Programming Languages

Tevfik Koşar

Lecture - XVI
March 16th, 2006

Roadmap

- Type Inference
- Records (Structures)
- Variant Records (Unions)
Type Inference

Suppose the following operation:

```plaintext
type Atype = 0..20
Btype = 10..20;
var  a: Atype;
    b: Btype;
```

What will be the type of `a+b`?

Possible values range from 10 to 40. So will the type 0..40?

In usual cases, the result of an arithmetic operation on a subrange has the subrange's base type, eg. `integer`

---

Records (Structures)

- **In C:**
  ```c
  struct element{
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
  }
  ```

- **In Pascal:**
  ```plaintext
type element = record
    name: two_chars;
    atomic_number: integer
    atomic_weight: real;
    metallic: Boolean;
  end;
  ```
Records (Structures)

Field layout for type `element`:

- `name`: two_chars
- `atomic_number`: integer
- `atomic_weight`: real
- `metallic`: Boolean

Figure 7.1: Likely layout in memory for objects of type `element` on a 32-bit machine. Alignment restrictions lead to the shaded “holes.”

- Fields of a record are stored in adjacent locations in memory.
- Compiler keeps track of the offset of each field within each record type.
- The `element` record occupies 20 bytes of memory (5 bytes are wasted)
- In an array of `element`s, compiler will denote 20 bytes for each member

Records (Structures)

Field layout for packed `element`:

- `name`: two_chars
- `atomic_number`: integer
- `atomic_weight`: real
- `metallic`: Boolean

Figure 7.2: Likely memory layout for packed `element` records. The `atomic_number` and `atomic_weight` fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

- Space optimization by pushing fields together
- To access a nonaligned field, compiler has to issue a multi-instruction sequence (retrieve multiple pieces and reassemble)
- Now `element` record consumes only 15 bytes
Records (Structures)

<table>
<thead>
<tr>
<th>name</th>
<th>metallic</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_number</td>
<td></td>
</tr>
<tr>
<td>atomic_weight</td>
<td></td>
</tr>
</tbody>
</table>

4 bytes/32 bits

\[
\text{type} \text{ element} = \text{record} \\
\quad \text{name: } \text{two_chars; } \\
\quad \text{metallic: } \text{Boolean; } \\
\quad \text{atomic_number: } \text{integer} \\
\quad \text{atomic_weight: } \text{real;} \\
\quad \text{end; }
\]

Figure 7.3: **Rearranging record fields to minimize holes.** By sorting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.

- An alternative way would be rearranging record’s fields
- Some compilers do this automatically
- Now \text{element} record consumes 16 bytes (only 1 byte wasted)

Variant Records (Unions)

- Provide two or more alternative fields, only one is valid at a given time.

\[
\text{type} \text{ element} = \text{record} \\
\quad \text{name: } \text{two_chars;} \\
\quad \text{atomic_number: } \text{integer} \\
\quad \text{atomic_weight: } \text{real;} \\
\quad \text{metallic: } \text{Boolean; } \\
\quad \text{case } \text{naturally_occuring}: \text{Boolean of} \\
\quad \quad \text{true:} ( \\
\quad \quad \quad \text{source: } \text{string_ptr;} \\
\quad \quad \quad \text{prevalence: } \text{real;} \\
\quad \quad \quad ) \\
\quad \quad \text{false:} ( \\
\quad \quad \quad \text{lifetime: } \text{real;} \\
\quad \quad \quad ) \\
\quad \text{end; }
\]
Variant Records (Unions)

<table>
<thead>
<tr>
<th>name</th>
<th>atomic_number</th>
<th>atomic_weight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>metallic</td>
<td>&lt;true&gt;</td>
<td>source</td>
</tr>
<tr>
<td>prevalence</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>atomic_number</td>
<td>atomic_weight</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>metallic</td>
<td>&lt;false&gt;</td>
<td>lifetime</td>
</tr>
</tbody>
</table>

Figure 7.4: Likely memory layouts for element variants. The value of the naturally_occuring field (shown here with a double border) determines which of the interpretations of the remaining space is valid. Type string_ptr is assumed to be represented by a (four-byte) pointer to dynamically allocated storage.

Variant Records (Unions)

type tag = (is_int, is_real, is_bool);
var test: record
    case which: tag of
        is_int : (i: integer);
        is_real: (r:real);
        is_bool: (b:Boolean);
    end;
----

test.which := is_real;
test.r := 3.0;
writeln(test.r);

Output: 3.0
Variant Records (Unions)

type tag = (is_int, is_real, is_bool);
var test: record
case which: tag of
  is_int  : (i: integer);
  is_real: (r:real);
  is_bool: (b:Boolean);
end;

----

test.which := is_real;
test.r := 3.0;
writeln(test.i);

Dynamic semantic error!

Variant Records (Unions)

type tag = (is_int, is_real, is_bool);
var test: record
case which: tag of
  is_int  : (i: integer);
  is_real: (r:real);
  is_bool: (b:Boolean);
end;

----

test.which := is_real;
test.r := 3.0;
test.which := is_int;
writeln(test.i);

Not an error, but the output will be junk!
Variant Records (Unions)

```
type tag = (is_int, is_real, is_bool);
var test: record
    case tag of
        is_int : (i: integer);
        is_real: (r:real);
        is_bool: (b:Boolean);
    end;
    ----
    test.which := is_real; \not required 
test.r := 3.0;
writeln(test.i);

Not an error, but the output will be junk!
```

Variant Records (Unions)

- Variant records with tags: discriminated unions
- Variant records without tags: nondiscriminated unions
Figure 7.5: **Likely memory layout for a variant record in Modula-2.** Here the variant portion of the record is not required to lie at the end. Every field has a fixed offset from the beginning of the record, with internal holes as necessary following small-size variants.