Data Abstraction

The elementary data structures in conventional processors are *memory cells* and *addresses of memory cells*. Compound data structures can be constructed using collections of memory cells. It is almost always a poor idea to conceptualize data at the level of hardware architecture. **Data abstraction** allows us to think about data forms conceptually, using representations which map closely onto the problem being addressed and the intuitive way we think about that problem.

The dominant error in data structure design is to confuse levels of modeling, to design for hardware or software when the problem is in the real world. A similar error is to think that hardware and software data structures model a real problem. Rather than asking: what data structure models the problem?, we should ask: what data structure should I construct to implement the mathematical model of the problem?

Data structures should be designed abstractly. This means that the data structure is *conceptually independent* of the implementation strategy. This is achieved by **abstraction barriers**, functions which isolate the implementation details from the data abstraction. When the storage format, or the implementation approach, or the internal representation, or the algorithm changes, the data abstraction itself does not.

Abstract data structures often closely model mathematical data structures. A **mathematical structure** consists of

- a **representation** of the elementary unit or constants, the **base** of the structure
- **recognizer** predicates which identify the type of the structure
- a **constructor** function which builds compound structures from simple units
- an **accessor** function which gets parts of a compound structure
- a collection of **invariants,** or equations, which define the structure's behavior
- possibly, a collection of **functions** which compute properties of specific structures
- an **induction principle** which specifies how a form is constructed and decomposed

An implemented data structure should have each of the above functionalities, and no others. Most modern languages permit construction of these functionalities, however very few provide the actual tools which would make implementation easy. Accessors, constructors, and recognizers are best expressed in a pattern-matching language; invariants are best implemented in a declarative language; and induction requires special features which are usually included only in theorem provers.

Several examples of data types expressed as abstract data structures follow. The example of *Natural Numbers* illustrates a simple ADS. The example of *Trees* is more complete, including the mathematical axioms and some recursive function definitions, The example of *Strings* illustrates an **abstract domain theory**, and includes specialized functions, an induction principle, and an example of symbolic proof by induction.

Abstract Data Structure: NATURAL NUMBERS

Base	`0`
Recognizer	`numberp[n]`
Constructor	`+1[n]`
Accessor	`-1[n]`

Some invariants

```
numberp[n] or not[numberp[n]]
numberp[+1[n]]
numberp[0]
not[+1[n] = 0]

(numberp[n] and not[n=0]) implies (+1[-1[n]] = n)
numberp[n] implies (-1[+1[n]] = n)
```

Induction

```
if F[0] and (F[n] implies F[+1[n]]) then F[n]
```

Abstract Data Structure: BINARY TREES

Predicates

```
atom[x]
tree[x]
```

Constructor

```
+[x,y]
```

Uniqueness

```
not[atom[+[x,y]]]
if (+[x1,x2] = +[y1,y2]) then (x1=y1 and x2=y2)
```

Left and Right

```
left[+[x,y]] = x
right[+[x,y]] = y
```

Decomposition

```
if not[atom[x]] then x = +[left[x],right[x]]
```

Induction

```
if F[atom] and
   (if F[x1] and F[x2] then F[+[x1,x2]])
then F[x]
```

Some recursive *binary tree functions*

```
size[x] =def=     size[atom[x]] = 1;
                  size[+[x,y]] = size[x] + size[y] + 1

leaves[x] =def=   leaves[atom[x]] = 1;
                  leaves[+[x,y]] = leaves[x] + leaves[y]

depth[x] =def=    depth[atom[x]] = 1;
                  depth[+[x,y]] = max[depth[x],depth[y]] + 1
```

Pseudocode:

```
leaves[x] =def=   if empty[x] then 0
                    else if atom[x] then 1
                    else leaves[left[x]] + leaves[right[x]]

leaves-acc[x,res] =def=
      if empty[x] then res
        else if atom[x] then leaves-acc[(), res + 1]
        else leaves-acc[right[x], res + leaves-acc[left[x]]]
```

Abstract Domain Theory: STRINGS

Here is the **Theory of Strings** as a complete example. Note that the **Theory of Sequences** and the **Theory of Non-Embedded Lists** are almost identical.

Constants: `{E}` — the Empty string

Variables (typed):
`{u,v,...}` — characters
`{x,y,z,...}` — strings

Functions: `{·, head, tail, *, rev, rev-acc, butlast, last}`

· is prefix, attach a character to the front of a string
* is concatenate, attach a string to the front of another string
[the rest are defined below as special functions]

Relations: `{isString, isChar, isEmpty, =}`

`isEmpty[x]` — test for the empty string
`isChar[x]` — test for valid character
`isString[x]` — test for valid string

Generator Facts:

```
isString[E]
isString[u]
isString[u·x]
```

Uniqueness:

```
not(u·x = E)
if (u·x = v·y) then u=v and x=y
```

Special char axiom:

```
u·E = u
E·u = u
```

Decomposition:

```
if not(x=E) then (x = u·y)
head[u·x] = u
tail[u·x] = x
if not(x=E) then (x = head[x]·tail[x])
```

Decompose equality:

```
if (u=v) then (u·x = v·x)
if (x=y) then (u·x = u·y)
```

Mapping:

```
F[u·x] = F[u]·F[x]
```

The String ***Induction*** *Principle:*

```
if F[E] and
     forall x:   if not[x=E],
                 then if F[tail[x]] then F[x]
then  forall x:  F[x]
```

Recursion, mapping:

`F[E]`	base
`F[u·x] = F[u]·F[x]`	general1
`F[x] = F[head[x]]·F[tail[x]]`	general2

Pseudo-code for testing *string equality*, using the Induction and Recursion templates for binary relations

```
if =[E,E] and
   forall x,y:
      if (not[x=E] and not[y=E]),
        then if (=[head[x],head[y]] and =[tail[x],tail[y]])
               then =[x,y]
  then forall x,y:      =[x,y]
```

`=[E,E]`	base
`=[x,y] = =[head[x],head[y]] and =[tail[x],tail[y]]`	general1

```
=[a,b]  =def=
      (a=E and b=E)
   or (=[head[a],head[b]] and =[tail[a],tail[b])
```

Some *axioms and theorems* for specialized functions

Concatenate, `*`, for joining strings together:

`E*x = x,    x*E = x`	base definition
`(u·x)*y  =  u·(x*y)`	recursive definition
`isString[x*y]`	type
`u*x = u·x`	character special
`x*(y*z) = (x*y)*z`	associativity
`if x*y = E, then x=E and y=E`	empty string

`if not(x=E) then head[x*y] = head[x]`	head
`if not(x=E) then tail[x*y] = tail[x]*y`	tail

Reverse, rev, for turning strings around:

`rev[E] = E`	base definition
`rev[u·x] = rev[x]*u`	recursive definition
`isString[rev[x]]`	type
`rev[u] = u`	character special
`rev[x*y] = rev[y]*rev[x]`	concatenation
`rev[rev[x]] = x`	double reverse
`rev[x*u] = u·rev[x]`	suffix

Reverse-accumulate, reverse the tail and prefix the head onto the accumulator:

`rev-acc[x,E] = rev[x]`	identicality
`rev-acc[E,x] = x`	base definition
`rev-acc[u·x,y] = rev-acc[x,u·y]`	recursive definition

Last and *Butlast*, for symmetrical processing of the end of a string:

`butlast[x*u] = x`	definition
`last[x*u] = u`	definition
`if not(x=E) then isString[butlast[x]]`	type
`if not(x=E) then char[last[x]]`	type
`if not(x=E) then x = butlast[x]*last[x]`	decomposition
`if not(x=E) then butlast[x] = rev[tail[rev[x]]]`	tail reverse
`if not(x=E) then last[x] = head[rev[x]]`	head reverse

Here is a function which mixes two domains, Strings and Integers:

Length, for counting the number of characters in a string

```
length[E] = 0

length[u·x] = length[x] + 1

length[x*y] = length[x] + length[y]
```

A symbolic proof by induction

To prove:	`rev[rev[x]] = x`	x is of type STRING
Base case:		Rule applied:
	`rev[rev[E]] =?= E`	1. problem
	`rev[E] =?= E`	2. rev[E] = E
	`E =?= E`	3. rev[E] = E, identity QED
Inductive case:		
	`rev[rev[x]] =?= x`	1. problem
	`rev[rev[u·x]] = u·x`	2. assume by induction rule
	`rev[rev[x]*u] = u·x`	3. rev[a•b] = rev[b]*a
	`rev[u]*rev[rev[x]] = u·x`	4. rev[a*b] = rev[b]*rev[a]
	`u*rev[rev[x]] = u·x`	5. rev[a] = a a is a char
	`u·rev[rev[x]] = u·x`	6. lemma a*b=a•b a is a char
	`rev[rev[x]] = x`	7. a•b = a•c iff b=c QED
Lemma:		
	`u*x =?= u·x`	1. problem
	`(u·x)*y = u·(x*y)`	2. prefix/concatenate distribution
	`(u·E)*y = u·(E*y)`	3. let x=E
	`u*y = u·(E*y)`	4. a•E = a
	`u*y = u·y`	5. E*a = a QED