Types in languages:
Modula-3

Amer Diwan

Why bother with Modula-3?

• A small and clean, yet usable object-oriented language
• Language design goal: the entire language definition should fit in 50 pages
• Buzzword compliant--statically and strongly typed, objects, exceptions, threads, garbage collection, modules, generics, ...
Themes in Modula-3

- Increased robustness through safety from unchecked run-time errors
- Systematic—few exceptions
  - But not at the expense of performance

Outline

- Modula-3’s notion of types
  - Type equality in Modula-3
  - Subtyping rules
  - Assignment rules (based on subtyping rules)
  - Information hiding in Modula-3
Type equality

- Modula-3 uses structural equivalence
- But structural equality can be “over-ridden” if needed
  - e.g.,
    
    BRANDED “one” OBJECT i: INTEGER; END;
    ≠
    BRANDED “two” OBJECT i: INTEGER; END;

Subtyping of pointer types

- Pointers may be traced or un traced
- Traced references are examined by the garbage collector.
  
  NULL <: REF T <: REFANY
- Un traced references are managed by the user
  
  NULL <: UNTRACED REF T <: ADDRESS
- Traced and un traced references are unrelated and may not be assigned to each other
Why have untraced references?

- Low-level code
- Performance

Subtyping of fixed arrays

- ARRAY I OF T <: ARRAY J OF T
  if NUMBER(I) = NUMBER(j)

  A1 = ARRAY[0..100] OF INTEGER
  A2 = ARRAY[100..200] OF INTEGER
  A3 = ARRAY[0..100] OF [0..255]

- Is A2 <: A1?
- Is A3 <: A1?
Subtyping of object types

• May be abstract
• NULL <: T OBJECT ... END <: T OBJECT ... END <: REFANY UNTRACED OBJECT ... END <: ADDRESS
• No explicit syntax for private, public, friend, ...
  – Uses opaque types

Examples

• T1 = BRANDED “one” OBJECT i: INTEGER; END;
  T2 = BRANDED “two” OBJECT i: INTEGER; END;
  ST1 = T1 OBJECT j: INTEGER; END;
  ST2 = T2 OBJECT j: INTEGER; END;
• T1 <: T2?
• ST1 <: T1?
• ST2 <: T2?
• ST1 <: T2?
Assignment rules

• Type $T$ is assignable to Type $U$ if
  – $T <: U$
  – $T$ and $U$ are ordinal types with at least one member in common
  – $U <: T$ and $T$ is an array type or reference type but not an ADDRESS type
• Why the exception in the third case?
• Note the implicit safe casts!

Opaque types

• The information hiding mechanism based on subtyping
• $\text{TYPE } T <: U$
  $U = \text{OBJECT } i: \text{INTEGER}; \text{END;}$
  $T$, an opaque type, is some subtype of $U$
• $\text{REVEAL } T = U \text{OBJECT } j: \text{INTEGER}; \text{END;}$
  $T$ is “revealed”: must be consistent with its opaque declaration
Revelations

- Revelations can be incremental
  \[
  \text{TYPE } T <: U \\
  U = \text{OBJECT } i: \text{INTEGER}; \text{END}; \\
  V = U \text{OBJECT } ch: \text{CHAR}; \text{END};
  \]
- REVEAL \( T <: V \);
- REVEAL \( T = V \text{OBJECT } j: \text{INTEGER}; \text{END}; \)
- Can reveal different views to different clients (trusted, etc.).

An example of using opaque types

- INTERFACE Counter;
  TYPE \( T <: \text{Public}; \)
  Public = \text{OBJECT} \text{METHODS} next(): \text{INTEGER}; \text{END};
  END Counter
- INTERFACE CounterFriends IMPORT Counter;
  REVEAL Counter.T <: U;
  TYPE \( U = \text{Counter.Public} \text{OBJECT } \text{last}_\text{value}: \text{INTEGER}; \text{END}; \)
  END CounterFriends
- MODULE Counter EXPORTS Counter, CounterFriends;
  REVEAL \( T = U \text{OBJECT } \text{otherstate}: \text{INTEGER}; \text{END}; \)
  END Counter.
Continuing with example

- MODULE TrustedClient; IMPORT Counter, CounterFriends;
  BEGIN
  END TrustedClient
- MODULE OtherClient; IMPORT Counter;
  BEGIN
  END OtherClient

Pros and cons of Modula-3’s mechanism

- Adv
  – clean from user perspective: need to know
  – selective revelation
  – fine control over revelation esp if have different levels of “trust”
- Disadv
  – different from Java
  – creates lots of types spread around different files
Unsafe parts of Modula-3

- Unsafe operations are restricted to modules especially marked as unsafe
  - Explicit deallocation: Untraced references may be deallocated only unsafe modules
  - Unchecked type casts: Called LOOPHOLE!
  - ...

Implications for implementation of types in Modula-3

- Need type descriptors at run time
  - Must support subtype tests
  - Must support equality tests
  - Must support garbage collection
  - Must support “size” queries
  - (Must support method dispatch)
An example run-time type structure

Instance variables for object

what do we need here?

Instance variables for object

Type equality and subtype tests for object

- Assign a number to each type in pre-order traversal
  Store number of highest-numbered subtype with each node
Type equality and subtype tests (cont.)

- \( T <: U \) if \( U.\text{id} \leq T.\text{id} \leq U.\text{maxid} \)
- \( T = U \) if \( T.\text{id} = U.\text{id} \)

Subtype and equality tests are relatively fast especially with linker tricks

- But at what cost?
- Does this trick work for multiple inheritance?
Supporting garbage collection

- The type descriptor must contain the size of the object and the types of its components (at least a pointer/non-pointer bit)
- Need additional support for non-objects (will be discussed later)
- What’s the complexity here?

Summary

- Modula-3 has a largely uniform type system
  - Tries to be clean and consistent most of the time
  - Occasionally relaxes “clean” for speed
- Uses an elegant mechanism for information hiding, but
  - Adds complexity to compilation/linking
  - Ties together a implementation reuse mechanism with information hiding: not always the right granularity
Next lecture: Types in Java

• How are the important type concepts implemented in Java and what are their implications?
• Reading: Java language definition chapters 4 and 5 (links on class web page)