CSc 520

Principles of Programming Languages

13 : Types — Introduction

Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

Why types?

- <mark>Types save typing</mark>.

- What does `a+b` mean?

- In Java it could be

  1. $a +_{int} b$.
  2. $a +_{float} b$.
  3. $a\ concat_{string}\ b$.
  4. `int2float(a)` $+_{float} b$.
  5. $a +_{float}$ `int2float(b)`.
  6. `int2string(a)` $concat_{string}\ b$.

  etc, all depending on the types of `a` and `b`.

Why types...?

- In Icon variables are not given explicit types. Instead, operations carry the types:

  1. `a|b` means binary or on integers.
  2. `a||b` means string concatenation.
  3. `a|||b` means list concatenation.

  Icon has **lots** of operators...

- In other words, without types, we would have to be much more explicit about which operations are performed where.

Why types…?

- Icon programs become a bit wordier since every operator effectively encode the required type of the operands.

- On the other hand, it also becomes more readable since we can see directly from the operator what operation will be performed.

```
global x,y,z
procedure p()
    x := x + y      # integer addition
    x := x || y     # string concatenation
    x := x ||| y    # list concatenation
end
```

Why types…?

- To figure out which operation is performed in a Java program, we have to find the declarations of all variables to find their declared type:

```
int x;
String y;
float z;
void p() {
    x = x + 5;        /* integer addition */
    z = z + 5.0;      /* float addition */
    y = y + "X";      /* string concatenation */
}
```

Why types…?

- **Types prevent errors**.
    - Types save the programmer from himself.
    - Types prevent us from adding a character and a record.

```
int A[20];
float x;
void p() {
    A[5] = x;
    A[x] = 5;
    x = x + A;
}
```

[6]

Why types…?

- Types permit optimization. A compiler can generate better code for `a+b` if it knows that both variables must be integers, than if the exact types aren't known until runtime:

```
global a,b
procedure p() {
    a = new array [20]
    ...
    b = new array [20]
    ...
    a = a + b    /* what operation is performe
}
```

Type Systems

- A <mark>type system</mark> consists of

  - a mechanism for <mark>defining types</mark>,

  - rules for <mark>type equivalence</mark>,

  - rules for <mark>type compatibility</mark>,

  - rules for <mark>type inference</mark>.

Type Systems...

- ==Type equivalence== determines when the types of two values are the same:

```
TYPE A = ARRAY [0..10] OF CHAR;
TYPE B = ARRAY [0..10] OF CHAR;
VAR a : A;
VAR b : B;
BEGIN
    a := b; (* legal? *)
END
```

- Are the types of a and b the same?

Type Systems...

- Type compatibility determines when a value of a given type can be used in a given context:

```
VAR a : float;
VAR b : int;
BEGIN
    a := a + b;
END
```

- Can you add an `int` and a `float`?

Type Systems...

- **Type inference** defines the type of an expression based on its parts and surrounding context:

```
global a,b,c
procedure p(x)
    if x = 5 then
        a := x
    else
        a := "hello"
    write(a)
end
procedure main()
    p(5)
end
```

- What type of data can be written here?

Type Checking

- **Type checking** ensures that a program obeys a language's type rules.

- A **type clash** is a violation of the typing rules.

```
class C {
    void p() {
        int x = new C();
    }
}
```

Type Checking — Strong Typing

- Language L is <mark>strongly typed</mark> if
    - $\oplus$ is an operator in L that expects an object of type T,
    - L prohibits $\oplus$ from accepting objects of any other type,
    - and L requires an implementation (a compiler, interpreter, etc) to enforce this prohibition.

- In other words, a strongly typed language does not allow us to perform operations on the "wrong" type of data.

Type Checking — Weak Typing

- In a <mark>weakly typed</mark> language there are ways to "escape" the type system.

- In C, for example, it is possible to cast a pointer to a `float`, add 3.14 to it, and cast it back to a pointer:

```
int main() {
    int* p = (int*) malloc (sizeof(int));
    float f = *((float*) &p) + 3.14;
    p = (int*)(*(int *)&f);
}
```

- Such operations are probably meaningless and a strongly typed language would prohibit them.

Type Checking — Static/Dynamic Typing

- A language <mark>statically typed</mark> if type checking is done at compile-time.

- A language <mark>dynamically typed</mark> if type checking is done at run-time.

- In practice, even languages which are considered statically typed do some checking at run-time.

- Languages can usually be classified as <mark>mostly strongly typed</mark>, <mark>mostly statically typed</mark>, etc.

Terminology

- Benjamin C. Pierce has said:

  > I spent a few weeks . . . trying to sort out the terminology of *strongly typed, statically typed, safe,* etc., and found it amazingly difficult. . . . The usage of these terms is so various as to render them almost useless.

- It is possible to say

  > My language is more strongly typed than your language.

  but harder to argue that

  > My language is strongly typed/statically typed, etc.

Examples — Pascal

- Pascal is mostly strongly and statically typed.

- <mark>Untagged variant records</mark> are a loophole. They allow us to turn a value of one type into an object of some unrelated type.

- Unlike C, array bounds are checked.

Pascal – Untagged Variant Records

```pascal
type rec = record
              a :  integer;
              case boolean of
                  true :  (x :  integer);
                  false :  (y :  char);
          end;

var r:  rec;
begin
   r.x := 55; r.y := 'A'; write(r.x);
end.
```

- This construct is used to bypass Pascal's strong typing.

Examples — C

- C is weakly and statically typed.

- Pointers can be cast willy-nilly which makes it easy to bypass the type system.

- Array references are not checked:

```
int main() {
    int A[20];
    int B[20];
    A[25] = 5;
}
```

Negative indices were used in the old days to overwrite the operating systems.

- Today, buffer overflows are how most viruses compromise security.

Examples — Ada

- Ada is strongly and mostly statically typed.
- Unlike Pascal, variant records must be tagged:

```
type Device is (Printer, Disk, Drum);
type Peripheral(Unit : Device := Disk) is recor
    case Unit is
        when Printer => Line_Count : Integer ;
        when others =>  Cylinder    : CIndex;
    end case;
    end record;
```

Examples — Ada…

- It is, however, possible to do <mark>non-converting casts</mark> (similar to C), but in a very explicit way:

```
function float2int is
    new unchecked_conversion(float,integer);
...
f := float2int(i);
```

- Some errors can't be checked at compile-time:

```
I, J : Integer range 1 .. 10 := 5;
K    : Integer range 1 .. 20 := 15;
I := J;   --  identical ranges
K := J;   --  compatible ranges
J := K;   --  will raise an exception if K>10
```

Examples — Scheme

- Scheme is completely dynamically typed, so programmers often insert extra checks:
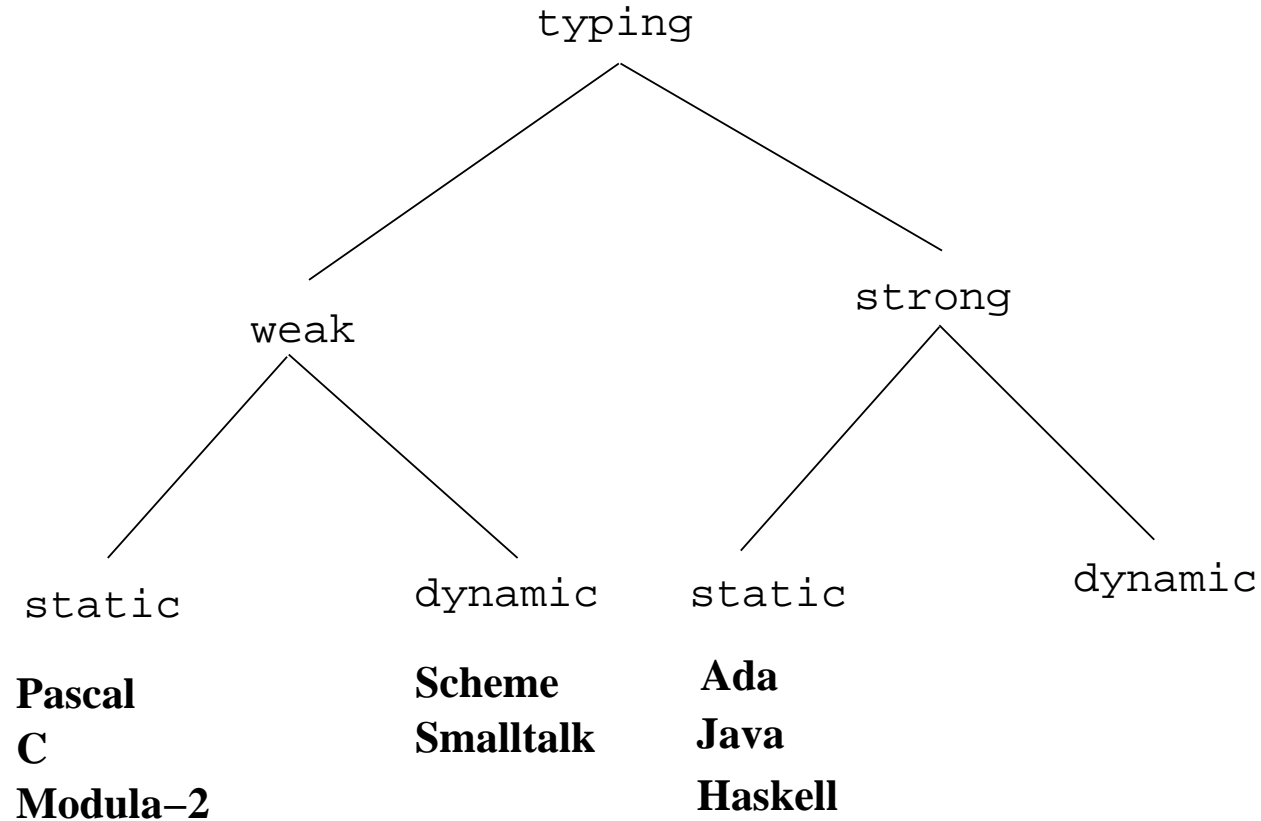
```scheme
(define (sum l)
   (cond
      ((null? l) 0)
      ((not (list? l))
            (error "list expected"))
      ((not (number? (car l)))
            (error "list of numbers expected"))
      (else (+ (car l) (sum (cdr l)))))
))
```

Examples — Java

- Java is strongly and mostly statically typed.

- An exception is thrown here because an `A`-object can't be cast to a `B`-object:

```
class A {}
class B extends A {
    int x;
}
void p() {
    B b = (B) new A();
}
```

Typing


```
                        typing
                       /      \
                      /        \
                   weak       strong
                   /  \        /    \
                  /    \      /      \
              static dynamic static  dynamic

              Pascal  Scheme    Ada
              C       Smalltalk Java
              Modula-2          Haskell
```

Type Inference

- In statically typed languages types are inferred in the compiler, before the program is run:

```
procedure p (x : integer);
var z : real;
var c : char;
begin
    write(x + z);  /* convert x to real,
                      write a real */
    write(c + z);  /* type error */
end
```

Type Inference...

- Haskell and similar languages don't require the programmer to give types to variables and functions.

- Instead, the compiler infers types.

- Given

```
len [] = 0
len _:xs = 1 + len xs
```

the Haskell translator will infer a <mark>most general type</mark>:

```
len :: [a] -> Int
```

- Haskell is strongly and statically typed, although the programmer rarely have to provide explicit type information.

So, What is a Type?

- There are three ways to think about types:
  1. <mark>denotational view</mark> — a type is a set of values;
  2. <mark>constructive view</mark> — a type is what we can construct from the type constructors in the language;
  3. <mark>abstraction-based view</mark> — a type denotes a data object and a well-defined set of allowable operators on this object.
- At different times, we may look at a type in any of these ways.

Denotational View

- A type T is a set of values $\{t_0, t_1, t_2 \ldots\}$.

- A value v is of type T if it belongs to the set.

- A variable v is of type T if it is guaranteed to always hold a value in the set.

- A `char` type in Pascal is the set of 128 seven-bit ASCII characters:

```
{...,
  "0",...,"9",...,
  "A",...,"Z",...,
  "a"...,"z",...}
```

Constructive View

- A Pascal type is (roughly)

  type ::=

      <u>integer</u> | <u>real</u> | <u>char</u> | <u>boolean</u> ...

      <u>[</u> expr <u>..</u> expr <u>]</u> |

      <u>SET OF</u> type |

      <u>ARRAY</u> type <u>OF</u> type |

      <u>RECORD</u> [field_list] <u>END</u>

- I.e., a Pascal type is either one of the built-in types, or ones we define ourselves by composing ==type constructors==, such as `ARRAY`, `RECORD`, etc:

```
END T = RECORD
    a : real;
    b : ARRAY ["a".."z"] OF SET OF char;
END;
```

Abstraction-Based View

- A type is an <mark>abstract data type</mark>.

- The next slides shows what the Modula-3 language manual says about the operations that are allowed on `Words`.

- The allowed operations include arithmetic and logical operations.

- There is no "pointer dereferencing" operation defined, however, so apparently this operation is not allowed.

Abstraction-Based View…

```
INTERFACE Word;
   TYPE T = INTEGER;
   PROCEDURE Plus  (x,y: T): T;
   PROCEDURE Times (x,y: T): T;
   PROCEDURE Minus (x,y: T): T;
   PROCEDURE Divide(x,y: T): T;
   PROCEDURE Mod(x,y: T): T;
   PROCEDURE LT(x,y: T): BOOLEAN;
   PROCEDURE LE(x,y: T): BOOLEAN;
   PROCEDURE GT(x,y: T): BOOLEAN;
   PROCEDURE GE(x,y: T): BOOLEAN;
   PROCEDURE And(x,y: T): T;
   PROCEDURE Or (x,y: T): T;
   PROCEDURE Xor(x,y: T): T;
   PROCEDURE Not  (x: T):  T;
   PROCEDURE Shift(x: T; n: INTEGER): T;
   PROCEDURE Rotate(x: T; n: INTEGER): T;
   PROCEDURE Extract(x: T; i, n: CARDINAL): T;
   PROCEDURE Insert(x: T; y: T; i, n: CARDINAL): T;
END Word.
```

[31]

Readings and References

- Read Scott, pp.307–312.