Abstract data Types and "Objects" in Aldat

T. H. Merrett*

McGill University

May 9, 2008

[1]

These notes give an example for Aldat constructs that we have developed earlier (see [Mer99, "Database programming"]). Specifically, an abstract data type (ADT) is a computation which returns computations among its (public) parameters, following [AM84], and an object is an instantiated ADT-with-a-state.

1. Abstract Data Type. We use vectors of Booleans for our example. We represent these as relations, and they may be nested within other relations. Here is such a nested relation.

```
domain pred strg;
domain val bool;
domain lightNdark(pred,val);
domain subject strg;
relation analogy(subject,lightNdark) <- { ("X",("F",f),("A",f),("H",f),("D",f)),
                                          ("Y",("F",t),("A",t),("H",t),("D",f)),
                                          ("Z",("F",t),("A",f),("H",t),("D",t))
                                        };
```

These declarations and initialization produce

The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

analogy		
(*subject*	*lightNdark*	)
	(*pred*	*val*)
X	F	**false**
	A	**false**
	H	**false**
	D	**false**
Y	F	**true**
	A	**true**
	H	**true**
	D	**false**
Z	F	**true**
	A	**false**
	H	**true**
	D	**true**

We see that this represents three vectors on the set of indices `F,A,H,D`, namely (written as bits)

X: `0000`
Y: `1110`
Z: `1011`

2. We would like to be able to do Boolean operations on these bit vectors without having to write each operation each time fully out in terms of Boolean operations of the individual elements.

Each operation can be written as a computation which can be defined once for all and then invoked. For example

comp *notV*(x, y) **is**
{ **let** *val′* **be not** *val*;
  let *val* **be** *val′*;
  y <− [*pred,val*] **in** [*pred,val′*] **in** *x*;
} **alt**
{ **let** *val′* **be not** *val*;
  let *val* **be** *val′*;
  x <− [*pred,val*] **in** [*pred,val′*] **in** *y*;
};

(where we include the **alt**ernative because **not** is symmetrical).

A binary operation is

comp *andV*(x, y, z) **is**
{ **let** *val′* **be** *val*;
  let *val″* **be** *val* **and** *val′*;
  let *val* **be** *val″*;
  z <− [*pred,val*] **in** [*pred,val″*] **in** (*x* **ujoin** [*pred,val′*] **in** *y*)
};

(where there is no **alt**ernative because **and** has no inverse).

Other binary operations are similar: *orV*, *xorV*, etc. (Discuss the **alt**ernatives for *orV* and *xorV*.)

3. Going further, we should package all these operations together. This brings us to the ADT.

comp *vecBool*(*notV*,*andV*,*orV*,*xorV*,*nxorV*) **is**
{ **comp** *notV*(x, y) **is**
  :
  comp *andV*(x, y, z) **is redop**
  :
  comp *orV*(x, y, z) **is redop**
  :

comp *xorV*(x, y, z) **is redop**
:
comp *nxorV*(x, y, z) **is redop**
:
};

This computation returns only other ("first class") computations, which provide the "methods" (in object-oriented terminology).

We have specified **redop** for the four binary operators that are associative and commutative, so that they may be used in **red**uction and **equiv**alence reduction in the domain algebra.

4. Here is an application. First we invoke *vecBool* to make the methods available.

comp *vecBool*(**out** *notV*, **out** *andV*, **out** *orV*, **out** *xorV*, **out** *nxorV*);

Then we use the methods—here, in **red**uctions.

let *allV* **be red** *andV* **of** *lightNdark*;
universal <− [**red ujoin of** *allV*] **in** *analogy*;
let *analogV* **be red** *xorV* **of** *lightNdark*;
conclusion <− [**red ujoin of** *analogV*] **in** *analogy*;

These give

universal

(*pred*	*val*)
F	**false**
A	**false**
H	**false**
D	**false**

conclusion

(*pred*	*val*)
F	**false**
A	**true**
H	**false**
D	**false**

(We used **red ujoin** to raise the level of the answers so that they are no longer nested.)

Universal is all false because **red and** means "for all".

Conclusion has an interesting interpretation in terms of analogical reasoning [Kle85]. Consider `F` to mean "female", `A` "adult", `H` "hates" and `D` "dark" (the complements being, of course, "male", "child", "loves" and "light", respectively). Then

`X` is "Boy loves light",
`Y` is "Woman hates light" and
`Z` is "Girl hates dark".

Pause for a second to think what the conclusion, `?` is if we made the analogy

`X` is to `Y` as `Z` is to `?`

Now interpret *conclusion*, above and see if you get the same answer.

5. "Objects". We have earlier (see [Mer99, "Database programming"]) given good reasons for "object-orientation", such as counters or stopwatches, which need a state and hence instantiation (e.g., we need two different counters to count sheep and goats). Here we illustrate the ridiculous extreme to which object-orientation is sometimes taken.

A class is an ADT with state, and an object is an instance of a class. We are going to incorporate the Boolean vector as the state of our ADT (modified from the above). This will mean that to **and** one vector with another, we "send a message", containing one vector, to the other with the instruction to perform *andV*.

Here is the ADT with state, showing the new implementations for *notV* and *andV*.

comp *vecBOOl*(*X,notV,andV,orV,xorV,nxorV*) **is**
{ **comp** *notV*(x) **is**
{ **let** *val′* **be not** *val*;
let *val* **be** *val′*;
x <− [*pred,val*] **in** [*pred,val′*] **in** *vec*
};

```
comp andV(x, y) is
{ let val′ be val;
  let val″ be val and val′;
  let val be val″;
  y <− [pred,val] in [pred,val″] in (x ujoin [pred,val′] in vec)
};
:
vec <−X;                << constructor: create "object"(s) on invocation>>
};
```

Note that *notV* has no **alt**ernative bacause it no longer has the symmetry to support an inverse, and *andV* cannot be a **redop** since *andV* is no longer a binary operator. Similar restrictions apply to the other operators.

6. This object-oriented variant no longer supports **red**uction, so we need a different application. We'll limit this to *notV*, although a slightly more elaborate example can illustrate the (formerly) binary operators.

First, we instantiate *vecBOOl* by joining it with a relation that contains some vectors, e.g., *analogy*.

```
objectSet <− analogy [lightNdark: ijoin :X] vecBOOl;
```

Then we can use the "method".

```
let vectN be notV[];
negAnal <− [subject, vectN] in objectSet;
```

This gives

negAnal

(*subject*	*vectN* (*pred*	*val*))
X	F	**true**
	A	**true**
	H	**true**
	D	**true**
Y	F	**false**
	A	**false**
	H	**false**
	D	**true**
Z	F	**false**
	A	**true**
	H	**false**
	D	**false**

Of course, it is a clunky way to use *notV*, with the argument implicit in the instantiated tuples. Using *andV*, supposing another vector attribute, say *vect*, were available as a result of joining *objectSet* with some other relation, is just as painful.

```
let vectA be andV[vect];
```

etc.

Attempts such as this to push object-orientation too far are no doubt what have led to the invention of the oxymoron "stateless object", which just reverts to ADTs to solve some problems; and many other aspects.

References

[AM84] M. P. Atkinson and R. Morrison. *Persistent First Class Procedures are Enough*, volume 181 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1984.

[Kle85] Sheldon Klein. The invention of computationally plausible knowledge systems in the upper paleolithic. Technical Report 628, Computer Sciences, University of Wisconsin-Madison, Madison, WI, Dec. 1985. Presented at The World Archeological Congress, Southampton and London, 1–7 Sept. 1986, Allen and Unwin.

[Mer84] T. H. Merrett. *Relational Information Systems.* Reston Publishing Co., Reston, VA., 1984.

[Mer99] T. H. Merrett. Relational information systems. (revisions of [Mer84]):
Data structures for secondary storage: http://www.cs.mcgill.ca/∼tim/cs420
Database programming: http://www.cs.mcgill.ca/∼tim/cs612, 1999.