Programming Languages

Types

Benjamin J. Keller

Department of Computer Science, Virginia Tech

Types Overview

- Type Systems
- Built-in types
- Aggregate types
- User-defined types
- Static and Dynamic typing

Type Systems

- Mechanism for defining types, and
- Set of rules for
  - type equivalence — when types are the same
  - type compatibility — when value of a type can be used
  - type inference — what type an expression has

Type Checking

- Test that program obeys type compatibility rules
- *Type Class* — violation of type rules
- *Strongly typed language* — prohibits application of an operator to an operand of wrong type.
- *Statically typed language* — strongly typed language for which type checking can be done at compile time.

Models of Types

- *Denotational* — a type is a set
- *Constructive* — a type is one of primitive types, or composite type constructed from other types
- *Abstraction-based* — a type is an interface consisting of operations with well-defined and consistent semantics

Built-In Types

- Primitive types
  1. Hide representation of data
  2. Allow type-checking at compile and/or run-time
  3. Help disambiguate operators
  4. Allow expression of constraints on accuracy of representation
     - (COBOL, PL/I, Ada) LongInt, DoublePrecision, etc.
     - Save space and check for legal values
- Aggregate types
- Come with built-in operations

Cartesian Products

- Product of types

$$S \times T = \{\langle s, t \rangle \,|\, s \in S, t \in T\}\,.$$

- Can also write as $\Pi_{i \in I} S_i = S_1 \times S_2 \times \ldots \times S_n$.
- If all types are the same, write as S^n.
- Ex. Tuples of ML: `type point = int * int`
- How many elements in product?
- S^0 called unit in ML.

Records

- Records in COBOL, Pascal, Ada
- Structures in PL/I, C, and Algol 68
- Heterogeneous collections of data
- Fields are labeled (different than product type)

```
record                            record
  x : integer;                      a : integer;
  y : real                          b : real;
end;                              end;
```

- Operations and relations: selection ".", assignment, equality
- Can use generalized product notation: $\Pi_{l \in \mathrm{Label}} T(l)$
- Ex. $\mathrm{Label} = \{x, y\}$, $T(x) = \mathrm{integer}$, $T(y) = \mathrm{real}$.

Disjoint Union

- Variant record — $T_1 \cup T_2$ with discriminant
- Support alternatives within type:

```
RECORD
  name : string;
  CASE status : (student, faculty) OF
    student: gpa : real;
             class : INTEGER;
    | faculty: rank : (Assis, Assoc, Prof);
  END;
END;
```

- Goal: save space yet provide type security.
- Space reserved for a variable of this type is the larger of the variants.

Type Security of Disjoint Unions

- Type security fails in Pascal and MODULA-2 since variants not protected
- Allow changing discriminant without changing corresponding data.
- Examples of type safe disjoint unions in Ada, Clu, ML
- In ML can create a disjoint union as (type safe)

```
datatype IntReal = INTEGER of int | REAL of real;
```

Ada Variants

- Declared as parameterized records:

```
type geometric (Kind: (Triangle, Square) := Square) is
        record
           color : ColorType := Red ;
           case Kind of
              when Triangle =>
                     pt1,pt2,pt3:Point;
              when Square =>
                     upperleft : Point;
                     length : INTEGER range 1..100;
           end case;
        end record;
```

Ada Variants (cont)

- Declarations
  - `ob1: geometric` — sets `Kind` as default `Square`
  - `ob2: geometric(Triangle)` — sets `Kind` as `Triangle`
- Illegal to change discriminant alone.
  - `ob1 := ob2` — OK
  - `ob2 := ob1` — generate run-time check to ensure Triangle
- If want to change discriminant, must assign values to all components of record:

```
ob1 := (Color=>Red,Kind=>Triangle,
        pt1=>a,pt2=>b,pt3=>c);
```

Ada Variants (cont)

- If write code

  ```
  ... ob1.length...
  ```

  then converted to run-time check:

  ```
  if ob1.Kind = Square then ... ob1.length ....
                       else raise constraint_error
  end if.
  ```

- Fixes type insecurity of Pascal

Disjoint Unions in C

- C supports undiscriminated unions:

  `typedef union {int i; float r;} utype.`

- No static or run-time checking is performed to ensure proper use

Disjoint Unions

- Note disjoint union is not same as set-theoretic union, since have tags.

$$\mathrm{IntReal} = \{\mathrm{INTEGER}\} \times \mathrm{int} + \{\mathrm{REAL}\} \times \mathrm{real}$$

Arrays

- Homogeneous collection of data
- Like function with finite domain (index type) to element type

  `Array [1..10] of Real`

  corresponds to map $\{1, ..., 10\} \rightarrow \text{Real}$
- Operations: indexed access, assignment, equality
- Sometimes a slice operation: `A[2..6]` represents an array composed of `A[2]` to `A[6]`

Array Bindings

- Attributes: index range (size) and location of array
- Static:
  - Index range and location bound at compile time
  - FORTRAN
- Semi-static:
  - Index range of array bound at compile time
  - Location is determined at run-time
  - Pascal — array stored on stack

Array Bindings

- (Semi-)dynamic:
  - Index range may vary at run-time
  - Attributes of a local variable may be determined by procedure parameter
  - Size fixed once procedure invoked
  - ALGOL 60, Ada
- Flexible:
  - Size may change at any time during execution
  - Can extend array size when needed
  - Algol 68 and Clu

Sets

- Collection of elements

```
set of elt_type;
```

- Implemented as bitset or dynamic structure (list)
- Operations: assignment, equality, subset, membership, etc.
- Base type generally needs to be primitive (why?)

Recursive Types

- ML Examples

```
tree = Empty | Mktree of int * tree * tree
list = Nil | Cons of int * list
```

- Supported by some languages: Miranda, Haskell, ML
- But built by programmer in others with pointers

```
list = POINTER TO RECORD
          first:integer;
          rest: list
       END;
```

Recursive Types (cont)

- Think of type as set, and type definition as equation
- Recursive types may have many solutions
- Example: $\text{list} = \{\text{Nil}\} \cup (\text{int} \times \text{list})$ has the solutions
  1. Finite sequences of integers followed by Nil: `(2,(5,Nil))`
  2. Finite or infinite sequences, where if finite then end with Nil
- Theoretical result: Recursive equations always have a least solution — although may give an infinite set if real recursion.

Recursive Types (cont)

- Can find via finite approximation.

$$
\begin{aligned}
\text{list}_0 &= \{\text{Nil}\} \\
\text{list}_1 &= \{\text{Nil}\} \cup (\text{int} \times \text{list}_0) \\
&= \{\text{Nil}\} \cup \{(n, \text{Nil}) | n \in \text{int}\} \\
\text{list}_2 &= \{\text{Nil}\} \cup (\text{int} \times \text{list}_1) \\
&= \{\text{Nil}\} \cup \{(n, \text{Nil}) | n \in \text{int}\} \cup \{(m, (n, \text{Nil})) | m, n \in \text{int}\} \\
&\vdots \\
\text{list} &= \bigcup_n \text{list}_n
\end{aligned}
$$

Recursive Types (cont)

- Construction like unwinding definition of recursive function

$$
\begin{aligned}
\text{fact}_0 &= \text{fun } n \Rightarrow \text{ if } n = 0 \text{ then } 1 \text{ else undef} \\
\text{fact}_1 &= \text{fun } n \Rightarrow \text{ if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}_0(n-1) \\
&= \text{fun } n \Rightarrow \text{ if } n = 0, 1 \text{ then } 1 \text{ else undef} \\
\text{fact}_2 &= \text{fun } n \Rightarrow \text{ if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}_1(n-1) \\
&= \text{fun } n \Rightarrow \text{ if } n = 0, 1 \text{ then } 1 \text{ else} \\
&\quad\ \text{if } n = 2 \text{ then } 2 \text{ else undef} \\
&\ldots \\
\text{fact} &= \bigcup_n \text{fact}_n
\end{aligned}
$$

- Some recursive type equations inconsistent with classical math, but used in computer science

Sequences

- Lists
  - Supported in most functional and logical languages
  - operations: head, tail, cons, length, etc.
- Sequential files
  - Operations: open, close, reset, read, write, check for end.
  - Persistent data — files.
- Strings
  - Operations: comparison, length, substring
  - Either primitive or composite
    - Composite (arrays) in Pascal, Modula-2, ...
    - Primitive in ML
    - Lists in Miranda and Prolog (no length bound)

User-Defined Types

- User gets to name new types.
- Rationale:
  1. more readable
  2. Easy to modify if definition localized
  3. Factorization — avoid work and mistakes of making copies of type expressions
  4. Added consistency checking in many cases.

Static and Dynamic Typing

- Most languages use static binding of types to variables, usually in declaration

```
int x;  //bound at translation time}
```

- FORTRAN has implicit declaration using naming conventions
  If start with "I" to "N", then integer, otherwise real.
- Other languages will infer type of undeclared variables.
- Both run real danger of problems due to typing mistakes

Errors and Typing

- Example in ML, if

  ```
  datatype Stack ::= Nil | Push of int;
  ```

  then define

  ```
  fun f Push 7 = ...
  ```

- What error occurs?
- Answer: Push is taken as a parameter name, not a constructor. Therefore f is given type: `A -> int -> B` rather than the expected: `Stack -> B`

Dynamic Binding

- Dynamic binding found in APL and LISP.
- Type of variable may change during execution.
- Example: One declaration of `x`, and at one point `x = 0` and at another `x = [5,2,3]`
- Can't allocate a fixed amount of space for variables.
- Often implemented as pointer to location of value.
- Determine which version of overloaded operator to use (`+`) when executing.
- Variable must have type tag

Type Equivalence

- When are types identical?

```
Type    T = Array [1..10] of Integer;
Var  A, B : Array [1..10] of Integer;
         C : Array [1..10] of Integer;
         D : T;
         E : T;
```

- Which variables have the same type?
- Name Equivalence
  - Same type name: D and E
  - Same type name or declared together: A and B, D and E
- Structural Equivalence — Same structure means same type (all same)

Structural Equivalence

- Different approaches to equivalence
- Do names matter? Does order matter?

```
T1 = record a : integer; b : real  end;
T2 = record c : integer; d : real  end;
T3 = record b : real; a : integer  end;
```

- Even worse:

```
T = record info : integer; next : ^T  end;
U = record info : integer; next : ^V  end;
V = record info : integer; next : ^U  end;
```

- Different languages make different choices

Problem

- Cannot distinguish

```
type student = record
    name, address : string
    age : integer
```

- and

```
type school = record
    name, address : string
    age : integer
```

- Structural equivalence allows

```
x : student;
y : school;
...
x := y;
```

Name Equivalence

- Name equivalence says types with different names are different
- Assumption: programmer named them that way so they would be different
- Most recent languages use name equivalence (Java for instance)
- Difficulty caused by *alias* types
  - May define data structure parameterized by type

    ```
    type stack_element = integer;
    ```

  - Want `integer` to be same as `stack_element`
  - May want distinct types to prevent mixed computations

    ```
    type celsius = real;
    type fahrenheit = real;
    ```

Name Equivalence

- *Strict name equivalence* — aliases are distinct types
- *Loose name equivalence* — aliases are equivalent types
- Difference

  ```
  type A = B;
  ```

  - is a definition under strict name equivalence
  - is a declaration under loose name equivalence
- Ada allows both

  ```
  subtype stack_element is integer; --- equivalent
  type celsius is new integer;      --- distinct
  type fahrenheit is new integer;   --- distinct
  ```

Type Conversion

- Explicit conversion (cast) of value from one type to another
- Cases:
  1. Types are structurally equivalent — no code generation required
  2. Types have nontrivial overlap of balues represented in the same way — may require check that value is in target type
  3. Types have distinct representations — conversions use special machine instructions (e.g., int to float)

Type Coercion

- Implicit conversion that occurs when operand type must be converted to match type expected by an operator
- Common in some languages (C), not performed in others (ML)
- C++ allows definition of coercion operators for classes
- Weaken type security — allow conversions that may not be desired by programmer

Type Inference

- Determining type of expression from subexpressions
- Mostly obvious

```
int x, y;
x = x + y;
```

- However type may not be *closed* on operations
  - Subranges — addition of values in range `10..20`
  - Composites — concatenation of length 3 character arrays
- Must perform runtime semantic checks

More Sophisticated Type Inference

- Some functional languages use sophisticated form of type inference
- *Type consistency* — type checking algorithm can find a unique type for every expression, with no contradictions and no ambiguous occurrences of overloaded operators

ML Type Consistency Rules

- All occurrences of an identifier must have same type
- In expression `if` b `then` e_1 `else` e_2, b must have type boolean, and e_1 and e_2 must have the same type
- A function has a type of the form `'a -> 'b` where `'a` is the type of the function's parameter, and `'b` is the type of the result
- In a function application, the argument type must be the same as the parameter type, and the result type is the type of the application

Type Unification

- Used to resolve types when must be same by consistency rules
- Similar to unification (matching) in Prolog
- Example: have expression `if` b `then` e_1 `else` e_2
- If know that e_1 has type `'a * int`, and that e_2 has type `string * 'b` then can unify by substituting `string` for `'a`, and `int` for `'b`.

Type Completeness Principle

- No operation should be arbitrarily restricted in the types of the values involved
- Avoid second-class types

  Ex. (Pascal) Restrictions on return values of functions, lack of procedure variables, etc.
- ML comes much closer to satisfying than many other languages

Summary Of Types

- Modern tendency is to strengthen static typing and avoid implicit holes in types system
- Can only explicitly bypass type system
- Make as many errors occur at compile time as possible by:
  - Requiring over-specification through typing
  - Distinguishing between different uses of same types (name equivalence)
  - Mandating constructs designed to eliminate typing holes
  - Minimizing or eliminating use of explicit pointers (especially user-controlled deallocation of pointers)

Summary Of Types (cont)

- Trend results in loss of flexibility provided by dynamic typing or lack of any typing
- Goal of current research: recovering flexibility without losing type safety
- Progress made over last 20 years includes polymorphism, ADT's, subtyping and aspects of object-oriented languages.