This Lecture

Data Representation: how to store various kinds of data.

- General issues
- Primitive types
- Record types
- Arrays
- Disjoint unions
- Recursive types

Data Representation?

- Objective: to store various kinds of data. Integers, characters, strings, arrays, trees, ...
- At our disposal: the memory:

<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>10200008</td>
<td>3E124C21</td>
</tr>
<tr>
<td>1020000C</td>
<td>FE7B1811</td>
</tr>
<tr>
<td>10200010</td>
<td>7A7CBB81</td>
</tr>
</tbody>
</table>

- We need to encode the data to be stored.

Nonconfusion (1)

Self-evident: if two different values are represented the same way, they cannot be told apart.

- Dynamically checked language: Every possible value must have a distinct representation.
- (Statically) typed language: Values of the same type must have distinct representations; the same representation may be reused for values of different types.

Uniqueness

Comparison of values is facilitated if each value has exactly one representation.

However, not essential. Common exceptions:

- 0 is represented by both 00...00 and 11...11 in the ones-complement representation of integers.
- Floating-point representations typically have a separate sign bit. Thus, the representation of +0 is distinct from the representation of −0.

Nonconfusion (3)

Example: Consider two enumeration types:

```plaintext
data Colour = Red | Green
data Size = Small | Large
```

It must always be the case that

- `repr(Red) ≠ repr(Green)`
- `repr(Small) ≠ repr(Large)`

Further, in a dynamically checked setting:

```plaintext
{repr(Red), repr(Green)} \ ∩ \ {repr(Small), repr(Large)} = ∅
```

Data Representation: Issues (2)

- Constant-size representation: The representations of all values of a given type occupy the same amount of space.
- Direct or indirect (via pointer) representation.

Constant-size representation enables compiler to statically plan storage allocation (since type and hence size is known statically).

If not possible/too wasteful: use some form of indirect representation.
### Direct or Indirect Representation

- **Direct representation:** the representation of a value $x$ is the binary representation of $x$: $\text{repr.} x$

- **Indirect representation:** $x$ represented by a handle that points to a binary representation of $x$ (on the stack or in the heap): $\text{repr.} x$

### Pros direct representation:
- efficient access
- no heap allocation/deallocation overhead

### Pros indirect representation:
- supports varying size data (like dynamic arrays)
- supports recursive types (like linked lists, trees)
- facilitates implementation of parametric polymorphism (as handles can be uniform)

### Representing Primitive Types

Primitive types are often supported directly by the underlying hardware. For example, a 32-bit machine might support:
- addressing of 8-bit bytes and 32-bit words
- 32-bit two's-complement integer arithmetic
- 64-bit floating point operations

There are also standard encoding conventions, such as the 7-bit ASCII or 8-bit ISO character codes, or the Unicode standard. Adopting such conventions facilitates interoperability and communication.

### Representing Records

A record consists of several fields, each of which has an identifier. For example:

```plaintext
type Date = record
  y: Integer,
  m: Integer,
  d: Integer
end;
```

```plaintext
type Details = record
  female: Boolean,
  dob: Date,
  status: Char
end;
```

### Representation of records:
- Sequence of representations of individual fields.
- Caveat: alignment restrictions. The underlying architecture might require that e.g. word-sized quantities start at a word boundary.
- Relaxing this is possible, but may require extra work; e.g., accessing a word byte by byte (four instructions instead of one).

### Alignment

- An address $a$ is $n$-byte aligned iff $a \equiv 0 \pmod{n}$.
- A variable/field etc. is $n$-byte aligned iff it is stored starting at an $n$-byte aligned address.
- To satisfy alignment requirements of its components, a variable of aggregate type like a record is often aligned according to the maximum alignment of its components.
- Padding often needed between variables/components to ensure the alignment requirements of each is met.

### Exercise: Representing Records

Assume:
- 1 word = 4 byte = 32 bit Integers
- 1 byte = 8 bit Boolean and Char
- Integer must be word aligned

What is the alignment and size of the type `Date`?

```plaintext
type Date = record
  y: Integer,
  m: Integer,
  d: Integer
end;
```

What is the alignment and size of the type `Details`?

```plaintext
type Details = record
  female: Boolean,
  dob: Date,
  status: Char
end;
```

Given a variable $x : \text{Details}$, what are the addresses of $x.female, x.dob.y, x.dob.m, x.dob.d, x.status$ relative to $\text{addr}(x)$?
Exercise: Representing Records (3)

Size of Date is 3 32-bit words, size of Details is 1 + 3 + 1 = 5 32-bit words:

<table>
<thead>
<tr>
<th>variable</th>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.female</td>
<td>addr(x)</td>
<td>1 (true)</td>
</tr>
<tr>
<td>x.dob.y</td>
<td>addr(x) + 4</td>
<td>1984</td>
</tr>
<tr>
<td>x.dob.m</td>
<td>addr(x) + 8</td>
<td>7</td>
</tr>
<tr>
<td>x.dob.d</td>
<td>addr(x) + 12</td>
<td>25</td>
</tr>
<tr>
<td>x.status</td>
<td>addr(x) + 16</td>
<td>117 ('u')</td>
</tr>
</tbody>
</table>

Representing Arrays (1)

- Array represented by sequence of representations of individual array elements.
- Two cases:
  - **Static Array**: Number of elements known at compile time.
  - **Dynamic Array**: Number of elements determined at run time.
- When accessing array elements, must ensure indices are within bounds.
- Address of element computed from base address of array, index, and size of elements.

Representing Arrays (2)


```plaintext
var a: Integer[10] (at [SB + 0])
LOADL 7
LOADA [SB + 0] JUMP #1
LOAD 3 #0: CALL ixerror
LOAD [ST - 1] #1: LOADL 1
LOAD 0 MUL
LOAD [ST - 1] ADD
JUMP #0 STOREI 0
LOAD [ST - 1] LOADL 10
```

Representing Arrays (3)

Representing Arrays (4)

- **Dynamic array**: size of array not known at compile time.
- **indirect representation**: array accessed via a handle
  - handle itself has **fixed size**
  - handle contains **pointer** to array proper and the **array bounds**
  - **storage** for array proper allocated at runtime
  - index checked by comparing with array bounds stored in the handle.

Representing Disjoint Unions (1)

- A disjoint union consists of a **tag** and a **variant** part.
- The value of the tag determines the type of the variant part.
- Mathematically: $T = T_1 + \ldots + T_n$; given tag $i$, the variant part is a value chosen from type $T_i$.
- Disjoint unions occur as
  - **variant records** in Pascal and Ada
  - **algebraic data types** in Haskell and ML
  - **object types** in OO languages like Java, C#

Representing Disjoint Unions (2)

- A disjoint union can be represented like a record.
- The value of the tag field determines the layout of the rest of the record.
- If constant size is necessary, size is the maximal size over the various possible layouts.

Representing Disjoint Unions (3)

Some Haskell Examples:

- `data OptInt = NoInt | JustInt Int`
  - The first tag is `NoInt`; no variant part.
  (Which is the same as saying that we have a trivial variant part of the unit type `()`)  
  - The second tag is `JustInt`; the variant part is a single integer field.

Representing Disjoint Unions (4)

- `data Colors = Red | Green | Blue`
  - three tags; no variant parts.
  - this is thus just an **enumeration type**.
Representing Recursive Types (1)

- A recursive type is one defined in terms of itself.
- Examples are linked lists and trees.
- Recursive types are usually represented indirectly since this allows values of arbitrary size to be referenced through a fixed size handle.

Representing Recursive Types (2)

In languages like C or Pascal, the programmer needs to introduce the indirect representation explicitly through pointer types.

Consider the following Pascal declarations:

```pascal
type IntList = ^IntNode;
IntNode = record
  head: Integer;
  tail: IntList
end;
var primes: IntList
```

Uniform Representation (1)

Languages like Haskell and ML adopts a uniform data representation: all values (even "primitive" ones) have an indirect representation (pointer):

- Uniform representation facilitates parametric polymorphism. E.g., the identity function
  ```plaintext
  id x = x
  ```
  can be compiled to a single piece of code working for values of any type because all values are represented same way. (Only when the value proper is accessed will differences be apparent and code no longer polymorphic.)

Uniform Representation (2)

- Recursive types supported automatically: “everything is already a pointer”.
- Many OO languages, like Java and C#, adopt a mostly uniform representation:
  - All objects are represented by pointers.
  - Recursive types thus supported.
  - OO-style polymorphism: an object of a class is also an object of any of the superclasses.
  - Layout of “common part” of object uniform to allow superclass method to work on subclass objects.

Example: Haskell Tree Type (1)

This example illustrates
- disjoint union representation
- recursive type representation
- uniform representation (through pointers) of values of all types.

```haskell
data Tree = Leaf Int
          | Node Tree Tree
```

Example: Haskell Tree Type (2)

```haskell
aTree = Node (Leaf 1)
       (Node (Leaf 2) (Leaf 3))
```

Example: Haskell Tree Type (3)

Of course, the tags (`Leaf`, `Node`, and `INT`) must also be represented. Two possibilities:

- A small integer, subject to nonconfusion. E.g.
  ```plaintext
  Leaf = 0, Node = 1, INT = 0
  ```
  (Representing both `Leaf` and `INT` with the small integer 0 does not lead to confusion in a statically typed language like Haskell.)
- A pointer to an information table.

Example: Haskell Tree Type (4)

```plaintext
<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10200008</td>
<td>INT</td>
</tr>
<tr>
<td>1020000C</td>
<td>1</td>
</tr>
<tr>
<td>10200010</td>
<td>INT</td>
</tr>
<tr>
<td>10200014</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2E4D0100</td>
<td>Leaf</td>
</tr>
<tr>
<td>10200008</td>
<td>INT</td>
</tr>
<tr>
<td>10200010</td>
<td>INT</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Example: Haskell Tree Type (5)

```plaintext
<table>
<thead>
<tr>
<th>address</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2E4D0104</td>
<td>10200010</td>
</tr>
<tr>
<td>2E4D0108</td>
<td>Leaf</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2E4D010C</td>
<td>10200018</td>
</tr>
</tbody>
</table>
```