Types

Amer Diwan

Primitive types

- Eg., int, char, string, ...
- Introduce four useful properties:
  - Invisibility of underlying representation
  - Correct use of variables can be checked
  - Disambiguation of operators can be done at compile time
  - Accuracy control: yields space optimization and gives more information to compiler about legal values

Aggregate types

- Further improve readability by allowing user to define specialized types
  - Cartesian product
  - Finite mapping
  - Sequencing
  - Recursion
  - Discriminated union
  - Powerset
  - structs, objects, datatypes
  - Pascal and C arrays
  - Strings, arrays in Modula-3
  - Linked lists
  - Pascal variant records
  - Pascal and Modula-3 sets

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. Java and Modula-3 do static type checking most of the time, but relaxes it in some cases

How strong is the checking

- Strongly-typed: All expressions are checked (and guaranteed) to be type-consistent at compile or run time (type-safe)
- Weakly-typed: Some expressions are not completely typechecked and type violations may happen at run time

Type checking discussion

- Which is “better”: static or dynamic typing?
- Which is “better”: strong or weak typing?
- Should all languages be strongly typed?
Operations allowed by the type system?

- When does the type system allow
  - assignments
  - passing parameters
  - ...

- The concept of subtyping is useful in talking about legal and illegal operations

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
- Subtype has more stringent membership requirements than supertype
- Subtype has fewer members than supertype
- Eg: $[10..20] <: \text{int}$
- class $T$: public $S [...] <: S$
- widen: subtype $\rightarrow$ supertype
- narrow: supertype $\rightarrow$ subtype

Uses of subtyping

- Code reuse
- Polymorphism
- Hiding information in a “typeful” manner

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

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?
  - $A = \text{SET of \{red, blue\}}$
  - $B = \text{SET of \{red, blue, yellow\}}$
- Yes by value inclusion, but real languages often disallow this!
When is type A <: type B (cont.)

- **Arrays**
  - \( A = \text{ARRAY}[1..10] \text{ OF INTEGER} \)
  - \( B = \text{ARRAY}[11..20] \text{ OF INTEGER} \)
  - \( A <: B \) if arrays are sequences of values, and they have the same element type and same length
- **Objects:** by inheritance

Type equality

- When are two types, \( T_1 \) and \( T_2 \), equal?
  - **Name equivalence:** when \( T_1 \) and \( T_2 \) were defined in the same place
  - **Structural equivalence:** when \( T_1 \) and \( T_2 \) have identical structure.
- Modula-3 and Algol have structural equivalence. C, Java, C++, and Add have name equivalence. Pascal leaves it undefined: up to the implementation

Type equality examples

\[
\begin{align*}
T_1 & = \text{RECORD} i: \text{INTEGER}; b: \text{BOOLEAN}; \text{END}; \\
T_2 & = \text{RECORD} i: \text{INTEGER}; b: \text{BOOLEAN}; \text{END}; \\
T_3 & = \text{T}_2; \\
T_1 & = T_2? \ T_1 = T_3? \ T_2 = T_3?
\end{align*}
\]

A simple algorithm for structural equivalence

\[
\begin{align*}
T_1 = T_2 & \Rightarrow \\
& \text{Replace all names in } T_1 \text{ and } T_2 \text{ with their expansion until } T_1 \text{ and } T_2 \text{ do not contain any type names} \\
& \text{ if their expansions are identical}
\end{align*}
\]

Structural equivalence example

\[
\begin{align*}
T_1 & = \text{RECORD} i: \text{INTEGER}; b: \text{T}_3; \text{END}; \\
T_2 & = \text{RECORD} i: \text{INTEGER}; b: \text{T}_4; \text{END}; \\
T_3 & = \text{RECORD} x: \text{BOOLEAN}; \text{END}; \\
T_4 & = \text{RECORD} x: \text{BOOLEAN}; \text{END}; \\
\text{T}_1 & ?= \text{T}_2 \text{ expands into:} \\
& \text{RECORD} i: \text{INTEGER}; b: \text{T}_3; \text{END}; ?= \\
& \text{RECORD} i: \text{INTEGER}; b: \text{T}_4; \text{END}; \\
\text{Which further expands into} \\
& \text{RECORD} i: \text{INTEGER}; b: \text{RECORD} x: \text{BOOLEAN}; \text{END}; ?= \\
& \text{RECORD} i: \text{INTEGER}; b: \text{RECORD} x: \text{BOOLEAN}; \text{END}; \\
& \text{which are identical}
\end{align*}
\]

Structural equivalence--it ain’t easy to do

\[
\begin{align*}
\text{TYPE } T_1 & = \text{RECORD} \text{ value: INTEGER; next: REF } T_1; \text{END} \\
\text{TYPE } T_2 & = \text{RECORD} \text{ value: INTEGER; next: REF } T_2; \text{END} \\
\text{The expansion is infinite for these types}
\end{align*}
\]
Structural equivalence: new algorithm

\[
T1 = T2
\]
- \text{FALSE} if \(T1.\text{kind} \neq T2.\text{kind}\), else
- \text{TRUE} if \(T1\) and \(T2\) are identical or in \text{assume-equal}, else
- add \((T1, T2)\) to \text{assume-equal}(T1, T2) and
do a component-wise equality test on \(T1\) and \(T2\).
\text{TRUE} if all components have equal types

Advantages and disadvantages of name equivalence

- **Advantages**
- **Disadvantages**

Type equivalence and distributed environments

Program Producer()
\[ a : \text{ARRAY}[1..1024] \text{OF INTEGER} \]
send(Consumer, a)

Program Consumer()
\[ a : \text{ARRAY}[1..1024] \text{OF INTEGER} \]
receive(a)

Summary

- **Basic concepts of a type system**
  - primitive and aggregate types
  - equality, subtyping, and assignability rules
- **Many dimensions to a type system**
  - Static or dynamic
  - Strong or weak
  - Structural or name equivalence

Next topic: Types in languages

- How are types incorporated into some common languages
- **Reading:**
  - “The Modula-3 type system” (handout)
  - Types in Java (web page)
  - Oberon (web page)