Data Structures

Programmers usually define composite types in order to implement **data structures** appropriate to an application and/or algorithm.

**Abstractly**, such data structures can be seen as mathematical operators on underlying **sets** of simpler values. A small number of type operators suffices to describe most useful data structures:

- Cartesian product \( (S_1 \times S_2) \)
- Disjoint union \( (S_1 + S_2) \)
- Mapping (by explicit enumeration or by formula) \( (S_1 \rightarrow S_2) \)
- Set \( (P^S) \)
- Sequence \( (S^*) \)
- Recursive structures (lists, trees, etc.)

**Concretely**, each language defines the internal **representation** of values of the composite type, based on the type constructor and the types used in the construction.

Example: The fields of a record might occupy successive memory addresses (perhaps with some alignment restrictions). The total size of the record is (roughly) the sum of the field sizes.

Often a range of representations are possible, from highly packed to highly indirected. There’s often a tradeoff between space and access time.
Representation of Data Structures

Historically, most languages provide direct representations only for a few data structures, usually those whose values can be represented efficiently on a conventional computer. Often, they are restricted so that all values will be of fixed size.

For conventional languages, this is the short list:

- **Records.**
- **Unions.**
- **Arrays.**

Many languages also support manipulation of pointers to values of these types, in order to allow moving data “by reference” and to support recursive structures.
Records = Cartesian Products

Records, tuples, “structures”, etc. Nearly every language has them.

“Take a bunch of existing types and choose one value from each.”

Examples (ML syntax):

```ml
  type emp = string * int (unlabeled fields)
  type emp =
    {name: string, age: int} (labeled fields)
```

ML also permits record values to be written without declaring explicit named type first.

(Java syntax)

```java
  class emp {
    String name;
    int age;
  }
```


Representation: Usually as described above. Because records may be large, they are often manipulated by reference, i.e., represented by a pointer. The fields within a record may also be represented this way.
Records (continued)

Allowed contexts: In many languages, treated like primitive values, e.g., can be assigned as a unit, passed to or returned by functions, etc. But since they may be large, some languages add restrictions.

Literals: Most languages allow a literal record to be specified by specifying each component, either by position or by name. Some require components to be initialized after creation.
**Disjoint Unions**

Variant records, discriminated records, unions, etc.

“Take a bunch of existing types and choose one value from one type.”

Pascal Example:

```pascal
type RESULT = record
    case found : Boolean of
        true: (value:integer);
        false: (error:STRING)
    end;

function search (...) : RESULT;
...
```

Generally behave like records, with **tag** as an additional field.

Represented by the variant’s representation, usually plus a tag (thus forming a record). Size typically equals the size of the largest variant plus tag size.
Variant Insecurities

Pascal variant records are insecure because it is possible to manipulate the tag independently from the variant contents.

```pascal
tr.value := 101;
write tr.error;

if (tr.found) then begin
  ...
  tr := trl;
  x := tr.value
```

These problems were fixed in Ada by requiring tag and variant contents to be set simultaneously, and inserting a runtime check on the tag before any read of the variant contents.
Disjoint Unions Done Properly

ML has very clean approach to building and inspecting disjoint unions:

```ml
datatype result =
    FOUND of integer |
    NOTFOUND of string

fun search (...) : result =
    if ... then
        FOUND 10
    else
        NOTFOUND "problem"

val r = search (...)

case r of
    FOUND x =>
        print ("Found it : " ^ (Int.toString x))
    | NOTFOUND s =>
        print ("Couldn’t find it : " ^ s)
```

Here `FOUND` and `NOTFOUND` tags are not ordinary fields. Case combines inspection of tag and extraction of values into one operation.

Java doesn’t support disjoint unions directly, but subclasses provide a (somewhat awkward) way to achieve the same effect.
Arrays and Mappings

Basic implementation idea: a table laid out in adjacent memory locations permitting indexed access to any element.

Mathematically: A finite mapping from an index set to a component set.

Index set is nearly always a set of integers 0..n, where n is small enough to allow space for the entire array, or some other small discrete set isomorphic to them.

More general index sets are seldom supported directly by language because of the lack of a single, uniform, good implementation. Arrays with arbitrary index sets are sometimes called “associative arrays”

Many languages require the index set (and hence size) of arrays to be specified as part of each array type declaration. Others permit the size independently for each array value, when the array is first created. (Java and ML both do this.) Arrays are often large, and hence manipulated by reference. They may or may not be first-class.

Major security issue for arrays is bounds checking of index values. In general, it’s not possible to check all bounds at compile time (though often possible in particular cases). Runtime checks are always possible, but may cost.
Functions and Mappings

Mathematical mappings can also be represented by an algorithmic **formula**.

A **function** gives a “recipe” for computing a **result** value from an **argument** value.

A program function can describe an infinite mapping.

But differs from mathematical function in that:

- it must be specified by an explicit algorithm
- executing the function may have side-effects on variables.

It can be very handy to manipulate functions representing mappings as first-class values.
Sequences

What about data structures of essentially **unbounded** size, such as **sequences** (or **lists**)?

“Take an arbitrary number of values of some type.”

Such data structures require special treatment: they are typically represented by small segments of data linked by pointers, and dynamic storage allocation (and deallocation) is required.

The basic operations on a sequence include **concatenation** (especially concatenating a single element onto the head or tail of an existing sequence) and **extraction** of elements (especially the head). Best representation depends heavily on what nature and frequency of various operations.

An important example is the (unbounded) **string**. Although it’s hard to give a single, uniformly efficient implementation for them, they are so useful that languages increasingly do provide a built-in implementation. (Both ML and Java do; Java’s is not completely “built-in.”)
Defining Sequences

Unless the programming language supports sequences directly, the programmer must define them using a recursive definition.

For example, a list of integers is either

- empty, or
- has a head which is an integer and tail which is itself a list of integers.

ML has particularly clean mechanisms for describing recursive types.

```ml
datatype intlist =
  EMPTY
| CELL of int * intlist
```

Internally, the non-empty case can be represented by a two-element heap-allocated record, containing an integer and a pointer to another list. (Obviously, the tail list itself cannot be embedded in the record, since it’s size is unknown.) The empty case is conveniently represented by a null pointer.
ML’s built-in lists

Actually, lists are so commonly used in ML that they are built in, with essentially the following definition:

\[
\begin{align*}
\text{infixr} &::= \\
\text{datatype } 'a \text{ list} &= \\
&\quad \text{nil} \\
&\quad | :: \text{ of } 'a \times 'a \text{ list}
\end{align*}
\]

and the special notation \([e_1, e_2, \ldots, e_n] \equiv e_1 :: (e_2 :: \ldots :: (e_n :: \text{nil})\ldots)\)
Recursive Types

Recursion can be used to define and operate on more complex types, in which the type being defined appears more than once in the definition.

Example: binary trees with integer labels (only) at the leaves.

```
datatype 'a tree =
    INTERNAL of {left:'a tree,right:'a tree}
  | LEAF of {contents:'a}
```

It is much more convenient to use recursion (rather than iteration with an auxiliary stack) to process the full tree:

```
fun sum(tree: int tree) =
    case tree of
      INTERNAL{left,right} => (sum left) + (sum right)
    | LEAF{contents} => contents
    end;
```
Reference Semantics

ML implementations implicitly allocate records (and disjoint unions) on the heap, and represent record values by references (pointers) into the heap. Java does the same thing with objects (although we must say new explicitly at points of allocation).

As a natural result, both languages use shallow copy semantics for assignment and argument passing. Example:

class emp {
    String name;
    int age;
}
emp e1;
e1.age = 91;
emp e2 = e1;
e1.age = 18;
System.out.println(e2.age);

prints 18

Neither language allows user programs to manipulate the internal pointers directly. And neither supports explicit deallocation of records (or objects) either; both provide automatic garbage collection of unreachable heap values, thus avoiding both dangling pointer and memory leak bugs.
Explicit Pointers

Many previous languages had **pointer types** to enable programmers to construct recursive data structures, e.g., in C:

```c
typedef struct intcell *intlist;
struct intcell {
    int head;
    intlist tail;
}
intlist mylist =
    (intlist) malloc(sizeof(struct intcell));
while (list != NULL)
    if (list->head != i) then
        list = list->tail;
```

In most such languages, pointers are restricted to addresses returned by allocation operations, but C/C++ allows the address of **anything** to be taken and later dereferenced, and supports **pointer arithmetic**. While this feature can support very sufficient code, it also destroys the safety of the type system.
Type Equivalence

When do two identifiers have the “same” type, or “compatible” types?

I.e., if $a$ has type $t_1$, $b$ has type $t_2$ and $f$ has type $t_2 \rightarrow t_3$, how must $t_1$ and $t_2$ be related for these to make sense?

$$
a := b \\
f(a)
$$

To maintain whatever security type-checking of primitive types gives us, we must insist at a minimum that $t_1$ and $t_2$ are structurally equivalent.

Structural equivalence is defined inductively:

- Primitive types are equivalent iff they are the same type.
- Cartesian product types are equivalent if their corresponding component types are equivalent. (Record field names are typically ignored.)
- Disjoint union types are equivalent if their corresponding component types are equivalent.
- Mapping types (arrays and functions) are the same if their domain and range types are the same. (Sometimes the cardinality of the index type of an array is ignored.)
Equivalence (continued)

Another way to say this: two types are equal if they have the same set of values.

Recursive types are a problem. Are these two types structurally equivalent?

\[
\begin{align*}
\text{type } t1 &= \text{int} \times t1 \\
\text{type } t2 &= \text{int} \times t2
\end{align*}
\]

Intuitively yes, but it’s (a little) tricky for a type-checking algorithm to determine this!
Type Names

Question becomes more interesting because of type names.

We name types for two possible reasons:

- As a convenient shorthand to avoid giving the full type each time. E.g.,

  ```
  fun f(x:int * bool * real) :
  int * bool * real = ...
  type t = int * bool * real
  fun f(x:t) : t = ...
  ```

- As a way of improving program correctness by subdividing values into types according to their meaning within the program.

  ```
  type polar = record r:real, a:real end;
  type rect = record x:real, y:real end;
  function polar_add(x:polar,y:polar) : polar .
  function rect_add(x:rect,y:rect) : rect ...
  var a:polar; c:rect;
  a := (150.0,30.0) (* ok *)
  polar_add(a,a) (* ok *)
  c := a (* type error *)
  rect_add(a,c) (* type error *)
  ```

For this to be useful, some structurally equivalent types must be treated as inequivalent.


Name Equivalence

Basic idea: Two types are equivalent iff they have the same name.

Supports polar/rect distinction.

But pure name equivalence is very restrictive, e.g.:

```
type ftemp = real
type ctemp = real
var x:ftemp, y:ftemp, z: ctemp;
x := y; (* ok *)
x := 10.0; (* probably ok *)
x := z; (* type error *)
x := 1.8 * z + 32.0; (* probably type error *)
```

Different types now seem too distinct; can’t even convert from one form of real to another.

Also: what about unnamed type expressions?

```
type t = int * int
procedure f(x: int * int) = ...
procedure g(x: t) = ...
var a:t = (3,4)
g(a); (* ok *)
f(a); (* ok or not ?? *)
```

Most languages use mixed solutions.
C Type Equivalence

C uses structural equivalence for array and function types, but name equivalence for `struct`, `union`, and `enum` types. For example:

```c
char a[100];
void f(char b[]);
f(a); (* ok *)
```

```c
struct polar{float x; float y;};
struct rect{float x; float y;};
struct polar a;
struct rect b;
a = b; (* type error *)
```

A type defined by a `typedef` declaration is actually just an abbreviation for an existing type.

Note this policy makes it easy to check equivalence of recursive types, which can only be built using `structs`.

```c
struct fred {int x; struct fred *y;} a;
struct bill {int x; struct fred *y;} b;
a = b; (* type error *)
```
ML Type Equivalence

ML uses structural equivalence, except that each datatype declaration creates a new type unlike all others.

```
datatype polar = POLAR of real * real
datatype rect = RECT of real * real
val a = POLAR(1.0,2.0)
and b = RECT(1.0,2.0)
if (a = b) ... (* type error *)
```

Note that the mandatory use of constructors makes it possible to uniquely identify the types of literals.

Note that a datatype need not declare a record:

```
datatype fahrenheit = F of real
datatype centigrade = C of real
val a = F 150.0
val b = C 150.0
if (a = b) ... (* type error *)
fun convert(C x) = F(1.8 * x + 32.0) (* ok *)
```

For type abbreviation, ML offers the `type` declaration, which simply gives a new name for an existing type.

```
type celsius = centigrade
fun g(x:celsius) = if x = b ... (* ok *)
```
Abstract Data Types

Is a new type name a genuinely new type, equivalent to the built-in types?

Ideally, to mimic the behavior of built-in types, user-defined types should have an associated set of operators, and it should only be possible to manipulate types via their operators (and maybe a few generic operators such as assignment or equality testing).

In particular, when new types are given a representation in terms of existing types, it shouldn’t be possible for programs to inspect or change the fields of the representation.

Such a type is called an abstract data type (ADT), because to clients (users) of the type, its implementation is hidden.

We can implement an ADT by combining a type definition together with a set of function operating on the type into a module (or package, cluster, class, etc.) Additional hiding features are needed to make the type’s representation more-or-less invisible outside the module.
Example: Environments in SML

signature ENV =
sig
  type env
  val empty : env
  val extend : env -> (string * int) -> env
  val lookup : env -> string -> int option
end

structure Env :> ENV =
struct
  type env = (string * int) list
  val empty = nil
  fun extend env (k,v) = (k,v)::env
  fun lookup ((k0,v0)::rest) k =
    if k = k0 then SOME v0
    else lookup rest k
  | lookup nil k = NONE
end (* Env *)

• ML also has a (non-module-based) abstype declaration form, but module form is just as powerful and more flexible.
Example: Environments in Java

class Env {
    private Link contents = null;
    private Env (Link c) {
        contents = c;
    }

    static final Env empty = new Env(null);

    Env extend(String k, int v) {
        return new Env(new Link(k,v,contents));
    }

    int lookup(String k) {
        Link c = contents;
        while (c != null) {
            if (c.key.equals(k))
                return c.value;
            else
                c = c.next;
        }
        return -1;
    }
}

- Note role of private constructor.
- Don’t confuse abstract data types with abstract classes!
Interface vs. Implementation

Ideally, the client of an ADT is not supposed to know or care about its internal implementation details – only about its exported interface. Thus, it makes sense to separate the textual description of the interface from that of the implementation, e.g., into separate files.

For example, ML distinguishes signatures (module specifications) from structures (module bodies), and encourages them to be in separate files. Specifications give the names of types, and the names and types of functions in the package. Bodies give the definitions of the types and functions mentioned in the specification, and possibly additional private definitions.

One advantage of this separation is that clients of module X can be compiled on the basis of the information in the specification of X, without needing access to the the body of X (which might not even exist yet!)

Many languages, particularly in the C/C++ tradition, don’t make this separation very cleanly. Java supports it rather awkwardly using abstract classes (interfaces don’t quite work for this purpose).
Is abstraction always desirable?

Although the idea of defining explicitly all the operators for a type makes good logical sense, it can get quite inconvenient.

Programmers are used to assigning values or passing them as arguments without worrying about their types. They may also expect to be able to compare them, at least for equality, without regard to type.

So most languages that support ADT’s have built-in support for these basic operations, defined in a uniform way across all types. They also usually have facilities for overriding the built-in definitions with type-specific versions. (Some of the complexity of C++ derives from this.)

Unfortunately, it is impossible to generate code for operations that move or compare data without knowing things like the size and layout of the data. But these are characteristics of the type’s implementation, not its interface. So these “universal” operations break the abstraction barrier around type.

Thus, supporting these operations conflicts with separate compilation, often weakening support for the latter. The problem can also be solved, at some cost in efficiency, by treating all abstract values as fixed-size pointers to heap-allocated values.