# COMP3048: Lecture 15

## *Run-Time Organization II*


Matthew Pike

University of Nottingham, Ningbo

# 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***:

| address | contents |
| --- | --- |
| . . . | . . . |
| 10200008 | 3E124C21 |
| 1020000C | FE7B3811 |
| 10200010 | 7A7CBBA3 |
| . . . | . . . |

- We need to ***encode*** the data to be stored.

# Data Representation: Issues (1)

- ***Nonconfusion***: Different values of a given type ***must*** have different representations.
- ***Uniqueness***: Each value should have exactly one representation.

Note: The discussion concerns ***run-time*** representation. Any value that is known ***statically*** potentially need no run-time representation at all, or might be represented in some specialised way on a case-by-case basis (e.g. multiplication by constant, loading large constants).

# 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.

# Nonconfusion (2)

Example: suppose both characters and small integers represented by 8-bit bytes:

- $\mathrm{repr}(\texttt{'A'}) = 01000001$
- $\mathrm{repr}(\texttt{65}) = 01000001$

Suppose a variable `x` contains this value $01000001$: Should `print(x)` print `'A'` or 65?

- No way to tell the representation of `'A'` and `65` apart in a dynamically checked setting.
- In a statically typed setting, the type is used to disambiguate.

# Nonconfusion (3)

Example: Consider two enumeration types:

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

It must *always* be the case that

$$\begin{aligned} \mathrm{repr}(\texttt{Red}) &\neq \mathrm{repr}(\texttt{Green}) \\ \mathrm{repr}(\texttt{Small}) &\neq \mathrm{repr}(\texttt{Large}) \end{aligned}$$

Further, in a dynamically checked setting:

$$\{\mathrm{repr}(\texttt{Red}), \mathrm{repr}(\texttt{Green})\} \cap \{\mathrm{repr}(\texttt{Small}), \mathrm{repr}(\texttt{Large})\}$$
$$= \emptyset$$

# Uniqueness

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

However, not essential. Common exceptions:

- $0$ is represented by both $00\ldots00_2$ and $11\ldots11_2$ 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 $-0$.

# Data Representation: Issues (2)

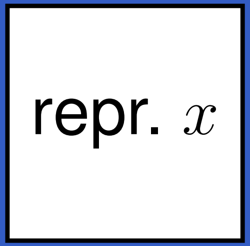
- ***Constant-size representation***: The representations of all values of a given type occupy the same amount of space.
- ***Direct*** or ***indirect*** (via pointer) representation.

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

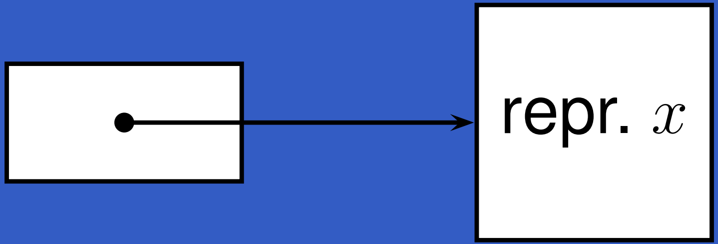
If not possible/too wasteful: use some form of indirect representation.

# Direct or Indirect Representation (1)

- ***Direct representation***: the representation of a value $x$ is the binary representation of $x$:




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



# Direct or Indirect Representation (2)

- 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 (1)

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 twos-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 Primitive Types (2)

On such a 32-bit machine, the following would be natural representation choices:

| Type | Representation | Size |
|---|---|---|
| `Boolean` | 0 for false; 1 for true | 8-bit byte |
| `Char` | ISO Latin 1 encoding | 8-bit byte |
| `Integer` | twos-complement repr. | 32-bit word |
| `Real` | floating point repr. | 64-bit word |

# Representing Records (1)

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

```
type Date = record
                y: Integer,
                m: Integer,
                d: Integer
            end;
type Details = record
                   female: Boolean,
                   dob:    Date,
                   status: Char
               end;
```

# Representing Records (2)

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 (1)

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?

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

# Exercise: Representing Records (2)

What is the alignment and size of the type Details?

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

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

# Exercise: Representing Records (3)

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

| variable | address | contents | |
|---|---|---|---|
| `x.female` | addr(`x`) | `1` | (true) |
| `x.dob.y` | addr(`x`) + 4 | `1984` | |
| `x.dob.m` | addr(`x`) + 8 | `7` | |
| `x.dob.d` | addr(`x`) + 12 | `25` | |
| `x.status` | addr(`x`) + 16 | `117` | ('u') |

# 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)

***Static array***: required storage space and array bounds known at compile time. Consider:

$$\texttt{var x :}\ \ T\texttt{[}n\texttt{]}$$

- Required storage: $n \times \mathrm{sizeof}(T)$
- Access of $\texttt{x[}i\texttt{]}$:
  - Verify that $0 \leq i \leq (n-1)$
  - Compute address $a$ of desired element:

$$a = \mathrm{addr}(\texttt{x[0]}) + \texttt{i} \times \mathrm{sizeof}(T)$$

  - Fetch/store value at address $a$.

# Representing Arrays (3)

Example: TAM code for `a[3] := 7` given `var a: Integer[10]` (at `[SB + 0]`)

```
    LOADL     7                    LSS
    LOADA     [SB + 0]             JUMPIFNZ #1
    LOADL     3               #0:  CALL     ixerror
    LOAD      [ST - 1]        #1:  LOADL    1
    LOADL     0                    MUL
    LSS                            ADD
    JUMPIFNZ  #0                   STOREI   0
    LOAD      [ST - 1]
    LOADL     10
```

# 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 Shape`
  ```
  = Triangle Point Point Point
  | Rectangle Point Point
  | Circle Point Radius
  ```
  - three tags; the variant parts are:
    - `Point` triple
    - `Point` pair
    - `Point` and `Radius` pair.
- `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:

```
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

  ```
  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)

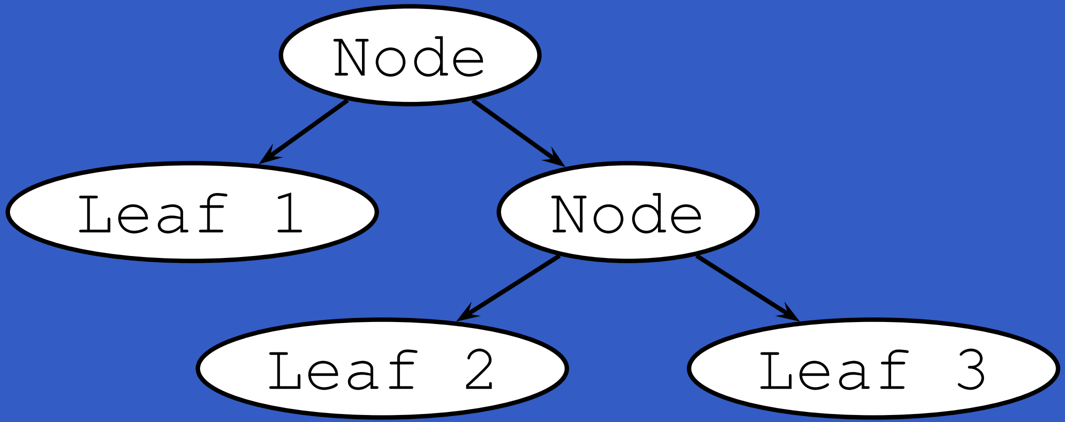
This example illustrates

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

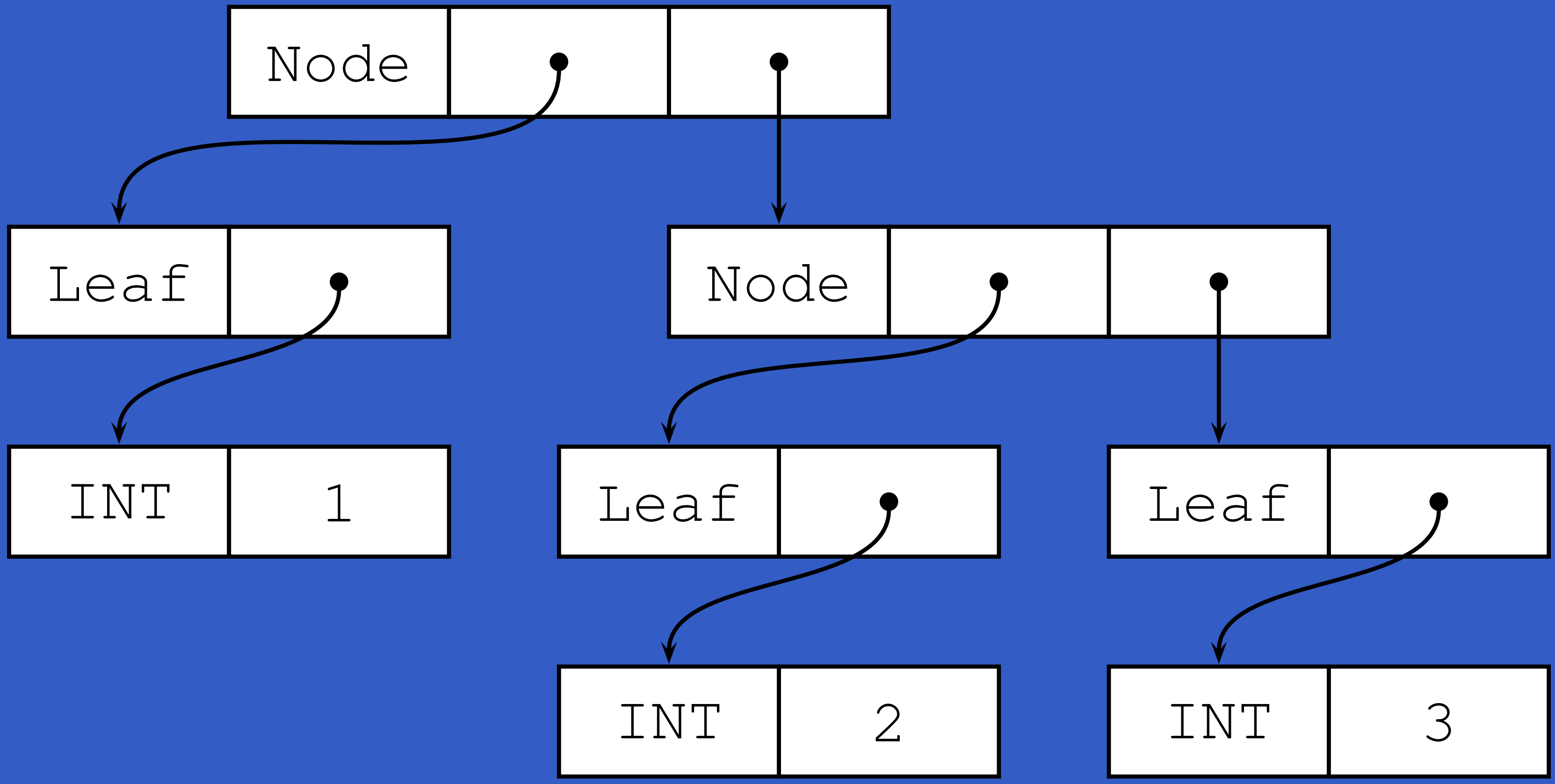
# Example: Haskell Tree Type (2)

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

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



# Example: Haskell Tree Type (3)



# Example: Haskell Tree Type (4)

| address | contents |
|---|---|
| ... | ... |
| 10200008 | INT |
| 1020000C | 1 |
| 10200010 | INT |
| 10200014 | 2 |
| ... | ... |
| 2E4D0100 | Leaf |
| 2E4D0104 | 10200010 |
| 2E4D0108 | Leaf |
| 2E4D010C | 10200018 |
| ... | ... |

| address | contents |
|---|---|
| ... | ... |
| 2E4D0200 | Node |
| 2E4D0204 | 2E4D0100 |
| 2E4D0208 | 2E4D0108 |
| 2E4D020C | Leaf |
| 2E4D0210 | 10200008 |
| 2E4D0214 | Node |
| 2E4D0218 | 2E4D020C |
| 2E4D021C | 2E4D0200 |
| ... | ... |

# Example: Haskell Tree Type (5)

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

- A small integer, subject to nonconfusion. E.g.

$$\texttt{Leaf} = 0,\ \texttt{Node} = 1,\ \texttt{INT} = 0$$

  (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.