
struct Point where
```

```
  x :: Int
```

```
  y :: Int
```

- ◀ Some struct types are parametrized, and such type constructors can be automatically instantiated with any specific type:

```
struct MyPoint a where
```

```
  x :: a
```

```
  y :: a
```

`MyPoint` is a type constructor

`MyPoint Int` and `MyPoint Float` are two concrete types

Types in Timber

- ◀ Class expressions have type `Class a`, where `a` is the type of the interface
- ◀ Action expressions have type `Action`
- ◀ Request expressions have type `Request a`, where `a` is the type of the returned value
- ◀ Procedure expressions have type `Cmd s a`, where `s` is the type of object state and `a` is the type of the returned value. Note that procedures defined outside classes get type `Cmd () a`, and if we need to give a type signature for a procedure defined inside a class, we need to write `Cmd _ a`, with wildcard type for the object's state so that the compiler finds the correct type on itself.

Types in Timber

- ◀ Note that classes, methods and procedures that take arguments are in fact functions that return a value of type `Class a / Action / Request a / Cmd s a`

For example:

```
counter :: Int -> Class Counter
```

```
counter initialValue = class
```

```
  s := initialValue
```

```
  inc :: Int -> Action
```

```
  inc a = action
```

```
    s := s + a
```

```
  result Counter{..}
```

Type signatures in Timber

- Any (bound) name or state variable can be given an explicit type in Timber. The type signature must immediately precede the binding or state variable declaration:

```
f :: Int -> String  
f a = ...
```

This will be accepted by the type checker, but the term will still be checked for type consistency

- Any subexpression can also have its type specified:
`x + ((f y) :: Int) -17`

Recursive types

- A list is an example of a recursive type:
`List a = Nil | Cons a List`

- We can create a recursive data structure using a recursive type:
`data R = A R | B`
`a = A b`
`b = A a`
Note that in this example we are using Timber's support for recursive bindings here