Types

Amer Diwan

Some important dimensions in types

- When are types checked?
- How "completely" are types checked?
- When is it legal to do an assignment?
  - Subtyping is a clean mechanism for expressing this!
- When are types equal?

When to do "type checking"

- Static type checking: all checking at compile/link time
- Dynamic type checking: all checking at run time
- Most languages fall somewhere in between e.g., Java and Modula-3

Examples of type checking

- int i;
  int j; — Static
  i = j;
- char *p;
  void *q; — Dynamic?
  p = q;
- int i;
  char c; — Dynamic?
  c = i;

How strong is the checking

- Strongly-typed: All expressions are checked (and guaranteed) to be type-consistent at compile or run time (type-safe).
  - Pascal, Modula-2, Modula-3, Java, ...
- Weakly-typed: Some expressions are not completely typechecked and unchecked type violations may happen at run time
  - C, C++, assembler

Strong and weak typing example

Strong typing

Weak typing

- char *p;
  void *q;
  check istype(q, char*)
  Good luck!
  p = q;
- int i;
  char c;
  check istype(i,char)
  Good luck!
  c = i;

Discussion: static versus dynamic typing

- Advantages of static
  - better performance
  - find errors early
  - more complete
- Disadvantages of static
  - less flexible (particularly for prototyping)
  - slower compilation

Discussion: strong versus weak typing

- Advantages of strong
  - more bugs found
  - may be faster
  - useful documentation in conjunction with static typing
- Disadvantages of strong
  - need to insert casts and checks for code to compile/run
  - device drivers
  - may be slower

What is legal?

- When is x:T = y:U legal?
- When is x:T op y:U legal?
- Two important concepts:
  - type equality
  - subtyping

Type equality

- When are two types, T1 and T2, equal?
  - Name equivalence: when T1 and T2 have the same name. *Anonymous* types have unique “names”.
  - Structural equivalence: when T1 and T2 have the same structure.
- Structural equivalence: Modula-3, Algol
  Name equivalence: Modula-2, C, Java, C++, Ada
  Undefined: Pascal

Type equality examples

```
TYPE T1 = RECORD i: INTEGER; b: BOOLEAN; END;
TYPE T2 = RECORD i: INTEGER; b: BOOLEAN; END;
TYPE T3 = T2;
```

- T1 = T2? T1 = T3? T2 = T3?

A simple algorithm for structural equivalence

- T1 = T2 =>
  - Replace all names in T1 and T2 with their expansion until T1 and T2 do not contain any type names
  - T1 = T2 if their expansions are identical

Structural equivalence example

TYPE T1 = RECORD i: INTEGER; b: T3; END;
TYPE T2 = RECORD i: INTEGER; b: T4; END;
TYPE T3 = RECORD x: BOOLEAN; END;
TYPE T4 = RECORD x: BOOLEAN; END;

T1 ?= T2 expands into:

RECORD i: INTEGER; b: T3; END; ?=
RECORD i: INTEGER; b: T4; END;

Which further expands into

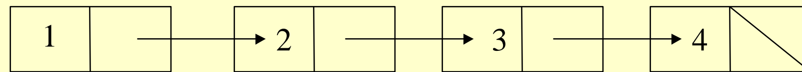
RECORD i: INTEGER; b: RECORD x: BOOLEAN; END; END; ?=
RECORD i: INTEGER; b: RECORD x: BOOLEAN; END; END;

which are identical

Structural equivalence--it ain't easy to do

TYPE T1 = RECORD	TYPE T2 = RECORD
value: INTEGER;	value: INTEGER;
next: REF T1;	next: REF T2;
END	END

The expansion is infinite for these types



Structural equivalence: new algorithm (simplified)

T1 = T2

- FALSE if T1.kind != T2.kind, else
- TRUE if T1 and T2 are identical or in *assume-equal*, else
- add (T1, T2) to *assume-equal*(T1, T2) and do a component-wise equality test on T1 and T2. TRUE if all components have equal types

Example

```
TYPE T1 = RECORD        TYPE T2 = RECORD
  value: INTEGER;          value: INTEGER;
  next: REF T1;            next: REF T2;
END                     END
```

assume-equal += (T1, T2)
compare(value: INTEGER, value: INTEGER)? returns TRUE
compare(next: REF T1, next: REF T2)?
assume-equal += (REF T1, REF T2)
compare(T1, T2)? returns TRUE since (T1,T2) in assume-equal

Advantages and disadvantages of name equivalence

- Advantage
  - Types with different names are treated differently
    - age=[0..100] ≠ temperature=[0..100]
- Disadvantages
  - Not systematic:
    - Are x: int; y: int; of the same type?
    - Are x: struct i: integer end; y: struct i: integer end; of the same type?
    - T = struct i: integer end; U=T; Are x: T; y: U of the same type?

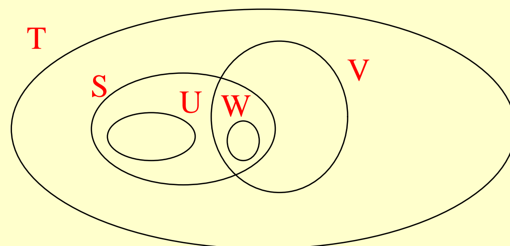
Type equivalence and distributed environments

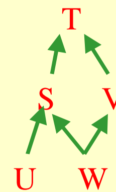
```
Program Producer()
a: ARRAY [1..1024] OF INTEGER
send(Consumer, a)
```

```
Program Consumer()
a: ARRAY [1..1024] OF INTEGER
receive(a)
```

Subtyping

- Type *S* is a subtype of type *T* if every value of type *S* is also a value of type *T*
- Written as S <: T





Ways of thinking about subtyping

- Subtype may be used whenever the supertype is expected
  - e.g., if x = i:int is legal then so is x = i:[10..20]
- Subtype has more stringent membership requirements than supertype
  - e.g., int versus [10..20]
- Subtype has fewer members than supertype
- widen: subtype -> supertype
  narrow: supertype ->subtype

More examples of subtyping

Assignability

When is b: B := a: A legal?

- Answer 1: A <: B   Too restrictive?
- Answer 2: A <: B or B <: A with some run-time type checking or conversions

```
VAR j: INTEGER; i: [10..20]        if j > 20 or j < 10 report_error
i := j                       →     else i := cast(j)
```

When is type A<: type B

- Trivially if A = B
- Transitively if A <: C and C <: B
- Subtyping between integers and subranges is easy: directly apply value inclusion
- How about sets?
  A = SET of {red, blue}
  B = SET of {red, blue, yellow}

Yes by value inclusion, but real languages often disallow this!

When is type A<: type B (cont.)

- Arrays
  A = ARRAY[1..10] OF INTEGER
  B = ARRAY[11..20] OF INTEGER
  - A <: B if arrays are sequences of values, and they have the same element type and same length
  - What if the arrays have different lengths?
  - What if the arrays have different element types?
  - Performance considerations creep in!
- Objects: by inheritance

Summary

- Many dimensions to a type system
  - Static or dynamic
  - Strong or weak
  - Structural or name equivalence
  - Type-checking rules
- Choices in the type system can affect performance

Next topic: Types in languages

- How are types incorporated into some common languages
- Reading:
  - *The Modula-3 type system*