Adtpp: algebraic data types for C — lightweight, efficient, safe, polymorphic, higher order

Lee Naish
Peter Schachte
Aleck MacNally
Computing and Information Systems
University of Melbourne

https://lee-naish.github.io/papers/adtpp/

Outline

Motivation

- Expressiveness and safety of ADTs in C
- Polymorphic types and functions
- Efficiency of C
- Lightweight implementation
- Share user-defined data types across language boundaries (eg, the Pawns language)

Defining algebraic data types (ADTs)

ADTs values are one of a number of defined *data constructors* (a “sum”) with a number of *arguments* of defined types (a “product”)

Supported first in Hope and now in many other functional languages

`Adtpp` takes a file with a “`.adt`” extension and creates a “`.h`” file

```
data point { Point(double, double); }
data color { Red(); Blue(); Green(); }
data tree {
    Empty();
    Node(long, tree, tree); }
data quad_roots {
    Noroot();
    Oneroot(double);
    Tworoot(double, double);
}
```

Using ADTs in C code

Each ADT defined leads to a C type of that name

ADT values are created using the data constructors (defined as C macros/functions); this may allocate memory

There is a “free” macro/function for each ADT defined which reclaims memory allocated for the top level data constructor

```
#include "mytypes.h" // generated by "adtpp mytypes.adt"
...
    tree t1, t2, t3;              // declare vars with ADTs
    t1 = Empty();                 // create values with ADTs
    t2 = Node(1L, t1, Empty());
    t3 = Node(2L, t2, t2);
...
    free_tree(t3);                // free memory for t3 only
```

Using ADTs in C code (cont.)

ADT values are tested and deconstructed (pattern-matched) using if-then-else (and switch) macros:

```
if_Node(t3, val, tl, tr)
    ...
else()
    ...
end_if()
```

```
if_Empty(t3)
    ...
else_if_Node(val, tl, tr)
    ...
end_if()
```

There are `if_C` and `else_if_C` macros for each data constructor `C`

There are also `if_C_ptr` and `else_if_C_ptr` macros which initialize pointers to arguments, so they can be updated

Using ADTs in C code (cont.)

```
// return the sum of the elements in a tree
long sum_tree(tree t) {
    switch_tree(t)
    case_Empty()
        return 0;
    case_Node(val, tl, tr)
        return val + sum_tree(tl) + sum_tree(tr);
    end_switch()
}
```

If-then-else and switch constructs create new C blocks where the “pattern” variables are declared with the correct type and limited scope

Eg, in `sum_tree` we can only refer to `val`, `tl` and `tr` in a context where they exist (in regular C we can use `t->left` where `t` may be `NULL`)

Polymorphism and higher order

Languages with ADTs generally support polymorphism (eg, type "list of *t*", where *t* can be any type, and functions such as `reverse` that work for lists of any type)

`Adtpp` only produces a "`.h`" file—there is no analysis of "`.c`" files

Due to the limitations of the C type system, in `adtpp` we must declare all instances of polymorphic types and functions used in the program (distinct identifiers are used for different types etc in the C code)

Typical functional languages also support functions being passed to and returned from functions, and used in data structures

C has "pointers to functions" but to support polymorphic higher order code, `adtpp` needs explicit support for higher order

Polymorphism

Type parameters are enclosed in “angle brackets” and instances of polymorphic types can be declared:

```
data pair<t1, t2> { Pair(t1, t2); }
data list<t> {
    Nil();
    Cons(t, list<t>);
}

type points = list<point>;
type colors = list<color>;
type ints = list<adt_int>;
type polygon = pair<color, points>;
type polygons = list<polygon>;
type pairs<t1, t2> = list<pair<t1, t2>>;
type polygons1 = pairs<color, points>;
```

Polymorphism (cont)

Instances have the type name appended to data constructor names, like

```
data ints {       // implicitly generated by adtpp
    Nil_ints();
    Cons_ints(adt_int, ints);
}
```

So we sum a list of integers with the following code:

```
int sumlist(ints xs) {
    if_Cons_ints(xs, head, tail)
        return head + sumlist(tail);
    else()
        return 0;
    end_if()
}
```

Polymorphism (cont)

Polymorphic functions and their instances must be declared

```
function<t> int length(list<t>);       // in ".adt" file
instance num_points = length<point>;
...
int length(list xs) {                  // in ".c file"
    int len = 0;
    while (1) {
        if_Cons(xs, head, tail)
            len++;
            xs = tail;
        else()
            return len;
        end_if()
   }
}
```

Polymorphism (cont)

```
function<t> list<t> concat(list<t>, list<t>); // ".adt" file
instance join = concat<point>;
instance concat_col = concat<color>;
...
list concat(list xs, list ys) {                 // ".c file"
    if_Cons(xs, head, tail)
        return Cons(head, concat(tail, ys));
    else()
        return ys;
    end_if()
}
...
    pts3 = join(pts1, pts2);
    cols3 = concat_col(cols1, cols2);
```

Multiple type parameters/generic types

```
// takes Pair(x,y) and returns Pair(y,x)
pair_swp swap(pair xy) {
    if_Pair(xy, x, y)
        return Pair_pair_swp(y, x);
    end_if()
}
...
function<t1, t2> pair<t2, t1> swap(pair<t1, t2>);
instance swap_polygon = swap<points, color>;

type polygon_swp = pair<points, color>;
type pair_swp = pair<adt_2, adt_1>;     // Needed for swap
```

adt_1, adt_2 etc are generic types, like type variables

```
type pair = pair<adt_1, adt_2>;     // implicitly generated
```

Higher order

Consider using an analogue of the Haskell function
```
zipWith :: (t1 -> t2 -> t3) -> [t1] -> [t2] -> [t3]
```
to take a list of colors + a list of lists of points & create a list of polygons

```
function<t1,t2,t3>
    list<t3> zipWith(t3 func(t1,t2), list<t1>, list<t2>);
instance mk_polygons = zipWith<color, points, polygon>;

type pointss = list<points>;
type list_2 = list<adt_2>;
type list_3 = list<adt_3>;
```

Note there are seven distinct list types to potentially confuse, eg
```
polys = (polygons) zipWith(((void*)(*)(void*,void*)) &Pair,
                                  (list) cols, (list) ptss);
```

Higher order

```
list_3 zipWith(adt_3 (*f)(adt_1, adt_2), list l1, list_2 l2){
    if_Cons(l1, hd1, tl1)
        if_Cons_list_2(l2, hd2, tl2)
            return Cons_list_3((*f)(hd1,hd2),
                                   zipWith(f,tl1,tl2));
        else()
            return Nil_list_3();
        end_if();
    else()
        return Nil_list_3();
    end_if()
}
    ...
    polys = mk_polygons(&Pair_polygon, cols, ptss);
```

Implementation

The standard portable way to implement ADTs in C is with a struct containing a tag and a union of structs (used in similar `adt` tool)

`Adtpp` uses tagged pointers—with 64 bit words and byte addressing we can use 3 tag bits (up to 8 data constructors with no extra space)

Constants are represented as small integers (so `Nil == Empty == NULL`)

Each data constructor with arguments is implemented using a struct (with a tag if needed) and each value is a pointer to a dummy struct; it is cast as required

```
typedef struct _ADT_quad_roots {} *quad_roots;
struct _ADT_quad_roots_Oneroot {double f0;};
struct _ADT_quad_roots_Tworoot {double f0; double f1;};
```

Implementation (cont.)

Data constructors and polymorphic function instances use inline functions

```
static __inline quad_roots Tworoot(double v0, double v1){
    struct _ADT_quad_roots_Tworoot *v =
        (struct _ADT_quad_roots_Tworoot*)
        ADT_MALLOC(sizeof(struct _ADT_quad_roots_Tworoot));
    v->f0=v0;
    v->f1=v1;
    return (quad_roots)(1+(uintptr_t)v);
}
static __inline polygons mk_polygons(
        polygon (*v0)(color, points), colors v1, pointss v2){
    return (polygons) zipWith((adt_3 (*)(adt_1, adt_2)) v0,
                              (list) v1, (list_2) v2);
}
```

Implementation (cont.)

If-then-else and switch constructs use macros

```
#define if_Tworoot(v, v0, v1) \
    {quad_roots _ADT_v=(v); \
    if ((uintptr_t)(_ADT_v) >= 1 && \
            ((uintptr_t)(_ADT_v)&ADT_LOW_BITS)==1) { \
        double v0=((struct _ADT_quad_roots_Tworoot*) \
            ((uintptr_t)_ADT_v-1))->f0; \
        double v1=((struct _ADT_quad_roots_Tworoot*) \
            ((uintptr_t)_ADT_v-1))->f1;
#define else_if_Noroot() \
    } else if (((uintptr_t)(_ADT_v))==0) {
#define else() } else {
#define end_if() }}
```

Performance (space)

	struct size			malloc size		
	Node	Empty	per key	Node	Empty	per key
adt	32	32	64	48	48	96
Regular C	24	0	24	32	0	32
adtpp	24	0	24	32	0	32

"Leaf optimized 234-tree" struct sizes (bytes)

	Two	Three	Four	TwoL	ThreeL	FourL	per key
% of nodes	22	15	3	20	25	15	
adt	64	64	64	64	64	64	40
Regular C	64	64	64	64	64	64	40
Red-Black tree	24	48	72	24	48	72	24
adtpp	24	40	56	8	16	24	13

Performance (time)

Summing elements in a tree with integers in internal nodes and leaves

Optimization	none	-O1	-O2	-O3
C, NULL base case	8.8	5.4	4.1	4.3
C, left==NULL base case	4.9	3.3	2.2	2.7
adt, if	18.2	16.0	9.3	3.2
adt, switch	10.4	8.4	4.4	2.4
adt `_FAST_`, if	4.8	3.6	2.6	2.8
adt `_FAST_`, switch	5.9	4.3	2.3	2.0
adtpp, if	7.6	4.0	2.8	3.0
adtpp, switch	7.6	4.6	2.2	2.0

Conclusion

Expressiveness and safety of ADTs is *great* - they should be supported well in many more languages and they can be supported reasonably in C without much effort

Polymorphism can be supported with a lightweight implementation - renaming (plus some hacks with multiple generic types) is a slightly inconvenient alternative having a full polymorphic type system in the language

Efficient implementation (with tagged pointers) is possible but only arguments of data constructors should be mutable and details of tags and pointers should be kept from the programmer