Introduction to types
Amer Diwan

Overview

• This topic: what are types, kinds of types, representation of types (1-2 lectures)
• Next topic: type checking, relationship between types, information hiding (2 lectures)
• Topic after next: polymorphism/OO (2 lectures)
Let’s think about the untyped world

• Untyped universe: 1 type
  – e.g., in machine code, everything is a string of bits
  – e.g., in smalltalk, everything is an object

But, ...

• Types often arise naturally...
  – Bit strings in computer memory are organized into integers, characters, instructions, ...
  – Objects in smalltalk are organized into rectangles, dictionaries, ...
    • But organized as sets of pairs, functions, ...
• Even untyped universes of objects decompose naturally into sets with uniform behavior
From untyped to typed worlds

Objects naturally fall into groups but code can violate the groups
A type system is a suit of armor that “protects” the groups

Why have types?

- Correct use of variables can be checked
- Types are valuable documentation
- Disambiguation of operators can be done at compile time
- Accuracy control: yields space optimization and gives more information to compiler about legal values
A value-centric notion of types

• A type is a collection of values...
  – ..., -100,...,0,...,100,...: integer
  – -128,...,0,...,128: char
  – {i=10, f=4.0},..., {i=20, f=5.0}:
    struct { int i; float f};
  – {i=10, f=4.0},..., {i=20, g=‘a’}: ???

• Not all sets of values make up meaningful types in programming languages
• Which set of values make up meaningful types?

So, which sets of values make up types?

• When values have some commonality
• When the type system of the language allows the set of values to be expressed!
  – Different languages allow different types: more in it later!
Organizing types

- Primitive types (e.g., int, boolean, ...)
- User-defined types
  - Ordinal types
    - values in the type are ordered
    - e.g., int
  - Composite types
    - made by applying a type constructor (e.g., record) to one or more types
  - Others: e.g., Pointer types, procedure types

E.g., ordinal type: enumerations

- \texttt{TYPE Color} = \{red, green, blue\};
  \texttt{VAR i: Color; i := Color.green;}
- Since enumerations are ordinal types, can compare the order between two values
  - e.g., \texttt{ORD(i) < ORD(j)}
Why have enumerations?

• Instead of:
  
  ```
  TYPE Color = {red, green, blue};
  VAR i: Color;
  ```

• Could have:
  
  ```
  CONST red = 1;
  CONST green = 2;
  CONST blue = 3;
  VAR i: INTEGER;
  ```

E.g. ordinal type:

Subranges

• ```
  TYPE Score = [0..100];
  VAR s: Score;
  s = 30;
  ```

• Why have subranges?
Composite types

- Further improve readability by allowing user to define specialized types
  - Records structs, objects, datatypes
  - Variant records C union,...
  - Arrays arrays in Modula-3, Java, ...
  - Sets Pascal and Modula-3 sets
  - Pointers Pointers in C, C++, ...
  - Classes (later) classes in Java, C++, M-3, ...

Records

- TYPE Point = RECORD
  x, y: INTEGER;
END;
- VAR p: Point;
- p.x := 10; p.y := 20;
  p := Point(10, 20); // sometimes!
Representation of records

p: Point

\[
\begin{array}{ll}
x & y \\
32 \text{ bit} & 32 \text{ bit}
\end{array}
\]

\[p.x /*(*(&p + 0)*/)/*
\[p.y /*(*(&p + 4)*/)/*

sizeof(int) may depend on machine (depends on how the language defines it)

Another example

\[
\text{TYPE R = RECORD}
\]

\[
\begin{array}{l}
\text{ch1: CHAR;}
\text{v: INTEGER;}
\text{ch2: CHAR;}
\end{array}
\]

\[
\text{END;}
\]

\[
\text{VAR r: R;}
\]

\[
\begin{array}{ccc}
\text{ch1} & v & \text{ch2}
\end{array}
\]

\[
\begin{array}{lll}
8 \text{ bit} & 32 \text{ bit} & 8 \text{ bit}
\end{array}
\]

\[r.v /*(*(&r + 1))*/
\]

What’s the problem?
Efficiency concerns

- On modern machines it is much faster to access memory of size $w$ bytes if it is aligned on a $w$ boundary

![Diagram showing field order in records]

Wasteful in space but faster to access
Modula-3 compilers will typically use the above representation unless the program specifies “packed”

Field order in records

- If a language allows a compiler to reorder fields then we can use a good compromise

![Diagram showing field order in records]

Since reordering interacts with things such as type equality and subtyping, most languages don’t allow it
Variant records

• Provides two or more alternative fields or collections of fields, only one of which is valid at any given time
• e.g.,
  ```c
  union U {
    int i;
    float f;
  };
  U u;
  u.i = 10;  u.f = 20.0;
  ```

So what does it really look like?

```c
struct { int i; float f; }

union { int i; float f; }
```

The fields of a union share the same space!
A little problem

- union U {
  int i;
  float f;
};
U u;
u.i = 10;
print(u.f)
- What’s the problem?

Discriminated union

- A discriminated union contains a tag that is checked on each field access and assignment
- u.i = 10;
  u.tag = “i”;
  print(check u.tag = “f”, u.f);
- Some languages such as Pascal have discriminated union
Representation of discriminated unions

```c
union { int i; float f; }
```
Arrays

- Two ways of viewing arrays
  - A sequence of values
  - A mapping from an index to a value
  - (Some languages, e.g., Modula-3, emphasize the first view)

- \texttt{VAR a: ARRAY [1..100] OF INTEGER;}
  \texttt{a[5] = 10;}
  \texttt{...print(a[5]);}

Some variations in arrays

- Are array sizes determined at compile time or at run time?
- Can the array size change dynamically at run time?
- How many dimensions does the array have?
- Are array references checked for “out-of-bounds” errors?
When is array size determined, or fixed versus open arrays

- Fixed arrays: size is fixed at compile time
  - e.g., `a: ARRAY [1..100] OF INTEGER;`
  - or `int a[100];`
  - Are the two above arrays the same size?
- Open arrays: size is not determined until run time
  - `a := NEW (ARRAY OF INTEGER, n);`
  - `a = new int[n];`
  - `procedure sort(A: array[lower..upper] of real)`

Representation of fixed arrays

```
a: ARRAY [1..100] OF INTEGER;
```

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3..99</td>
<td></td>
</tr>
</tbody>
</table>
```

```
a[i] => *(&a + (i-1))
```

0 based arrays may be more efficient since they don’t need a subtraction
Representation of open arrays

\[
\text{a} := \text{NEW (ARRAY OF INTEGER, n)}; \\
\]

0 1 n-1

Does the above representation work for open arrays?

A better representation of open arrays

\[
\text{a} := \text{NEW (ARRAY OF INTEGER, n)}; \\
\]

dope vector

A dope vector contains all the information that one needs about an open array at run time
Can open arrays be allocated on the stack?

```cpp
procedure sort(A: array[1..u] of real) {
    int i, j;
    ...
}
```

What’s the problem here?

The fix

**Observations:**
- Dope vector has a fixed size; elements have a compile-time unknown size
- Elements are accessed only by going through the dope vector: so we can place them anywhere
Comparision of fixed versus open arrays

• Expressiveness

• Simplicity

• Ease of implementation

• Efficiency

Can array sizes be changed at run-time?

• In most languages, once an array is allocated, its size remains fixed
  – Exceptions: C/C++ (sometimes with realloc, vectors in Java standard libraries, Smalltalk using “become”)
Single or multi-dimensional

- FORTRAN, Modula-3, Pascal etc. have multi-dimensional arrays
  - `a: ARRAY[1..20,1..15] OF INTEGER;
    a[i,j] := 10;`

- Java gets some of the power of multi-dimensional arrays using arrays of arrays
  - `a = new int [10][20]` is not a multi-dimensional array but an array of arrays

Difference is easy to see in the representation

```
a: ARRAY[1..2,1..15] OF INTEGER;
```

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>...</th>
<th>1,15</th>
<th>2,1</th>
<th>2,2</th>
<th>...</th>
<th>2,15</th>
</tr>
</thead>
</table>

`a[i,j] => &a + (i-1)*15 + (j-1)`

- e.g., `a[1,5] => &a + (1-1)*15 + (5-1) => &a + 4`
- e.g., `a[2,3] => &a + (2-1)*15 + (3-1) => &a + 17`

Accessing any element of the 2 (or n-dimensional) fixed array involves adding a constant to the array base
Representation of array of array...

```
a = new int [2][]
a[1] = new int[10]
```

\[ a[i,j] \Rightarrow a[i] + j \]

multi-dimensional versus array of array...

- Expressiveness
- Simplicity
- Ease of implementation
- Efficiency
Out of bounds errors

- In Modula-3 and Java all references to arrays are checked for out of bounds errors
- In C and C++ there is no such check
  - Thus, there is no need for dope vectors in dynamically allocated C/C++ arrays
- We will see more on this in the next topic

Sets

- An unordered collection of an arbitrary number of distinct values of a common type
- Colors = \{red, blue, green\};
  a, b, c: SET of Colors
- a := a + Colors.red;
  b := ... 
  c := a + b (union)
  c := a * b (intersection)
  IF Colors.red IN c THEN
Representation of sets

c: SET of Colors
Use 1 bit per possible member

(Of course, with alignment constraints, Set of Colors will be more than 3 bits)

Efficiency consideration of sets

• SET OF INTEGER
• Like switch statements, sets lose their compactness and efficiency if the possible membership is large!
Pointers

• Pointer: a reference to some object
• Critical for building recursive data structures
• `struct LL {
   int val;
   struct LL *next;
};`
• `struct LL {
   int val;
   struct LL next;
}`

Memory representation of pointers

• Usually represented as memory addresses

Many languages allow a pointer to be represented as an index into a table of addresses
Issues with pointers

- Safety
  - *p: is “p” a valid pointer?
- Memory management
  - when to free an object referenced by a pointer?
  - more on this in a few lectures

Relationship to readings

- Reading goes into a wider range of examples, particularly of variant records and arrays
- Reading presents more details on implementation
  - Very useful to know since it gives good insight into the types themselves
Next topic: Relationship between types

- Reading: 7.2