SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Adtpp: lightweight efficient safe polymorphic algebraic data types for C[*]

Lee Naish[1], Peter Schachte[1] and Aleck MacNally[1]

[1]*Computing and Information Systems, Universtity of Melbourne*

SUMMARY

`Adtpp` is an open-source tool that adds support for algebraic data types (ADTs) to the C programming language. ADTs allow more precise description of program types and more robust handling of data structures than is directly supported by C. ADT definitions and other declarations are put in a file that is preprocessed by `adtpp` to produce a C header (".h") file which can be included in C source files. The generated header file contains C type definitions, macros, and inline functions that support type-safe construction, deconstruction, and pattern matching of ADT values, while avoiding unsafe operations such as casts, and avoiding the risk of errors such as dereferencing NULL pointers and accessing inappropriate fields of unions. Values are represented efficiently, using techniques from the implementation of declarative languages. For many simple data types, the memory representation is identical to a direct implementation in C, with no loss of efficiency. For more complex types, the `adtpp` representation is more efficient than common C representations, while preserving type safety and convenience. As an example, we present a new variation of 234-trees which is very compact. `Adtpp` also supports parametric polymorphism such as defining a type "list of t", where t can be any ADT, and generic functions such as length. However, polymorphic code is somewhat more verbose than for typical declarative languages, due to our reliance on the limited type checking available in C. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Algebraic data types; C language extension; generics; polymorphism; safety; static checking; open-source

1. INTRODUCTION

Algebraic data types (ADTs[‡]) combine other types. They were first introduced in the Hope programming language [1] and have been adopted in many other declarative languages, such as ML [2], Haskell [3] and Mercury [4]. An ADT value consists of one of a number of alternative data constructors, each of which has a number of arguments or fields of particular types. The choice of data constructor gives a "sum" ("discriminated union", "disjoint union" or "variant type") and the arguments of the data constructor give a "product" ("tuple" or "record"). For example, an ADT representing a bank account may have credit or savings as the data constructors, where credit has arguments representing the card number, security code, credit limit and balance, and savings has arguments representing an account number, bank branch number and balance. C supports product types using structs. Undiscriminated unions are also supported, but discriminated unions (distinguishing between different data constructors) require extra programming, which can be error prone.

[*]to appear in *Software Practice and Experience*
[*]Correspondence to: E-mail: {lee,schachte}@unimelb.edu.au
[‡]This acronym is also used for *abstract* data types, unrelated to our use here.

Most languages that support ADTs also support polymorphism. By using a type variable t, the type list of t can be defined as Nil (representing an empty list) or Cons, with arguments of type t and list of t, representing the head and tail, respectively. This polymorphic type definition can then be instantiated to define lists of integers, lists of bank accounts, etc. The polymorphic type can also be used in polymorphic (generic) code, such as a length function that computes the length of a list of any type of element. The only support for polymorphism in C is the generic type pointer to void. Polymorphic code can be written and used by casting values to and from the type pointer to void, which is also error prone.

This paper describes the `adtpp` tool we have developed, which supports ADTs in C code. It is similar to the `adt` tool [5] but gives improved compile-time checking, more compact data representations and much more extensive support for polymorphism. Our goals include:

- **Primarily we want to obtain the expressiveness and type safety of ADTs in C.** Most notably, compared to the types supported in C, ADTs allow us to more precisely describe the values we want to represent. Having more precise types allows us to avoid some common programming errors. For example, the type system of C does nothing to ensure that a pointer is not NULL before allowing it to be dereferenced. Even when `listx` is NULL, representing the empty list, C allows evaluation of `listx->head`, leading to a fatal runtime error. For the `adtpp` version of lists, the programming constructs provided do not allow access to the head of a list in a context where it may be empty, because `Nil` has no argument representing the head. Similarly, in C a function can return a value plus an error flag, but C allows the value to be used even if the error flag is set. An ADT that represents either a value or an error would not permit such a mistake.
- **We want support for polymorphic types (and their instances) and functions.** This allows significant code re-use and abstraction. With polymorphic code it is particularly important that type safety is ensured at compile time.
- **We want to maintain the efficiency of C.** Our implementation imposes essentially no efficiency penalty compared with direct implementation in C. For example, a linked list expressed as an ADT in the natural way is represented in memory in exactly the same way as a linked list expressed in the most natural way using C types, with an empty list represented as NULL and a non-empty list represented as a pointer to a struct containing the head and tail. For more complex ADTs, the `adtpp` implementation not only provides type safety, it also allows more concise definition and manipulation of types than in plain C, and most surprisingly, tends to be *more* efficient, both in time and space, than a typical direct implementation in C.
- **We aim for a lightweight implementation, to improve portability, maintainence, and ease of integration.** All `adtpp` does is produce a C header (".h") file. It does not process C source files and there are no changes to the C compiler. This constraint does affect the ability to perform some checking, leading to somewhat more verbose code in the presence of polymorphism, and somewhat less informative error messages.
- **Finally, we would like to be able to share non-trivial user-defined data types across language boundaries.** Many declarative languages that support ADTs have interfaces to C that only support simple data types such as integers and floats. By ensuring `adtpp` uses the same representation as a particular declarative language, a C interface can allow other data types to be passed. This is done in the Pawns language [6], which also uses `adtpp` as an integral part of the implementation — it compiles to C using `adtpp`.

We have not studied the impact of using `adtpp` on large scale software production but Hartel and Muller [5] discuss the use of the (similar) `adt` tool and give examples where around 20% of the code for a project is produced by the tool. Although `adtpp` does not produce as many primitives for each ADT, the representation of values is often more complex and the better support for polymorphism helps with code re-use. Of course the savings will vary according to the data structures used in the project. Perhaps of more importance is the elimination of certain classes of errors, such as dereferencing NULL pointers, and `adtpp` does have advantages over `adt` in this respect. `Adtpp` also allows very simple integration with automatic memory management tools, avoiding another class of errors. Support for ADTs can also have a positive influence on the choice of data structures used in a project. For example, using a (dynamically allocated) ADT rather than an array avoids size limitations and index out of bounds errors. We believe `adtpp` can significantly increase productivity of C programmers who are familiar with ADTs and their benefits, while retaining excellent performance.

In what follows, we assume the reader is familiar with ADTs and their benefits. The rest of this paper is structured as follows. Section 2 gives an overview of `adtpp` and describes how monomorphic ADTs are defined. Section 3 describes how such types can be used in C code. Section 4 discusses support for polymorphism and higher order code ("pointers to functions" in C terminology). Section 5 describes the implementation of `adtpp`, including the representation of ADTs. Section 6 discusses space and time performance of code that uses `adtpp`. Section 7 discusses additional related work. Section 8 describes possible future extensions to `adtpp`. Section 9 concludes.

2. ADT DECLARATIONS

The `adtpp` tool inputs ADT definitions and other declarations from an "ADT file", whose name has an extension "`.adt`", and outputs a C header ("`.h`") file that contains declarations and definitions of ordinary C functions, macros and types (`struct` and `typedef`) that support the ADTs. Any C code that uses the ADTs need only `#include` the generated "`.h`" file, capitalising on C's type system to detect errors in ADT use. The `adtpp` tool does not process the C code at all, so it does not require a parser for C. The following command would create `mytypes.h` from `mytypes.adt` (appropriate generic rules for build tools such as `make` are easy to construct):

```
adtpp mytypes.adt
```

In this section we describe monomorphic ADT definitions. Section 4 describes polymorphism and the other declarations allowed in ADT files. ADT definitions provide the same information as in other languages such as Hope, ML and Haskell — the name of the type and, for each data constructor, the name and the number and types of arguments (each data constructor name can only appear in one type). However, the syntax is inspired by C. Braces and semi-colons are used, white-space is ignored and context (rather than upper/lower case) is used to distinguish types and data constructors. Here we use upper case for the first letter of data constructor names, but users can adopt their own conventions.

Figure 1 shows four ADT definitions. The first type, `point` can be used to represent the Cartesian coordinates of a point in two dimensions. It has a single data constructor, `Point` with two arguments, both of type `double`. This is a simple example of a product type that can just as easily be implemented with a C struct containing two `double`s. ADT definitions are prefixed with the `data` keyword and type and data constructor names must be valid C identifiers, and are case-sensitive. It is possible to use any C type as the type of an argument of a data constructor, but in general it requires the use of `typedef` since the C syntax for compound types such as arrays and structs are not supported by `adtpp`. For example, we could replace `double` by `mytype` in the definition and `adtpp` would process the definition in the same way. Of course `mytype` must then be defined appropriately using `typedef` (or `#define`) before the generated "`.h`" file is included.

The type `color` is a sum type with three alternative data constructors and can easily be implemented in C with an `enum`. Type `tree` is a sum of products so representation is more complex. However, because there are only two data constructors and only one has arguments, it can be implemented in C using a *pointer* to a struct containing a `long` and two `trees` (representing a `Node`). `Empty` can be represented as `NULL`, a special value that can be distinguished from other pointer values in C. Type `quad_roots` also takes the general form of a sum of products but cannot so easily be represented in C for two reasons: there are more than two data constructors (so we can't determine the data constructor by simply comparing with `NULL`) and there is more than one data constructor with arguments (`NULL` pointers cannot contain other values). We discuss how it can be represented in C in Section 5.

```
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);
}
```

Figure 1. Monomorphic ADT definitions in `adtpp` syntax

3. USING ADTS IN C CODE

Each ADT defined in the ADT file results in a C type of the same name defined (using `typedef`) in the "`.h`" file produced. The types are intended to be abstract in the sense that the programmer does not need to know how they are represented, but just has access to certain operations on the types. The operations either construct a value of the type (that may allocate memory), free the memory used by the value of a type or deconstruct (and/or test) a value of the type. These are described next.

3.1. Constructing ADT values

The only way to construct an ADT value is via the data constructors of the type. These are used as C functions/macros in C source code. Appropriate arguments with correct types must be supplied. Figure 2 gives an example of declaring and constructing values of type `tree`. Memory is allocated using `malloc` by default, but this can be overridden by defining the macro `ADT_MALLOC` to be an alternative function with equivalent behaviour.

```
#include "mytypes.h" // generated by adtpp from mytypes.adt
...
    tree t1, t2, t3;
    t1 = Empty();
    t2 = Node(1L, t1, Empty());
    t3 = Node(2L, t2, t2);
```

Figure 2. Declaring and constructing values of type `tree` in C code

3.2. Freeing ADT values

For each ADT defined there is a free function whose name is the type name with "`_free`" appended, which (by default) calls `free` as required to reclaim memory. The free function does not recursively free values within data constructors. Thus a call such as `tree_free(t3)` at the end of the code in Figure 2 would free the memory for the top level `Node` of the tree but not affect the memory allocated for `t2`.

An attractive alternative to explicitly coded memory management is to use some form of automatic memory management. `Adtpp` can be used very simply with the Boehm-Weiser conservative garbage collector [7]. All that is required is a compilation flag to link the appropriate library and the following definitions before the header file is included:

```
#define ADT_MALLOC(s) GC_MALLOC(s)
#define ADT_FREE(s) GC_FREE(s)
```

With this option the memory taken by ADT values that are no longer used is reclaimed automatically and calling the "`_free`" functions is not required. This simplifies coding and greatly reduces the chance of bugs related to memory management, including memory leaks. Our small-scale experiments indicate it can improve performance as well.

3.3. Testing and deconstructing ADT values

In many declarative languages pattern matching is used to both test if a value has a particular data constructor and extract the argument values. Successful pattern matching results in the argument values being used to initialize local variables that can be used in a section of code that is executed, whereas failure of pattern matching results in the transfer of control to other code where those variables are undefined. Similar constructs are supported in `adtpp`. One class is based on if-then-else, which can be the most convenient if there are few data constructors being tested for. The other is based on switch, and can be more efficient and convenient when there are larger numbers of data constructors tested. Both force the programmer to write code that is naturally safe. For example, the left subtree of a tree can only be accessed by using a variable in a `Node` pattern that is matched with the subtree. In C, if `t3` is a tree represented by a pointer to a struct, `t3->left` is potentially unsafe because `t3` may be `NULL`.

Figure 3 shows two uses of if-then-else constructs for the `tree` type. The `if_Node` primitive tests if `t3` is a `Node`. If the match succeeds, the "if" branch is executed, otherwise the "else" branch is executed. Before executing the "if" branch, `val`, `tl` and `tr` are initialized to the three `Node` arguments, respectively. These three variables are automatically declared just for the "if" branch and have type `long`, `tree` and `tree`, respectively. The alternative coding is slightly more verbose (and potentially a little less efficient). It first tests for `Empty` and uses `else_if_Node` to extract the arguments (note that `t3` is not used here — `else_if` primitives test the same variable as the initial `if` primitive). There are also `if_Node_ptr` and `else_if_Node_ptr` primitives that bind the last three arguments to pointers to the respective arguments of a `Node`, allowing update of these components.

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

Figure 3. Two uses of `adtpp` if-then-else constructs in C code

`Adtpp` generates `if` and `else_if` primitives for each data constructor of each ADT. The syntax supported is less C-like than we would wish for, but is the best we have been able to achieve with the standard C preprocessor and the goal of forcing the user to write safe code. We discuss this further in Section 5. The switch primitive is illustrated in Figure 4, which gives a function that returns the sum of the elements in a tree. Pointers to arguments of a `Node` can be obtained by using `case_Node_ptr` instead of `case_Node`, and `default()` can be used as a default label. As with the if-then-else primitives, `val`, `tl` and `tr` are automatically declared (with limited scope) and initialized. The type of each case must match the type of the switch. In Section 7 we compare this to similar code supported by the `adt` tool.

```
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()
}
```

Figure 4. Summing tree elements using `adtpp` switch

4. POLYMORPHISM AND HIGHER ORDER FUNCTIONS

Typical declarative programming languages that support ADTs also support parametric polymorphism — definitions of generic types such as "list of t", where t can be instantiated to any type, and generic functions such as reversing a list of t. There can also be multiple type parameters, for example, keys of type k and values of type v. In regular C, this style of code typically requires use of pointers to void and casts (which are unsafe). Declarative languages also support higher order programming — functions can be incorporated into data structures, passed as arguments and returned as results. In C we normally refer to "pointers to functions" rather than functions, which is a lower level view (the way such functions are represented is with pointers). `Adtpp` supports parametric polymorphism and some additional syntactic sugar to support higher order programming. We discuss these features next.

4.1. Polymorphism

To keep `adtpp` lightweight and unobtrusive, it does not access the C source file, and must rely on C's monomorphic type system for all its type checking. Therefore, each instance of a polymorphic type that is used must be explicitly declared in the ADT file, allowing `adtpp` to map it to a separate (monomorphic) C type. While this aspect of `adtpp` is not as elegant as typical declarative languages, it can be helpful to name compound types anyway — using `points` rather than `list(point)` throughout the code can make the code more concise, for example. In addition, if a polymorphic function is defined in the C code, it must be declared in the ADT file (this is used to generate the appropriate C function prototype) along with each instance used. Thus there are four kinds of definitions/declarations supported in the ADT file. Polymorphic ADT definitions have the same syntax as monomorphic definitions except that type names have additional parameters enclosed in angle brackets (based on the template syntax of C++). Polymorphic types and type parameters can be used in place of types in the body of the definition.

Figure 5 shows several polymorphic ADT definitions that are analogous to commonly used pre-defined types in Haskell and other declarative languages. The last defines the type list of `t` as either the empty list, `Nil`, or `Cons` with two arguments of type `t` and list of `t`, respectively. The analogous type in C is a pointer to a struct containing a pointer to void and a "next" pointer to the same type of struct. Polymorphic type definitions lead to the same functions/macros as monomorphic definitions. These can be used to define generic functions (see Figure 9 later, for example).

New polymorphic types are defined by "data" declarations and their instances are defined by "type" declarations; Figure 6 shows several examples. Each instance of each polymorphic type used in the program must be declared and given a separate name using a type declaration. `Adtpp` generates a definition of a C type of that name, as well as the associated macros/functions. This is needed to ensure that, for example, a list of points is never used where a list of colors is expected. Mapping these two polymorphic type instances to distinct C types allows the C type checker to detect such errors. The type parameters used to define such instances must be algebraic data types, or at least have the same size as a pointer (currently this is not checked by `adtpp`). There are some predefined types in `adtpp`, such as `adt_int`, which is an integer type of the correct size. The

```
data pair<t1, t2> { // simple product type, like a struct
    Pair(t1, t2);
}
data either<t1, t2> { // simple sum type, *discriminated* union
    Left(t1);
    Right(t2);
}
data maybe<t> { // like a C pointer (possibly NULL)
    Nothing();
    Just(t);
}
data list<t> { // generic list, like a C pointer to a struct
    Nil();
    Cons(t, list<t>);
}
```

Figure 5. Polymorphic ADT definitions in `adtpp` syntax

```
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>;
```

Figure 6. Defining names for instances of polymorphic ADTs

names of data constructors for the new type are the same as for the polymorphic type, but with an underscore and the new type name appended. For example, the `ints` type has data constructors `Nil_ints` and `Cons_ints`. Internally, `adtpp` generates the following data declararation:

```
data ints {
    Nil_ints();
    Cons_ints(adt_int, ints);
}
```

Note that code that uses monomorphic instances of polymorphic types must explicitly use these monomorphic data constructor names both for constructing and deconstructing. Therefore, while the structure of code to construct and deconstruct values remains the same, the constructor names are more verbose, as can be seen by comparing the coding of `sumlist` in Figure 7 with that of `concat` in Figure 9.

```
// returns the sum of list of integers xs
int sumlist(ints xs) {
    if_Cons_ints(xs, head, tail)
        return head + sumlist(tail)
    else()
        return 0;
    end_if()
}
```

Figure 7. Deconstruction of a monomorphic instance of a polymorphic type

Type declarations can also define new polymorphic types in terms of types defined elsewhere. For example, Figure 6 defines a polymorphic "list of pairs" type, `pairs`, and an instance of this type, `polygons1`, which is equivalent to type `polygons`.

```
function<t> int length(list<t>);
instance num_colors = length<color>;
instance num_polygons = length<polygon>;
instance num_points = length<point>;

function<t> list<t> concat(list<t>, list<t>);
instance join = concat<point>;
```

Figure 8. Declaring polymorphic functions and their instances

```
// returns length of (generic) list xs; iterative coding
int length(list xs) {
    int len = 0;
    while (1) {
        if_Cons(xs, head, tail)
            len++;
            xs = tail;
        else()
            return len;
        end_if()
    }
}

// returns concatenation of (generic) lists xs and ys,
// result shares ys; coded in recursive pure functional style
list concat(list xs, list ys) {
    if_Cons(xs, head, tail)
        return Cons(head, concat(tail, ys));
    else()
        return ys;
    end_if()
}
```

Figure 9. Defining polymorphic functions

Figure 8 shows declaration of polymorphic functions and instances of those functions. It declares `length` to be a generic function with a single type parameter, `t`. It takes a list of any type `t` and returns an int. An instance of this function, `num_colors`, is declared, which takes a list of colors. The header file generated by `adtpp` includes a function prototype for the generic length function and a definition of `num_colors` that calls `length` in a way that ensures safety. The `concat` function takes two lists and returns a list, with the same type of element for each of the lists, and `join` is an instance where the element type for each list is `point`.

Figure 9 shows how polymorphic functions `length` and `concat` can be defined using the macros/functions generated by `adtpp` for the generic types. To code `concat` in regular C we would normally have a generic list of pointers to void. To concatenate two lists of (pointers to) points we would cast both to lists of pointer to void and cast the result of `concat` to a list of (pointers to) points. All three casts can hide possible type errors that would not be caught by the C compiler. Calling `join` has the same effect as calling `concat` with appropriate casts, but `adtpp` ensures that any type errors can be picked up by the C compiler and any casts are encapsulated in code generated by adtpp and are safe.

4.2. Multiple type parameters

In the previous section we avoided some details that can become important when polymorphic types or functions have more than one type parameter. We discuss them now.

```
// takes Pair(x,y) and returns Pair(y,x)
swap(pair xy) {
    if_Pair(xy, x, y)
        return Pair(y, x);
    end_if()
}
```

Figure 10. Definition of swap the function for pairs

Consider the definition of the polymorphic swap function given in Figure 10. It takes a value Pair(x,y) and returns Pair(y,x). Although the argument and result are both generic pairs, they are different types — one is type pair<t1,t2> and the other is type pair<t2,t1>. If a polygon was passed to swap, a different type would be returned. Adtpp has a class of builtin generic types with no constructors that can be used as type variables. They are named adt_1, adt_2, *etc.* For type pair, the arguments of the Pair data constructor (the types of x and y in swap) are of type adt_1 and adt_2, respectively. To obtain the generic version of each polymorphic type adtpp instantiates the type with adt_1, adt_2, *etc.* This is equivalent to automatically supplying a type declaration defining this instance. If the generic types occur as parameters but in a different order, an explicit type declaration must be given.

```
type polygon_swp = pair<points, color>;
type pair_swp = pair<adt_2, adt_1>;

function<t1, t2> pair<t2, t1> swap(pair<t1, t2>);
instance swap_polygon = swap<points, color>;
```

Figure 11. Declarations for the swap function

Figure 11 gives appropriate type, function and instance declarations for swap. The generic types must be used to explicitly define pair_swp, the swapped version of type pair. Adtpp processes the function declaration for swap to produce a function prototype that takes a pair and returns a pair_swp. Its instance swap_polygon takes a polygon and returns the swapped version, polygon_swp, which must also be declared as an instance of pair.

4.3. Higher order functions

Adtpp can support any C type as an argument of a data constructor or polymorphic function if a suitable typedef is supplied. This includes "pointer to" function types. However, typedef cannot be used to define polymorphic types, so adtpp also has some direct support for higher order programming. For function (and other) declarations, arguments and results of functions can be declared as function types that may contain type parameters. We use a slightly different syntax to C: rather than "(*)" we use the keyword func. Thus a ("pointer to" a) function that takes two ints and returns a double would be written double func(int,int) in an ADT file rather than the C equivalent, double (*)(int,int).

Figures 12 and 13 give appropriate declarations, definition and a use of a version of the Haskell higher order polymorphic zipWith function. It takes two lists, with elements of type t1 and

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

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

Figure 12. Declarations for zipWith

```
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);
```

Figure 13. Definition and use of zipWith

t2, respectively, and produces a list with elements of type t3. It uses a function that takes two arguments, of type t1 and t2, respectively, and returns a value of type t3, and applies this function pair-wise to the elements of the input lists. For example, given a list of colors and list of lists of points, they can be combined to form pairs of elements that represent polygons. It is necessary to declare two aditional generic list instances and an instance for lists of lists of points. Furthermore, zipWith must be declared as a polymorphic function (with a polymorphic function as an argument — this is where the adtpp syntax for function types must be used). A function instance, mk_polygons, must be declared for creating lists of polygons. The function that creates a polygon (the argument of mk_polygons) is simply Pair_polygon, the instance of the Pair data constructor generated when the polygon type was declared as an instance of the pair type.

The code is somewhat more verbose and less elegant than the Haskell version because all instances of polymorphic types and functions must be declared. Furthermore, due to the multiple type parameters in zipWith, different versions of generic lists and the associated macros must be used in the function definition. In Haskell it may well be worthwhile defining types such as points and the function mk_polygons, but for adtpp we also need to use specialised generic types (list_2 instead of list<t2>), data constructors (Nil_list_2 instead of Nil) and matching primitives (if_Cons_list_2 instead of if_Cons). However, the code has the same structure and safety properties. The structure ensures that NULL pointers are never dereferenced and variables (hd1, tl1, hd2 and tl2) are initialized before they are used. Types are respected because the actual definitions of the C types are not used in the code and only a limited number of abstract operations are supported. For the generic types such as adt_1, no “if” or “case” macros are generated because there are no data constructors. Such types can be passed around and put into and extracted from data structures but not tested or modified by adtpp primitives. Of course if user code contains explicit casts then no safety guarantees can be made. Similarly, explicitly freeing memory is generally unsafe (unless the Boehm-Weiser conservative garbage collector is used — by default it ignores calls to GC_FREE). Also, because ADTs are represented as pointers (see Section 5) there is unfortunately nothing to prevent pointer arithmetic being used on them.

Instances of types and data constructors are made explicit in the code by using longer identifiers, but this seems a small price to pay for enabling the C compiler to type check generic code. For example, type checking ensures there is no inappropriate mixing of the seven distinct list instances: list, list_2, list_3, polygons, colors, points and pointss. In contrast, analogous code in regular C may not be structured in a safe way. It only has a single generic type, void*, and (typically) a single generic list type. Explicit casts are used to convert between distinct list instances, as the C compiler provides no direct help in avoiding confusion between the different list instance types. For example, the assignment to polys from Figure 13 would be written as follows:

```
    polys = (polygons) zipWith(((void*)(*)(void*,void*)) &Pair,
                                (list) cols, (list) ptss);
```

If the last two arguments of the call were swapped there would be no compile time error but a list of “swapped” polygons would be constructed, most likely resulting in a runtime error at some later point in the computation. The adt tool provides a built in polymorphic list type but there is no support for safe conversion between different instances, so similar explicit casts must also be used.

5. IMPLEMENTATION

The adtpp tool is implemented using a combination of flex and bison for tokenising and parsing ADT files, and C code written using the adtpp tool itself. An earlier prototype of adtpp written in a declarative language was used for bootstrapping. A polymorphic list ADT is used, with several instances. One key step in the implementation is to produce a list of monomorphic type declarations where the right hand sides are type expressions that do not contain types defined using type declarations (they may be defined with data declarations or they may be built-in or assumed to be defined elsewhere in C code). For example, the type declaration for polygons is expanded as follows:

```
type polygons = list<pair<color, list<point>>>;
```

This is a canonical form for type expressions and is used to map type expressions to unique C type names. Type expressions can occur in instance declarations and (instances of) data, type and function declarations. They are first expanded to the canonical form then replaced by the corresponding monomorphic type name. For example, in the polygons instance of list<t>, the second argument of Cons is expanded to list<pair<color, list<point>>> then replaced by polygons. If several monomorphic types are equivalent, such as polygon and polygon1, the first name is chosen[§] and subsequent ones are processed to make them the same as the first. We do not describe further details of how adtpp works. Of more interest is the C code adtpp outputs. Here we discuss representation of ADT values then describe details of the macros and functions generated in the header file.

5.1. *Data representation*

We first describe a straightforward way of representing ADTs in C, then discuss how it is done in adtpp. The N arguments of a single data constructor can simply be represented by a C struct with N fields. For a data type with M distinct data constructors, the most straightforward representation in C is to use a union of M structs (as above) plus a tag that encodes which data constructor the value has. The tag can be in an outer struct, along with the union, thus we have a struct containing a tag and union of structs. This is the representation used by the adt tool[¶] [5] (an alternative is to rely on

[§]Monomorphic types defined by data declarations and generic monomorphic types such as list are given priority over those explicitly defined by type declarations.
[¶]The adt tool adds a flag field into every outer struct and, by default, other fields as well. We ignore these in this paper.

```
typedef enum { Noroot=1, Oneroot=2, Tworoot=3 } quad_roots_tag ;
typedef struct quad_roots_struct {
    quad_roots_tag tag ;
    union {
        struct {
            double _root ;
        } _Oneroot ;
        struct {
            double _root1 ;
            double _root2 ;
        } _Tworoot ;
    } data ;
} quad_roots ;
```

Figure 14. Representation of `quad_roots` type in `adt` tool (simplified)

field alignment constraints to add a tag as the first field to each of the structs in the union, resulting in just a union of structs). Figure 14 is a slightly simplified version of the `adt` tool representation for the `quad_roots` ADT.

To assign to such a data type we must assign to the `tag` field of the outer struct and each field of the inner struct that corresponds to the tag value (if it is not `Noroot`). To extract the value we must test the `tag` field then extract the corresponding inner struct, or individual fields of it. C provides no way of ensuring consistency between tag values and which structs are read or written — it is left to the programmer and hence error prone. For example, we can first assign to the `Oneroot` struct and later read from the `Tworoot` struct (returning garbage) because initially the tag was not set properly or later it was not tested properly. A significant benefit of well-supported ADTs is that it is impossible to make such errors. Another disadvantage of unions in C is that the size of a union is (at least) the maximum size of all its elements. For the example above it is very likely that three words will be used, whatever the number of roots.

For `adtpp` we use representations that are similar to those used in many strongly typed declarative languages and are often more compact. Constants (data constructors with no arguments) are represented as integers, starting with zero, in a single word. Data types that only have constants are thus represented like enumerated types in C. For non-constants with N arguments a pointer to a dynamically allocated struct with N fields is used. Where regular C code would use statically allocated space this may be less efficient, but for data structures such as linked lists and trees, dynamic allocation is the norm so `adtpp` has no additional overhead. For a data type that has both constants and non-constants, the two are distinguished by comparing the value in the word with the number of constants in the type. For types with a single constant it is equivalent to testing for `NULL` in regular C. For simple linked lists and trees, the representation in `adtpp` is identical to that in regular C and the code does the same runtime tests. For types with several constants and at least one non-constant we rely on the fact that addresses of dynamically allocated space are greater than or equal to the number of constants in the type (if necessary, we can ensure this by simply wasting some memory with low addresses).

Distinguishing between different non-constants requires a tag. If there are a small number of non-constants, the least significant bits of the pointer are sufficient to store this information. Modern architectures use byte-addressing but dynamically allocated space is aligned to word boundaries so there are at least two and generally three wasted bits that can be used for a tag. Thus we can generally distinguish eight different non-constants without allocating extra space. If there are more than eight non-constants, seven can be distinguished using bits of the pointer and the remainder can have a "secondary" tag stored with the arguments.

To create a value of such a data type, in the worst case we must `malloc` a struct, assign to each of its fields, including the secondary tag, and add in the primary tag. To extract a value we must compare the value with the maximum constant of the type, extract the primary tag, dereference the pointer minus the primary tag, extract the secondary tag and then the other arguments. For most values of most data types, some of these steps are not needed and the code adtpp produces only performs runtime tests and memory accesses that are needed. For example, in the quad_roots ADT, Noroot is represented as a single word containing zero and extracting it only requires a comparison with zero. For Oneroot we simply have a pointer to a (struct containing one) double and extracting it requires an additional test of the least significant bit of the pointer and a dereference. For Tworoot we have a pointer, with 1 added in the least significant bit, to a struct containing two doubles. We must subtract 1 as we dereference the pointer (this typically has no runtime overhead). The Boehm-Weiser conservative garbage collector ignores tags added to pointers (by default) so automatic memory management is not affected.

The details of the C types we use are as follows. Each ADT is defined using typedef as a pointer to a dummy struct, defined for that type, with no fields. This is cast to an integer type when the value is a constant and for operations on the tag. For each data constructor with $N > 0$ arguments we define a struct with N fields, or $N + 1$ fields if a secondary tag is used. We cast between pointers to this struct and the ADT type where necessary. For example, the quad_roots ADT results in the following types defined:

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

The representation used in adtpp could potentially be used by a regular C programmer. However, it is dependent on the number of tag bits available and requires numerous casts, various numeric and bit operations on these values that are pointers et cetera. Having such low level, unreadable and error prone code would normally be unacceptable. But when it is encapsulated in a tool such as adtpp it can be maintained separately and the code that uses it is actually higher level, more readable and less error prone than regular C code, while retaining the efficiency advantages.

5.2. *Macros and functions*

One of the principles behind the design of adtpp is that compile-time error checking should be maximized. One problem with the low level data representation is that casts must be used and this makes it more difficult to take advantage of the type checking in C. We use a couple of techniques to ensure type checking is possible even though casts are used. Because ADT definitions translate to types that are pointers to dummy structs, type errors in C code that uses ADTs most commonly result in error or warning messages such as "assignment from incompatible pointer type". Some C compilers, such as gcc, just give a warning by default but stricter type checking can be enforced by appropriate compiler flags. Although it may sometimes be helpful to provide more information (such as the names of the ADTs), letting the programmer know there is some kind of type error at a specific source code location is generally enough information to quickly identify the error. We also employ other techniques to limit scope and catch various other errors at compile time. Some techniques may potentially impose a small runtime overhead, but even minimal optimisation is sufficient to remove it. We do not give a full description of how all the primitives are implemented but describe enough to cover all the techniques we use.

The primitives for creating ADT values (one for each data constructor) are implemented by static inline functions. Functions are used rather than macros so the types of arguments and returned value can be checked by the C compiler (and we can use "pointers to" these functions in higher order code such as our zipWith example). The use of static functions means there is no problem with name clashes when linking multiple object files, and the functions are all very small and best inlined. The generated definition of Tworoot is given in Figure 15. The arguments are both declared as doubles and the result is quad_roots so incompatible arguments and usage will be picked up by the C compiler type checking. The macro ADT_MALLOC is used to allocate the appropriate struct, the two fields are initialized, and the pointer with the tag value of one added is returned (the pointer is cast to uintptr_t, an unsigned integer the same size as a pointer, and later cast to quad_roots). There

```
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);
}
```

Figure 15. Definition of the `Tworoot` function

```
#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() }}
```

Figure 16. Some if-then-else macros for the `quad_roots` type

is also a static inline free function for each ADT type that checks the tag and, for non-constants, calls `ADT_FREE` on the value with the tag removed.

Each "if" primitive is defined as a macro, as is `else()` and `end_if()`. The separate `end_if()` does not fit well with the style of C but is necessary due to the way we limit scope of variables. These macros include braces to create new blocks in which variables are declared. The braces are not properly balanced within each macro. Incorrect use can unfortunately result in somewhat obscure error messages, but fixing the code is generally straightforward. Without knowledge of the macro definitions, the C source code can appear to be syntactically incorrect (this is also the case for the `adt` tool). However, the macros are designed so that additional semicolons do no harm. Thus "`if_Noroot(v);`" can be used in place of "`if_Noroot(v)`" and a syntax directed editor will assume this is a function call.

The "if" primitives check the tag corresponds to the appropriate data constructor and if so, declare and initialize the variable names used to represent the arguments. The variable scope is limited to the new block created, so arguments of a particular data constructor can only be accessed in code where it is known the deconstructed variable has that data constructor as its value. The arguments are declared with the appropriate types so type checking of arguments can be done. To allow type checking of the deconstructed variable we assign it to a temporary variable, `_ADT_v`, of the appropriate type. This also allows us to avoid re-evaluating the first argument of the "if" (in case it is an expression rather than a variable) and pass the value to any subsequent "else if". However, the assignment can often be optimized away easily. The definitions of the `if_Tworoot`, `else_if_Noroot`, `else()` and `end_if()` macros are given in Figure 16.

The switch primitives are implemented in a similar way to "if". A temporary variable of the appropriate type is introduced to type-check the deconstructed variable. The constant value or primary and secondary tags are extracted as needed and mapped to the range $0 \ldots N-1$, where N is the number of data constructors in the type. This is encapsulated in a macro generated for the type and used as the target of a C switch statement. Each "case" has a label corresponding to the

```
#define switch_quad_roots(v) \
    {quad_roots _quad_roots_tchk, _ADT_v=(v); \
    switch(quad_roots_constructorNum((uintptr_t)(_ADT_v))){{{
#define case_Tworoot(v0, v1) \
    break;}} case 2: \
    {{quad_roots _SW_tchk=_quad_roots_tchk;} \
    {char _quad_roots_tchk; \
    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 default() break;}} default: {{
#define end_switch() }}}}
```

Figure 17. Some switch macros for the `quad_roots` type

data constructor number and initialized declarations for each argument of the data constructor, in a new block to limit the scope. There is also a dummy assignment to ensure the case matches the type of the switch and the dummy variable is redefined so that a "case" can only occur immediately inside a switch. The switch for `quad_roots` and case for `Tworoot` are given in Figure 17.

Polymorphic types are implemented by generating multiple monomorphic instances, as declared by the programmer in the ADT file. The generic types `adt_1` et cetera are defined as pointers to dummy structs, like other ADTs. Instances of polymorphic functions result in static inline functions with appropriate type instances declared for arguments and the result. The definition simply calls the generic function, with arguments cast to the appropriate generic types and the result cast to the type instance. This consistent casting between generic types and other types is safe due to the type checking performed for the generic code. For example, `mk_polygons` is defined as follows:

```
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);
}
```

Where multiple types are equivalent, the later types and all associated macros etc. are defined in terms of the first type. Generating different structs for equivalent types would result in spurious type errors, even if the structs had identical fields. For example, the `polygons1` definition results in the following macros being output (among others):

```
#define polygons1 polygons
#define free_polygons1(v) free_polygons(v)
#define if_Cons_polygons1(v0, v1, v2) if_Cons_polygons(v0, v1, v2)
```

6. PERFORMANCE

The tagged pointer representation we use can have a significant space efficiency advantages compared with using structs and unions with a tag field. For example, consider the `tree` definition given ealier. Figure 18 gives the numbers of bytes allocated per node in various cases. For all benchmarks we use `gcc` version `4.8.2-19ubuntu1` under Ubuntu on Intel `x86_64` hardware (pointers and long integers are both 8 bytes). We give the size in bytes of the structs used to represent internal nodes and empty subtrees, and the (approximate) number of bytes per key overall (which is the sum of the two, since a binary tree with N keys has N internal nodes has $N + 1$ empty subtrees; we assume no sharing of subtrees). We also give the memory used by the standard library version of `malloc` for these structs and the overall tree. Tags and constants are not optimised in `adt` and the total space used for such a tree is around three times as much as for `adtpp`. A regular C version would typically represent empty trees using NULL and non-empty trees using a struct containing a long integer and two pointers, with the same space requirements as `adtpp`.

	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

Figure 18. Binary tree data sizes (bytes)

Regular C can be cumbersome with more complex data types, and most C programmers would not optimize representation in the way `adtpp` does. For example, 234-trees [8] have several different kinds of nodes that need to be distinguished in some way, and regular C code is greatly simplified if a struct that can represent a 4-node is used uniformly (a struct containing three data values, four pointers and a tag to indicate the kind of node), with some space wasted for other nodes. With algebraic data types we can use different data constructors for the different node types, and `adtpp` will use a more compact representation as well as avoiding certain clases of errors. Space can be reduced further (at the cost of somewhat more complex code) by using different data constructors for leaves (which are the majority of nodes), since these nodes don't have subtrees. For example, Figure 19 shows a possible ADT definition for such a "leaf-optimized" 234-tree containing integer keys.

The `adt` tool would yield a representation with the same size as the regular C implementation outlined above, due to the use of a union. Red-Black trees [9] are isomorphic to 234-trees but have a much simpler representation in C (a single kind of node which is the same as a binary tree node but the "color", either red or black, must also be stored), at the cost of conceptually more complex code. Figure 20 shows the percentage of different kinds of nodes with one million random insertions, the space taken for the structs using the different representations and the space taken per key, assuming this node distribution. We assume that for red-black trees the node color can be represented with no additional space. The `adtpp` representation only uses around 13 bytes per key compared with around 40 bytes for the unoptimized representation and 24 bytes for simple (unbalanced) binary search trees. Using around 0.57 pointers per key, this is a very compact data structure for supporting $O(\log N)$ search and update. In the worst case, where there are only 2-nodes, the `adtpp` representation only uses 16 bytes (an overhead of one pointer) per key and the other 234-tree representations use 64; red-black trees always have the same overhead.

```
data t234 {
    Empty234();
    Two(t234, long, t234);
    Three(t234, long, t234, long, t234);
    Four(t234, long, t234, long, t234, long, t234);
    TwoL(long);
    ThreeL(long, long);
    FourL(long, long, long);
}
```

Figure 19. ADT definition for leaf optimized 234-tree

Following [5], we have performed timing experiments for various programs that sum the elements of a binary tree that has a long integer in each internal node as well as in each leaf node. We use two regular C versions, both using a struct with a long and two pointers to represent each node. For

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

Figure 20. Leaf optimized 234-tree structure and data sizes (bytes)

```
// data ltree {
//     Leaf(long);
//     Branch(long, ltree, ltree);
// }

long sum_adtpp_if(ltree cur) {
  if_Leaf(cur, val)
    return val;
  else_if_Branch(val, left, right)
    return val + sum_adtpp_if(left) + sum_adtpp_if(right);
  end_if()
}

long sum_adtpp_sw(ltree cur) {
  switch_ltree(cur)
  case_Leaf(val)
    return val;
  case_Branch(val, left, right)
    return val + sum_adtpp_sw(left) + sum_adtpp_sw(right);
  end_switch()
}
```

Figure 21. `Adtpp` code for timing experiments

leaves both pointers are NULL and for internal nodes neither pointer is NULL. One version stops the recursion when the argument is NULL (the same code can be used to sum the elements in a tree that doesn't distinguish leaf nodes) and the other stops recursion when the left pointer of the argument is NULL (the argument itself is assumed to be not NULL). Two of the faster and elegant codings using the `adt` tool are used: one using an if-then-else construct and the other using a switch construct (Figure 23 has code in the same style). These are compiled both in the default way and using the "`_FAST_`" flag, which omits some runtime safety checks (in [5] it is suggested this flag should be used when the code is fully debugged). The code for the C and `adt` versions is in [5]. We also use two `adtpp` versions — see Figure 21. As in [5] we use a balanced tree where the branches are shared at each level, resulting in space usage proportional to the depth, even though there are (conceptually) an exponential number of nodes.

Figure 22 gives the times in seconds for a tree with depth 28 (around 27 million nodes). The extra safety afforded by `adtpp` has an apparent cost for the if-then-else code, which tests the tag twice for internal nodes — it is slightly slower than the `adt` version that omits tag checks. However, the cost is quite small (much less than the cost in the `adt` tool for all but the most agressive optimization level) and the switch version is equal fastest (along with the `adt` version with checks omitted). Furthermore, the data structure used by `adtpp` is the most compact: without sharing subtrees and ignoring `malloc` overheads it uses 16 bytes per key compared to 24 for regular C and 32 for the `adt` style of representation.

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

Figure 22. Times for summing elements of a tree (seconds)

7. RELATED WORK

We cannot hope to include even a superficial discussion of all programming languages that are related to our extension of C. Instead, we discuss the most closely related work on extending languages to include algebraic data types, and also comment on how our treatment of polymorphism is related to polymorphism in other languages. There are language extensions such as jADT[||], which extends Java, but our work is closely tied to the features of C, such as the low level representation details and use of macros. The Extended Objective-C library[**] includes support for ADTs. It uses macros to implement the extension (rather than a separate preprocessor) and uses the struct containing a union of structs representation, but no polymorphism.

The most closely related work is clearly the `adt` tool [5], which we have discussed some aspects of already, such as the data representation. It provides some support for polymorphism, with a small number of built-in polymorphic types, and it is (reportedly) relatively easy to modify the tool to support new polymorphic types. There is no automatic safe casting as provided in `adtpp`. However, the `adt` tool provides a greater selection of macros, particularly for pattern matching and deconstruction. `Adtpp` supports equivalents of what are considered in [5] to be the most elegant and efficient pattern matching primitives, and provides more error checking. For example, Figure 23 shows the "best" version of `sum_tree` (Figure 4) supported by the `adt` tool. The `adt` code is lower level than our `adtpp` version due to the explicit pointer types and extraction of a tag from `t` to determine the data constructor. It is also more error prone than `adtpp` code. The variables `val`, `tl` and `tr` have explicit declarations (which can be wrong) and the variables can be used before they are assigned to (for example, in the `Empty` case). Furthermore, the target of the switch may be unrelated to the different cases. For example, a case for the empty list (`Nil`) could be used instead of `Empty`; this would result in a compile-time error in `adtpp` but not in the `adt` tool. We consider our `adtpp` version to be more elegant. It definitely has fewer opportunities for errors and, as discussed in Section 5, is also more efficient.

The polymorphic types supported by `adtpp` are closely related to the types of various declarative languages such as ML [2], Haskell [3] and Mercury [4]. The compilers of these languages infer types for each instance of a function, data constructor and variable. For example, the `Cons` data constructor takes a value of type t and a value of type list of t and returns a value of type list of t. Each occurrence of `Cons` in the program may have a different instance of t, and this is determined by the type checking/inference pass of the compiler. Because `adtpp` simply relies on the C compiler, which has a much simpler type checking algorithm, we rely on the programmer to explicitly give different versions of `Cons` for different occurrences. Similarly, there must be different versions of polymorphic functions. This means instances must be declared and named, and makes the code somewhat more verbose, but no less flexible.

[||]`http://jamesiry.github.io/jADT/index.html`
[**]`https://github.com/jspahrsummers/libextobjc`

```
long sum_adt(tree *t) {
    long val;
    tree *tl, *tr;
    switch(gttreetag(t)) {
    csEmpty(t)
        return 0;
    csNode(t, val, tl, tr)
        return val + sum_adt(tl) + sum_adt(tr);
    }
}
```

Figure 23. Summing tree elements using the best method in the `adt` tool

A little flexibility is lost because types of variable occurrences must also be fixed, rather than inferred by the compiler. In pure functional code, a variable `xs` may have type list of t, and different occurrences of the variable can have different instances of the type. However, this is can be unsafe in the presence of destructive update. ML allows some destructive update and imposes the "value restriction" [10], which rules out using polymorphic types for variables that can potentially be updated. In `adtpp` there are no restrictions on destructive update so it seems inevitable that variables with polymorphic types cannot be supported while maintaining type safety.

8. FURTHER WORK

There are many ways in which `adtpp` could be enhanced and extended. Here we briefly mention a few. The variety of macros/functions produced could be extended in various ways. For example, a version of `free` that traverses and frees an entire data structure, and perhaps other traversal functions could be provided, as in the `adt` tool. Macros that combine deconstruction with a while loop could be useful, particularly for data types that have recursive cases in just one data constructor (such as lists and trees). `Adtpp` could support an analogue of Haskell's `newtype` declarations, which are like `data` declarations but there must be just one data constructor with one argument. This allows two distinct isomorphic types and the implementation can be optimised to avoid the extra indirection. For example, the type `userid` below would be distinct from `adt_int`, so the two would not be accidentally confused, but the representation could be a struct containing an `adt_int` rather than a pointer to such a struct.

```
newtype userid {
    Userid(adt_int);
}
```

This idea of avoiding pointers could be extended to other data types that are no larger than a pointer. Unfortunately, the macro processor `cpp` cannot evaluate '`sizeof`' expressions, but potentially a two-pass approach could be used, where `adtpp` outputs C code, which is then compiled and run to produce a header file. Safety could be enhanced further by avoiding defining ADTs as pointer types (so pointer arithmetic cannot be performed). Pseudo-random names could also be used for struct type and member names, to make it more difficult for programmers to circumvent `adtpp`'s type safety.

Pattern matching and constructing monomorphic instances of polymorphic types could be made less verbose by allowing the user to declare the specialised constructor names for each monomorphic instance.

9. CONCLUSION

The support for algebraic data types in various declarative languages has many advantages. The data types are very expressive, allowing precise descriptions of valid values. The programming constructs for constructing and testing plus deconstructing values lead to naturally safe code. Parametric polymorphism allows significant code reuse and abstraction. `Adtpp` is a software tool that gives very similar support for ADTs in C by processing a file containing ADT definitions and other declarations and producing a "`.h`" C header file. Compared with programming in regular C, many potential errors can be detected at compile time. Data is represented in ways similar to declarative language implementations. In many simple cases the representation is identical to that in regular C and for more complex types it is often more compact. `Adtpp` produces macros and inline functions with only a small overhead that can generally be eliminated by an optimising compiler. Support for polymorphism is limited by the reliance on the standard C compiler for type checking — some of the work done by compilers for declarative languages must be done by the programmer instead. Each type instance of polymorphic functions and data constructors much be declared and named separately in the source code rather than this information being inferred by the compiler. Other than this, the code can have identical structure and safety properties to code in declarative languages. The system is open-source and available from `https://bitbucket.org/Macnaa/adt4c-with-polymorphism.git`.

REFERENCES

1. Burstall RM, MacQueen DB, Sannella D. HOPE: an experimental applicative language. *LFP '80: Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, ACM: New York, NY, USA, 1980; 136–143.
2. Milner R, Tofte M, Macqueen D. *The Definition of Standard ML*. MIT Press: Cambridge, MA, USA, 1997.
3. Jones SP, Hughes J, Augustsson L, Barton D, Boutel B, Burton W, Fasel J, Hammond K, Hinze R, Hudak P, *et al.*. Report on the programming language Haskell 98, a non-strict purely functional language, February 1999 1999; .
4. Somogyi Z, Henderson F, Conway TC. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Log. Program.* 1996; **29**(1-3):17–64.
5. Hartel PH, Muller HL. Simple algebraic data types for C. *Softw., Pract. Exper.* 2012; **42**(2):191–210. URL `http://dblp.uni-trier.de/db/journals/spe/spe42.html`.
6. Naish L. Sharing analysis in the Pawns compiler. *PeerJ Computer Science* 2015; **1**(e22), doi:10.7717/peerj-cs.22. URL `https://dx.doi.org/10.7717/peerj-cs.22`.
7. Boehm HJ, Weiser M. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.* Sep 1988; **18**(9):807–820, doi:10.1002/spe.4380180902. URL `http://dx.doi.org/10.1002/spe.4380180902`.
8. Bayer R. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* 1972; **1**(4):290–306, doi:10.1007/BF00289509. URL `http://dx.doi.org/10.1007/BF00289509`.
9. Guibas LJ, Sedgewick R. A dichromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, IEEE Computer Society: Washington, DC, USA, 1978; 8–21, doi:10.1109/SFCS.1978.3. URL `http://dx.doi.org/10.1109/SFCS.1978.3`.
10. Wright A. Simple imperative polymorphism 1995; **8**(4):343–356.