Simple algebraic data types for C

Pieter Hartel and Henk Muller

Version 8, 2nd September, 2010

Abstract

Adt is a simple tool in the spirit of Lex and Yacc that makes monomorphic algebraic data types, polymorphic built-in types like the list and an efficient form of pattern matching available in C programs. C programs built with ADTs typically use NULL pointers only to indicate don't care values, and not as sentinels. This reduces the scope for errors involving NULL pointers. The Adt tool generates runtime checks, which catch many of the remaining NULL pointer dereferences. The runtime checks may consume a significant amount of CPU time; hence they can be switched off once the program is suitably debugged.

1 Introduction

Brooks [3] advocates writing support tools to avoid repetitive and error prone work. A task that we have often encountered is the construction of an Algebraic Data Type (ADT) and associated functions needed to build and traverse a parse tree. Lex and Yacc deliver the raw parsing power, but provide little support for writing the semantic actions. Therefore, our contribution is a simple tool called Adt that, on the basis of a set of monomorphic, user-defined ADT specifications and/or polymorphic built-in data types, generates C type and function declarations that create and manipulate the corresponding data structures. The Adt tool is small and efficient, and it interworks well with Yacc.

The reader may wonder why we propose a new tool for C. The reason is that C is still used, especially in embedded systems, where speed and size matter. C is a "low level" high level language, which gives programmers more control than any other high level language [1]. Embedded systems contain sophisticated code that makes heavy use of dynamic data structures, for example MP3 players, video game consoles, and navigation systems, where we believe our Adt tool support would be welcome.

The plan of the paper is as follows. The next section introduces ADTs. Section 3 describes how the Adt tool realises ADT support for C. Section 4 describes the support for polymorphic built-in lists and trees. Section 5 describes the Adt tool itself in some detail. Section 6 reports on our experiences using the Adt tool.

2 Algebraic data types

An ADT is: a *recursively defined sum* of *product* type. To explain what this means we define as an example the ubiquitous binary tree:

base case A Leaf is a binary tree.

recursive case A Branch with two sub trees, both of which are binary trees, is also a binary tree.

no junk Nothing else is a binary tree.

All three aspects of the ADT are present in the example. In the second definition we see that a binary tree is defined in terms of itself, this makes the ADT *recursively defined*. The base case for a Leaf and the recursive case for a Branch are the two summands of the *sum* type. The two sub trees in the recursive case, both of which must be present for a tree to be a binary tree, form the factors of a *product* type. The last definition states that functions operating on a binary tree do not have to take special precautions to deal with anything else than binary trees. The definition follows what we will call the "recursive sum-of-product" pattern.

2.1 Recursive sum-of-product pattern

The power of an ADT comes from the fact that we can use the recursive sum-of-product pattern from the definition also in functions to decompose objects of the ADT. For example a function to count the number of nodes in a binary tree would have two cases:

base case A Leaf has one node.

recursive case A Branch has as many nodes as 1 + the number of nodes in the left sub tree + the number of nodes in the right sub tree.

Again the three aspects of the ADT are present, but this time in the function definition. In the recursive case we see that the function calls itself recursively, to deal with the left and right sub trees (*recursively defined*). For each of the two possible forms of a binary tree we have a clause that can deal with this form comprehensively (*sum*). One is added to the number of nodes in the two sub trees

```
datatype tree = Leaf of (int)
              | Branch of (int * tree * tree);

fun sum(Leaf(val)) = val;
  | sum(Branch(val, left, right))
                   = val + sum(left) + sum(right);

main = sum(Branch(10, Leaf(20), Leaf(30)));
```

Figure 1: The SML binary tree and sum function.

added together. This combines the results from the two factors in a result for the product (*product*). The fact that every function operating on a binary tree follows the same recursive sum-of-product pattern helps to avoid mistakes, especially, if we can automate some of the tedious work with the Adt tool. But first we will look at programming languages that already support ADTs for inspiration.

2.2 Languages that support ADTs

There are many programming languages that provide ADT support. Most of these are declarative languages, such as Prolog [5], and Standard ML [15], but there are also object oriented languages that support ADTs, such as Scala [17]. We use Standard ML to illustrate how the recursive sum-of-product pattern is used in that language.

To make our binary tree running example a bit more interesting, we give, in Figure 1, the definition of a slightly enhanced binary tree in Standard ML. The enhancement consists of adding an integer value to every node, so that we can add the values together, rather than simply counting the nodes.

Both the data type declaration and the function declarations follow the recursive sum-of-product pattern. The vertical bar `|` separates the two definitions of the data type as well as the two definitions of function declaration. The two summands are each identified by a constructor: `Leaf(.)` or `Branch(.,.,.)`. The left hand sides of the function definitions use pattern matching to deconstruct a binary tree. Firstly, the constructor in the actual argument is matched to the constructor specified in the pattern of the formal argument. Hence a tree like `Leaf(20)` will be matched by the first function definition `Leaf(val)`, but not by the second. As part of the matching process, the value stored in a node (here 20) is bound to the variable `val`, so that it can be used in the function body. In this case the function body returns the value to its caller.

The expression `main` constructs a sample binary tree with three nodes and applies the `sum` function to the tree. When executed, the answer will be computed as 60.

The Standard ML program is short and elegant, thanks to the support provided for ADTs. We will now investigate how C programmers approach the `sum` problem.

```
typedef struct tree_struct {
  int val;
  struct tree_struct *left;
  struct tree_struct *right;
} tree;

tree *mkBRANCH(int val, tree *left, tree *right) {
  tree *result = calloc(1,
                     sizeof(struct tree_struct));
  if(result == NULL) {
    printf("panic\n");
  }
  result->val = val;
  result->left = left;
  result->right = right;
  return result;
}

int krsum1(tree *cur) {
  if(cur == NULL) {
    return 0;
  } else {
    return cur->val + krsum1(cur->left)
                    + krsum1(cur->right);
  }
}

int krsum2(tree_cc *cur) {
  /*assert cur->left==NULL <==> cur->right==NULL*/
  if(cur->left == NULL) {
    return cur->val;
  } else {
    return cur->val + krsum1(cur->left)
                    + krsum2(cur->right);
  }
}

void test() {
  tree *r = mkBRANCH(30, NULL, NULL);
  tree *l = mkBRANCH(20, NULL, NULL);
  tree *t = mkBRANCH(10, l, r);
  printf("%d\n", krsum1(t));
}
```

Figure 2: C version of the binary tree and two sum functions.

2.3 C does not have ADT support

C does not provide ADTs but it does provide the necessary building blocks for ADT support, such as `struct` and `union`. We will first look at the traditional way of creating a binary tree in C, inspired by Kernighan and Richie [12, Page 141].

Like the Standard ML program of Figure 1, the C program of Figure 2 creates a binary tree with three nodes, calls `krsum1` to visit all three nodes, adds the values found in the `val` fields, and prints 60.

The difference between the C *data type* `tree` and the Standard ML version is that in C we have no explicit cases for Branch and Leaf. Instead the two cases are distinguished implicitly by the conventional use of the special pointer value `NULL`. The problem with this approach is that C programmers use NULL pointers for two different purposes. The first is to indicate the end of a list, or the leaf nodes of a tree. In this case, a NULL pointer is essentially used as a sentinel. If there are several different data types in a program, the same `NULL` value is used with each different data type thus preventing the compiler to spot type errors. The second use of NULL is to indicate an uninitialized or *don't care* value. These two totally different meanings can lead to errors, for example if an "uninitialised" value is interpreted as an "end of list". Tony Hoare, who invented the `NULL` pointer, has called this his "billion dollar mistake" [9].

The difference between the C *functions* of Figure 2 and the Standard ML version is that in C we have to handle the memory allocation and de-allocation for the binary tree explicitly. To do this, we have packaged the calls to the library function `calloc` in the function `mkBRANCH`, which checks the return value from `calloc`, and which fills in the three fields of the `struct`. We could have added several more functions, such as a function to free the storage occupied by a tree, getters and setters for the fields etc. This is fine as long as there is only one dynamic data structure in a program, but it gets tedious and error prone to write a similar set of functions for different data types. C programmers usually resort to a kind of polymorphism by using functions with `void *` parameters and/or results. Again this prevents the compiler from finding type errors, thus making life harder on the programmer than necessary.

Figure 2 actually shows two versions of the sum function. The first version `krsum1` looks natural, whereas the second version `krsum2` looks unnatural. However, the former makes about twice as many recursive calls as the latter. This means that natural looking C code is not actually efficient!

It is our objective to avoid the billion dollar mistake as well as `void *` pointers by creating efficient support for ADTs in C. This is the topic of the rest of the paper.

```
tree = /* base case % */
       LEAF(int val)
     | /* recursive case % */
       BRANCH(int val, tree *left, tree *right);
```

Figure 3: Tree ADT.

3 ADT support for C

The Adt tool allows C programmers to describe an ADT in a syntax inspired by the Standard ML syntax of Figure 1. The tool reads the ADT specification and generates a `.h` file and a `.c` file which provide all the C declarations necessary to be able to use the recursive sum-of-product pattern of programming in arbitrary C code.

To begin with we present the binary tree data type in the input syntax of the Adt tool in Figure 3.

The differences between the Standard ML definition of `tree` and the Adt tool version are threefold. We dispensed with some keywords, such as `datatype` since the Adt tool only deals with data types, and nothing else. Secondly, we have given names (`val`, `left`, and `right`) to the summands so that we can refer to them by a meaningful name. Thirdly, more subtly, to acknowledge that a binary tree is a dynamic data structure explicitly allocated on the heap, we use the `*` annotation of C to indicate pointers, and not as in standard ML, to separate the summands.

In the next section we will discuss the C type definitions generated by the Adt tool for the `tree` ADT. This will be followed by a description of the C functions and macros necessary to manipulate the `tree` in four different pattern matching styles. We will refer to the collection of these functions as the *standard* functions, to distinguish them from the *extra* functions generated for the polymorphic built-in types discussed in Section 4.

3.1 C type definitions

The two `typedefs` below are generated for the ADT `tree`. No user defined code should access any of the fields directly; all access should be mediated by functions generated by the Adt tool, as discussed in the following sections.

```
typedef enum {tree_BIND=0,LEAF=1,BRANCH=2} tree_tag;
typedef struct tree_struct {
  tree_tag tag;
  int      flag;
  int      lineno;
  int      charno;
  char    *filename;
  union {
    struct tree_struct **_binding;
    struct {
      int _val;
    } _LEAF;
    struct {
      int _val;
      struct tree_struct *_left;
      struct tree_struct *_right;
    } _BRANCH;
  } data;
} tree;
```

The tag field is used to distinguish the various summands, which are represented here as members of the union. The Adt tool ensures that all tags are unique, which is useful for debugging. The flag field is use to avoid printing shared structures more than once (See Section 3.4). As one of the intended uses of the Adt tool is building compilers, the lineno, charno, and filename fields can be used for generating error messages. (These three fields can be removed by selecting -e option of the Adt tool.)

3.2 Constructor functions

The Adt tool generates two constructor functions for the tree data type. These are mkLEAF and mkBRANCH as shown below.

```
tree *mkLEAF(int _val);
tree *mkBRANCH(int _val, tree *_left, tree *_right);
```

The type and number of the arguments of the two constructor functions correspond exactly to the type and number of the factors of each product type. The constructor functions use the calloc library function to allocate the data on the heap. Checks for heap overflow are compiled in automatically.

```
tree *cptree(tree *subject);
void  mvtree(tree *subject, tree *object);
```

Sometimes it is necessary to copy or move nodes. The function cptree allocates storage on the heap and makes an exact copy of the subject. The function mvtree copies the contents of the subject to the object, thereby destroying the old contents of the latter.

3.3 Destructor functions

Neither C nor the Adt tool support provide garbage collection, hence the Adt tool generates functions to free the space allocated by the constructor functions. The function frtree releases the node passed as a reference, the function fdtree (the d stands for deep) releases the entire tree:

```
void frtree(tree **subject);
void fdtree(tree **subject);
```

The free functions must be called with the address of variable so that the variable can be set to NULL to indicate that the node previously available through the variable is no longer available. This helps to avoid de-allocated data being used by accident. It is only a partial solution, as there are usually other variables holding the same value, which are not set to NULL. To achieve this too, something like C++ Smart Pointers [21] would be needed, which beyond the scope of this work because it requires garbage collection.

The fdtree function assumes that the data structure is a tree and not a graph. If this assumption is not valid, the programmer will have to deallocate the graph using an appropriate algorithm that calls frtree for individual nodes.

3.4 Debugging functions

The Adt tool generates a debugging function called prtree, which prints an indented representation of the tree data structure. The print function uses a flag in each node to remember whether the node has been printed, so that it can handle graphs. The function cltree resets this flag.

```
void prtree(int indent, tree *subject);
void cltree(tree *subject);
```

3.5 Access functions

A variety of access functions is available. The function gttreetag returns the constructor tag, represented as one of the enumerated types LEAF, and BRANCH. For each of the factors in the product types there are get and set functions, i.e. gtLEAFval gets the val factor of the product with constructor tag LEAF:

```
int   gttreetag(tree *subject);
int   gtLEAFval(tree *subject);
void  stLEAFval(tree *subject, int value);
int   gtBRANCHval(tree *subject);
void  stBRANCHval(tree *subject, int value);
tree *gtBRANCHleft(tree *subject);
void  stBRANCHleft(tree *subject, tree *value);
tree *gtBRANCHright(tree *subject);
void  stBRANCHright(tree *subject, tree *value);
```

The main purpose of the access functions is to make sure that any access to a NULL pointer is detected as early as possible, and to check that the tag is correct. For example the Adt tool generates the following code for the gtLEAFval function:

```
int gtLEAFval(tree *subject) {
  if(subject == NULL) {
    abort(...)
  }
  if(subject->tag != LEAF) {
    abort(...)
  }
  return subject->data._LEAF._val;
}
```

Once the program is sufficiently debugged, one may consider using (by setting a compilation flag) the following macro definition instead:

```
#define gtLEAFval(subject) \
                        ((subject)->data._LEAF._val)
```

The macro does not perform checks, and it will reduce the runtime (See Section 3.7 for details).

3.6 Pattern matching functions

A powerful feature of languages with built-in support for ADTs is pattern matching, where a pattern is matched and bound to an instance of an ADT. The ADT tool supports pattern matching by functions with a mt prefix.

```
bool mttree(tree *pattern, tree *subject);
```

The function mttree requires two arguments, both of type tree *. The first argument represents a pattern to which the second argument subject is matched. The mttree function compares both arguments, starting with the root of each, recursively descending into the sub trees of each argument. If the tags of the roots pattern and subject match, the children of the roots will be compared until either the pattern or the subject is exhausted. Pointers to variables can be stored in a pattern which will be bound when a match is made.

The patterns can be created on the stack by macro definitions with the prefix in, or on the heap by functions with the prefix pt. Since these forms of pattern matching are relatively expensive it is also possible to a switch statement with macros with prefix cs for the cases. Finally, it is possible to use the access functions directly to deconstruct an object. These four different styles of programming are discussed next. We begin by discussing the solution that stores a pattern on the heap.

3.6.1 Pattern matching on the heap

We can write C code that corresponds to the Standard ML code of Figure 1 by building a pattern on the heap that is matched against an actual tree. A pattern is allocated on the heap by functions with a pt prefix. The Adt tool generates two such functions for the tree ADT:

```
tree *ptLEAF(int _val);
tree *ptBRANCH(int _val, tree *_left, tree *_right);
```

These functions are in fact, apart from the name, identical to constructor functions of Section 3.2.

To be able to bind actual values to variables, we need the set of functions generated by the Adt tool with the prefix bd.

```
int   bdint(int *binding);
tree *bdtree(tree **binding);
```

For example a call like bdtree(&left) allocates a node on the heap to save the address of the variable left, so that the matching function knows where to store the result of the match. To distinguish a binding node from the regular Leaf and Branch nodes it has a special tag BIND.

The function ptsum uses functions with the pt and bd prefixes as follows:

```
int ptsum(tree *cur) {
  int val = -1;
  tree *left, *right;
  tree *leaf = ptLEAF(bdtint(&val));
  if(!mttree(leaf, cur)) {
    tree *branch = ptBRANCH(bdtint(&val),
                   bdtree(&left), bdtree(&right));
    if(mttree(branch, cur)) {
      val = val + ptsum(left) + ptsum(left);
    }
    fdtree(branch);
  }
  fdtree(leaf);
  return val;
}
```

The pattern ptLEAF(...) creates a tree_struct on the heap that stores the integer found in a LEAF node in the local variable val. Similarly, when a match succeeds, the pattern ptBRANCH(...) stores the integer value found in a BRANCH node in the local variable val via the binder bdint(&val), and it stores pointers to the left and right children in the local variables left and right via the binders bdtree(&left), and bdtree(&right). The calls to the fdfree function are necessary to deallocate the heap storage occupied by the patterns before the function returns.

The C function ptsum is reasonably elegant. It is possible, using the same pair of pattern and matching functions to create arbitrarily nested patterns. One of the points of criticism could be that if the first match fails, the second must succeed (by the no junk rule), thus making the second if redundant. While this is true in case of the example, in general this need not be the case, as there could be values in the pattern that fail to match. For example we could be interested in treating a Branch with a 0 value differently from the others.

In the next section we will discuss a set of macros (with the in prefix) that can be used to allocate patterns on the stack instead of the heap. This improves the performance but reduces flexibility.

3.6.2 Pattern matching in the stack

The macros with the in prefix that allocate patterns on the stack are actually C99 style initialisers. This is not a restriction of the usability of the Adt tool, as C99 is supported by a variety of compilers, including GCC.

```
#define intree(binding) { \
  .tag = tree_BIND, \
  .data._binding = binding \
}
#define inLEAF(val) { \
  .tag = LEAF, \
  .data._LEAF._val = (int)val \
}
#define inBRANCH(val, left, right) { \
  .tag = BRANCH, \
  .data._BRANCH._val = (int)val, \
  .data._BRANCH._left = left, \
  .data._BRANCH._right = right \
}
```

The macros with the in prefix do exactly the same as the functions with the pt prefix; the only difference is in the storage area for the pattern (stack or heap). Since the macros are implemented as initialisers, they cannot be nested syntactically, hence the verbose formulation at the beginning of the else branch below. However, arbitrary dynamic nesting is still possible.

```
int insum(tree *cur) {
  int val;
  struct tree_struct valbind = inint(&val);
  struct tree_struct leaf = inLEAF(&valbind);
  if(mttree(&leaf, cur)) {
    return val;
  } else {
    tree *left, *right;
    struct tree_struct leftbind = intree(&left);
    struct tree_struct rightbind = intree(&right);
    struct tree_struct branch =
         inBRANCH(&valbind, &leftbind, &rightbind);
    if(mttree(&branch, cur)) {
      return val + insum(left) + insum(left);
    }
  }
}
```

The C function insum is not particularly elegant but demonstrates that patterns can be built that are matched dynamically, and that local variables are bound to matched values. Since the patterns are stored on the stack, there is no need to free the patterns explicitly, thus saving execution time.

3.6.3 A pattern matching case statement

The third style of pattern matching consists of macros with a cs prefix as shown below. Only the constructor can be matched; a local variable will be bound to each field. It is thus not possible to nest the macros.

```
#define csLEAF(_tree_, val) \
  case LEAF : \
    val = _tree_->data._LEAF._val;
#define csBRANCH(_tree_, val, left, right) \
  case BRANCH : \
    val = _tree_->data._BRANCH._val; \
    left = _tree_->data._BRANCH._left; \
    right = _tree_->data._BRANCH._right;
```

With the cs macros it is possible to produce even more legible code, which is also more efficient code than with the pt functions and the in macros.

```
int cssum(tree *cur) {
  int val;
  tree *left, *right;
  switch(gttreetag(cur)) {
  csLEAF(cur, val)
    return val;
  csBRANCH(cur, val, left, right)
    return val + cssum(left) + cssum(right);
  }
}
```

We believe the code above to be readable, which is more easily seen if we compare the code to the final version of the sum function that uses access functions. This is the topic of the next section.

3.6.4 Field access functions

The access functions of Section 3.5 can be used also to deconstruct objects. This avoids pattern matching completely and is more in line with traditional C programming:

```
int gtsum(tree *cur) {
  switch(gttreetag(cur)) {
  case LEAF :
    return gtLEAFval(cur);
  case BRANCH :
    return gtBRANCHval(cur) +
      gtsum(gtBRANCHleft(cur)) +
      gtsum(gtBRANCHright(cur));
  }
}
```

The code of gtsum above is a little cluttered with the many access functions that are not present in cssum. Therefore we argue that pattern matching helps to write readable code. The version cssum of Section 3.6.3 is, in our view, the best.

We have now completed the presentation of the standard functions generated for each ADT. Before we discuss polymorphic builtin types we will take stock of the

legend	ptsum	insum	cssum	gtsum	krsum1	krsum2	ttsum
Section	3.6.1	3.6.2	3.6.3	3.6.4	2.3	2.3	4.3
Time per call	157±3 ns	16±1 ns	5±0 ns	5±0 ns	7±0	5±0 ns	7±0 ns
Times slower	31×	3.3×	1×	1×	1.4×	1×	1.4×
elegance	+/-	-	++	+/-	-	−	+
nesting	+	+	-	-	n.a.	n.a.	n.a.

Table 1: Performance and elegance of the four styles of pattern matching. All run time checks are off.

development thus far by looking at the performance of the various sum functions. This is the topic of the next section.

3.7 Quantitative Evaluation

The performance of the generated code depends on the style of pattern matching used. To measure the performance of the code we wrote a simple test program that creates a graph with 30 maximally shared nodes that when interpreted as a binary tree appears to have 2^{30} nodes. The data structure will fit into any cache, thus ensuring that the computation is CPU bound. We have benchmarked our test programs on a GNU/Linux system with an Intel Xeon E5504 processor, which has a clock speed of 2 GHz. We enable the `-O3` aggressive optimisation of the GNU C compiler. The average run time over 10 runs in ns per call to the sum function of all versions of the test program is shown in Table 1.

With 5ns on average per call, the functions `cssum`, `gtsum`, and `krsum2` are the winners.

With 7ns on average per call, the functions `krsum1` and `ttsum` are a little slower. `krsum1` makes unnecessary recursive calls (See Section 2.3) and `ttsum` uses *Currying* [2, Chapter 1], which also requires extra calls (See Section 4.3).

The versatility of nested patterns comes at a price. The function `insum`, which allocates patterns on the stack, is about 3 times slower than the fastest sum function. The version `ptsum`, which allocates patterns on the heap, is about 31 times slower than the fastest.

The runtime checks for NULL pointers are off in all of the benchmarks. Turning them on only makes a significant difference to `cssum` and `gtsum`, because these versions use the access functions from Section 3.5. Turning the NULL pointer checks on slows down `cssum` by a factor 1.2, whereas `gtsum` slows down by a factor 2.

As a conclusion we submit that C code with ADTs can be even more efficient than traditional C code.

4 Polymorphic built-in types

The ADTs that the user can define are monomorphic, which means that the data structure has a specific type. However, our source of inspiration, i.e. the type system of Standard ML, is polymorphic. This means that where we have thus far used specific types one could use more general, polymorphic types, which can be instantiated to different specific types. For example instead of saying that the values in a tree are of type `int`, in Standard ML one might leave this decision for later and define a tree with a polymorphic type like this:

```
datatype 'a tree = Leaf of ('a)
                 | Branch of ('a * tree * tree);
```

With this definition we can make a tree of integers, but also a tree of strings, a tree of trees of integers etc, simply by instantiating the type parameter `'a` to `int`, `string`, or `(int tree)` respectively.

Polymorphic types are powerful, but the implementation brings certain complications that are not appropriate for a tool like Adt that is meant to be simple. Hence we have decided not to extend the Adt with support for user defined polymorphic functions. Instead, we offer polymorphism only for certain, often used types, such as lists and trees, by building the types and the functions operating on the types into the Adt tool itself. The built-in types are actually extensions of the regular ADTs, and thus share most of the machinery that we have already presented. The three built-in types will be discussed in the following section.

At present three polymorphic built-in types and the most relevant functions operating on those types have been implemented: the singly linked list, the doubly linked list, and the binary tree. Implementing the doubly linked list took one day of work, and implementing the binary tree just one hour, so we expect that building other polymorphic types into the Adt tool would not be that time consuming.

The Adt tool generates the full complement of standard functions and macros as described in Sections 3.2 . . . 3.5 for the built-in types, as well as one or more extra functions that are specific to the built-in type.

4.1 Singly linked list

For the singly linked list the Adt tool generates the standard functions as well as two extra functions that are often used on lists: append and map. More functions could be added. Consider as an example a list of trees defined thus:

```
forest = [tree *tree];
```

The type of the element of the list is `tree *` and the name of the element is `tree` (there is a separate name space for types, so the same identifier may be used).

The declaration of `forest` above is actually interpreted as the regular monomorphic ADT below, but with a twist: an extra field `next` field is added to link subsequent items of the list together. The end of the list is indicated by the NULL pointer as a sentinel, as in traditional C code.

```
forest = FOREST(tree *tree, forest *next);
```

This interpretation causes the Adt tool to generate all the C types and standard functions that it generates for any monomorphic ADT, including `forest`. On top of this the Adt tool generates a number of extra functions; two in this case. The prototypes of the append function `apforest` and the map function `itforest` (it for iterate) are:

```
forest *apforest(forest *subject, forest *object);
void itforest(void (*f) (void *, tree *), void *x,
                                  forest *subject);
```

The function `apforest` traverses the spine of the `subject` list to the last node and sets its `next` field to point at the `object` list. A functional version of `apforest`, which makes a copy of the spine of the `subject` list, attaching the `object` list at the end of the copy could be implemented easily.

The function `itforest` traverses the spine of the `subject` list, supplying each element to the function `f` passed as the first argument to `itforest`. To increase the flexibility, `itforest` has a second argument `x` that is passed to `f` at each call. `x` can be used to accumulate results. This is the only place where `void *` pointers are used in the Adt tool. This mechanism allows for the functionality of Currying to be implemented. (See Section 4.3 for a more detailed example using the same mechanism.)

The notation `[tree *tree]` in the Adt input syntax is actually syntactic sugar for `INSTANCE list (tree *tree)`. This notation is meant to convey that we that are instantiating the (implicit) type parameter of the polymorphic built-in type `list` with `(tree *tree)`. In the next two sections we will see two more examples of type instantiation.

4.2 Doubly linked list

A doubly linked list can be instantiated with `INSTANCE dlist (...)`. Consider as an example a doubly linked list of trees:

```
forest = INSTANCE dlist (tree *tree);
```

This declaration of `forest` is interpreted as the regular ADT below. Here we are adding two fields, one for the pointer to the next node, and one for the pointer to the previous node (where the next node of the last element of the list is the first and vice versa).

```
forest = FOREST(int val, forest *prev, forest *next);
```

The Adt tool generates the standard functions as well as a set of extra functions taken from the C++ STL container class `list` [7, Section 22.2.2]:

```
int     size_forest(forest *subject);
bool    empty_forest(forest *subject);
forest *front_forest(forest *subject);
forest *back_forest(forest *subject);
forest *push_back_forest(forest *subject,
                                     forest *object);
forest *push_front_forest(forest *subject,
                                     forest *object);
forest *pop_back_forest(forest *subject);
forest *pop_front_forest(forest *subject);
forest *erase_forest(forest *subject,
                         forest *start, forest *end);
forest *insert_forest(forest *subject,
                      forest *start, forest *object);
```

The remaining 19 member functions from the STL container class could be added also.

4.3 Binary tree

Finally, we have implemented some support for binary trees (See the Appendix for the full source code of the implementation). The following Adt input is almost a complete program, only the `main` program that calls `ttsum` with a sample tree is missing.

```
%{
#include "primitive.h"
%}
tree = INSTANCE tree (int val);
%{
void f(void *x, int val) {
  *(int*)x += val;
}

int ttsum(tree *t) {
  int s = 0;
  trtree(f, &s, t);
  return s;
}
%}
```

This time we decided to follow the recursive sum-of-product pattern for the implementation of the binary tree, and interpret the tree instance as the following ADT:

```
tree = TREELEAF(int val)
     | TREEBRANCH(int val, tree *left, tree * right);
```

The extra function generated here is `trtree`, which performs an in-order traversal of the `subject` tree:

```
void trtree(void (*f) (void *, int), void *x,
                                     tree *subject) {
  switch(subject->tag) {
  case TREELEAF:
    f(x, subject->data._TREELEAF._val);
    break;
  case TREEBRANCH:
    trtree(f, x, subject->data._TREEBRANCH._left);
    f(x, subject->data._TREEBRANCH._val);
    trtree(f, x, subject->data._TREEBRANCH._right);
    break;
  }
}
```

The function `trtree` uses Currying to apply the function `f` to the value contained in every node of the tree. Since the actual function `f` adds the node value `val` it receives to `s`, the latter will contain the sum of all node values when the recursion terminates.

As we have seen in Table 1, this version of the sum program is slowed down a little due to the overhead of the calls to the Curried function `f`.

This concludes the discussion of the polymorphic built-in data types and the extra functions that they generate. We have shown that the Adt tool supports both the recursive sum-of-product pattern as well as the traditional C style with NULL pointers as sentinels.

It would not be difficult to extend the repertoire of built-in types, for example taking the other STL container classes as a source of inspiration. However, we suggest that this is done as future work.

4.4 C-style polymorphic functions

While it has been a deliberate design choice to avoid the use of `void *` as much as possible (the only exceptions are the functions with the `it` prefix for singly linked lists and the function with the `tr` prefix for binary trees), this has the drawback that user defined functions cannot be polymorphic. However, it is possible to build C-style polymorphism on top of the facilities provided by the Adt tool. For example creating a C-style "polymorphic" list could be achieved as follows:

```
voidlist = [void *v];
```

This creates the full complement of functions, including:

```
voidlist *mkVOIDLIST(void *_v, voidlist *_next);
void     *gtVOIDLISTv(voidlist *subject);
```

```
%{
#include "primitive.h"
%}

input   = INPUT(string header, body *body,
                                         string trailer);

body    = [ def *def ];
def     = DEF(string comment, ident *ident,
                         product *product, adt *adt);

adt     = ADT(sum *sum, string class,
                                product *parameters);

sum     = [ summand *summand ];
summand = SUMMAND(string comment, ident *ident,
                                   product *product);

product = [ factor *factor ];
factor  = FACTOR(string comment, int visibility,
             ident *type, string star, ident *field);

ident   = IDENT(string ident);
```

Figure 4: The data type specification of the Adt tool.

The `mkVOIDLIST` function stores any pointer type in the list, and `gtVOIDLISTv` retrieves the pointer again. There is no need for explicit type casts when these functions are used. No type errors will detected by the C compiler. Only the use of unique tags guarantees that at least at runtime, incorrect use of the elements of a void list is detected.

5 The Adt tool

Having described how the Adt tool is used, we will now describe two aspects of the tool itself. The first aspect is that the Adt tool has been used to build itself. The specification of the ADTs in the input syntax of the Adt tool is shown in Figure 4.

The specification introduces nine ADTs, `input`, `body`, `def`, `adt`, etc. We have already introduced the details of the notation, so here we just point out one design decision. Some types are pointers and others are not. The Adt tool gives the programmer control over this, whereas most other tools reviewed in the related work section assume that all fields of a product type are pointers. For example in ASDL [25] all primitive types are pointers. By exposing the representation of a field, our notation is less abstract but at the same time it provides the user with an appropriate level of control.

The YACC grammar of the Adt tool is shown in Figure 5. This shows how the functions such as `mkINPUT`, `mkBODY`, etc. generated by the Adt tool are used in the

```
input   : text body text
                { root = mkINPUT($1, $2, $3); }
        ;
text    : TEXT
        | /**/   { $$ = NULL; }
        ;
comment : COMMENT
        | /**/   { $$ = NULL; }
        ;
body    : def    { $$ = mkBODY($1, NULL); }
        | def ’;’{ $$ = mkBODY($1, NULL); }
        | def ’;’ body
                { $$ = mkBODY($1, $3); }
        ;
def     : comment ident ’=’ adt
                { $$ = mkDEF($1, $2, NULL, $4); }
        | comment ident ’=’ ’(’ product ’)’ adt
                { $$ = mkDEF($1, $2, $5, $7); }
        ;
adt     : sum    { $$ = mkADT($1, NULL, NULL); }
        | ’[’ product ’]’
                { $$ = mkADT(NULL, "list", $2); }
        | INSTANCESY TOKEN ’(’ product ’)’
                { $$ = mkADT(NULL, $2, $4); }
        ;
sum     : summand{ $$ = mkSUM($1, NULL); }
        | summand ’|’ sum
                { $$ = mkSUM($1, $3); }
        ;
summand : comment ident ’(’ product ’)’
                { $$ = mkSUMMAND($1, $2, $4); }
        | comment ident
                { $$ = mkSUMMAND($1, $2, NULL); }
        ;
product : factor { $$ = mkPRODUCT($1, NULL);}
        | factor ’,’ product
                { $$ = mkPRODUCT($1, $3); }
        ;
factor  : comment flags ident ’*’ ident
                { $$ = mkFACTOR($1,$2,$3,"*",$5); }
        | comment flags ident ident
                { $$ = mkFACTOR($1,$2,$3,"", $4); }
        ;
flags   : HIDDENSY
                { $$ = FACTOR_HIDDEN; }
        | SHORTCUTSY
                { $$ = FACTOR_SHORTCUT; }
        | STRUCTURALSY
                { $$ = FACTOR_STRUCTURAL; }
        | /**/   { $$ = FACTOR_STRUCTURAL; }
        ;
ident   : TOKEN  { $$ = mkIDENT($1); }
        ;
```

Figure 5: The Yacc specification of the Adt tool

semantic actions that create a parse tree. We note the uncluttered appearance of the semantic actions.

The second aspect concerns a number of convenient features that have not been touched upon yet.

- The `input` production rule shows that a `body` (which is a set of ADTs) can be preceded and followed by some `text`. This is used to a header and a trailer to the generated C code. For example the first three lines of Figure 4 `%{...%}` form a header, listing the include file to be used in the generated code. The trailer is empty in this case. Similarly, it is possible to begin the productions for `def`, `summand`, and `factor` with a comment, as illustrated by the occurrences of `/*...*/` in Figure 3.

- The Adt tool can generate a template traversal function for each ADT in the input (driven by the command line option `-t`). This is useful as a starting point for writing code. Unfortunately, any later changes in the ADT specification will have to be incorporated manually in the traversal functions that have been edited in the mean time.

- The Adt tool can also generate a LaTeX document for the ADTs (driven by the command line option `-l`), with comments appropriately edited. For example all occurrences of `%` (See Figure 3) are replaced by the identifier immediately following the comment.

We have tried to make the Adt tool as lean and mean as possible, implementing only those features that we felt were essential.

6 Qualitative evaluation

In this section we assess the strengths and weakness of the Adt tool using the Adt tool itself based on two real-world case studies. We look at fitting ADTs in a new program, at retrofitting ADTs into an existing program, at the avoidance of `void *` pointers, at NULL pointer checks, and at the use of NULL pointers in general.

The Adt tool itself represents a small case study. The ADT specification of Figure 4 comprises 22 lines (9 data types). This is expanded to 723 lines for the header file and 2054 lines (195 functions) for the code. About 18% of all the Adt code is generated by the tool itself, in which errors typical for humans (such as cut and paste errors) do not occur.

The Adt tool has been written such that the only occurrences of the `->` operator are in the code generated by the Adt tool. This ensures that any dereference of a NULL pointer is caught as early as possible by the run time checking code generated by the Adt tool. We found that the best way to use the Adt tool is with discipline. We offer two suggestions.

Firstly the ADTs are best designed using the recursive sum-of-product pattern. The design of the data types would naturally go hand in hand with the development of the Yacc grammar. If the semantic actions look uncluttered, as in Figure 5, the ADTs are probably well designed.

Secondly any code is best developed after the ADTs, again using the recursive sum-of-product pattern. Without this discipline, one soon discovers that changing the ADT causes significant changes in the interface functions, in turn requiring extensive editing to the code that uses the interface functions.

6.1 Fitting versus retro fitting

The two medium size case studies that we have are a System C compiler that was developed using the Adt tool from scratch, and a state-of-the-art commercial network intrusion detection system (NIDS) for which retrofitting ADTs was considered.

The SystemC compiler translates SystemC 1.0 to VHDL. The compiler has an ADT specification of 103 lines (16 data types), expanding to 3042 lines for the header file and 8884 lines for the code. The whole compiler consists of 14,760 lines of hand written Lex, Yacc, Adt and C code (thus excluding any machine generated code). If it had been written without the Adt tool, another 4,226 lines of C would have had to be written by hand which are now generated by the Adt tool. Again the machine generated code represents a significant fraction (22%) of the total amount of code that is free from trivial errors.

Writing the System C compiler from scratch using the Adt tool saved us work. We avoided writing the repetitive code for data structure access that the Adt tool generated for us, and as such it avoided us writing code in which trivial errors could hold us up.

The NIDS performs anomaly based network intrusion detection. The NIDS comprises 30,567 lines of C code with heavy pointer handling. There are 11 data structures held together by doubly linked lists of `void *` and trees of `void *`, and there are 6,756 uses of `->` in the NIDS code. Many of the `->` dereference to `void *` pointers that point to a data structure that is not a-priori known, and all of these could be NULL too.

We translated all the 11 data structures of the NIDS into Adt notation, using the polymorphic built-in doubly linked list and the polymorphic built-in tree. The NIDS makes list instances of four different types, so instead of one set of list access function, now we have four sets of machine generated access functions, each of which can be type checked by the C compiler. The NIDS makes one instance of a tree. In principle, this should get rid of many of the `void *` pointers in the code.

However, we have not been able to carry through the code renovation as, we struggled to try and replace all occurrences of `->` in the NIDS with the strongly typed access functions. Because many references are through `void *` pointers this replacement is non trivial, and each `->` will have to be analysed by hand to decide which of the four element types for the list is appropriate.

We conclude that retrofitting ADTs to existing source code can be a significant task, especially if code has been obfuscated by the use of `void *` pointers.

6.2 Avoiding Void Pointers

Studying the design of the NIDS highlights the use of `void *` for the implementation of sum types, and for the implementation of polymorphism. In many cases `void *` pointers can be avoided because only a limited set of instances is used, and unions with structs and tags could be used instead. Instead, `void *` pointers are often used because the use of unions and tags is too cumbersome to be done by hand. With tool support the situation changes.

We argue that code that does not use `void *` pointers is more transparent, and easier to understand and maintain. Especially when the code is maintained by people who were not part of the original implementation team, it can be virtually impossible to second guess `void *` pointers (similarly to using the `Object` class in Java 1.4 or earlier). Using tools such as Adt from the beginning would have avoided typing ambiguity, and avoided obfuscating the code.

6.3 NULL pointer checks

Code that is written using the Adt tool does not need to use the `->` operator. All occurrences are inside the code generated by the Adt tool. This ensures that any dereference of a NULL pointer is caught as early as possible by the run time checking code generated by the Adt tool.

For example the creators of the runtime analyzer Purify [20] observe that over 60% of the errors found by Purify are uninitialized memory read errors (caused essentially by a NULL pointer dereference). Similarly, the makers of the static analyzer PREfix [4] observe that more than 50% of all warnings reported by PREfix are NULL pointer dereferences.

The Adt tool wraps all pointer manipulation code in functions or macros that are instrumented with the same checks that Purify and PREfix implement to catch NULL pointer dereferences. Purify and PREfix implement further checks that the Adt tool does not provide, but these checks come at a considerable performance cost. For example Purify slows down execution by a factor of 5, whereas using the Adt tool may even speed up code.

We conclude that the Adt delivers some of the benefits of Purify or PREfix at no performance cost for projects where source code is created from scratch. Retrofitting ADTs on existing code is costly.

6.4 Avoiding NULL pointers as a programming method

We have already discussed the dangers of using NULL pointers in Section 2.3, and we have shown that the recursive sum-of-product pattern avoids using NULL pointers as sentinels: thinking about algebraic data types naturally leads one to think about the base case(s) and the recursive case(s) separately, which are then bundled in a sum type. As a result of this algebraic thinking, none of the 20 data structures in the SystemC compiler use NULL pointers as sentinels, whereas all of the 11 data structures in the NIDS do.

There are still many NULL pointers in C code generated by the Adt tool for the reason that the polymorphic built-in singly linked list is implemented using the traditional C style sentinel with a NULL pointer. This is a design choice that we made to show that Adt generated code does not have to be different from traditional C code. By contrast, our implementation of the built-in tree data type using the recursive sum-of-product pattern shows that in principle Adt generated code can be completely free from NULL pointers.

7 Related work

The literature provides a large variety of tools that support ADTs.

Firstly all declarative programming languages provide ADTs and pattern matching. Some object oriented languages, such as the Java derivative Pizza [18], Scala [17], offer the same powerful support for ADTs and pattern matching as found in declarative languages. In all these cases the compiler has to process the entire source code, whereas our approach requires the Adt tool to process only the ADT definitions. The TOM approach [16] is interesting because it represents an intermediary stage in between two extremes above: the TOM compiler only translates the type declaration and pattern matching constructs that are embedded in a host language, such as C or Java. However, this cannot be done without a special separate type checker, which thus makes the Tom approach more powerful, but at the same time less simple than ours.

Secondly, it is possible to use ADTs and pattern matching in a standard language, usually C++. For example McNamara and Smaragdakis [14] focus on the polymorphic typing issues and Läufer [13] focuses on the higher order issues. Standard C is not sufficiently powerful to support ADTs. The limits of what is possible in standard C are described by Ianello, who offers a programming discipline that achieves some of the advantages of working with ADTs [10]. The price to be paid is a severely reduced ability to type check programs.

Thirdly, there exists a large variety of tools dedicated to building abstract syntax trees for compilers. The Ast generator [8] and ASDL [25] focus on the development of intermediate representations that can be marshalled and unmarshalled in a variety of programming languages. The TTT tool [22] implements ADTs in C using runtime type checking, whereas our approach uses compile time type checking. The ApiGen tool [23] implements abstract syntax trees in Java, focusing on maximal sub-term sharing. By contrast, our Adt tool leaves decisions about sharing to the programmer. The tool proposed by Overbey and Johnson [19] focuses on rewriting abstract syntax trees for the purpose of code renovation. The work of de Jong and Olivier [6] focuses on the maintainability of the code generated for ADTs. By contrast, the Adt tool does not help with either code renovation or maintainability.

The main purpose of the ATerm library is the exchange of ADTs between programs written in languages such as C and Java [24]. The ATerm provides powerful pattern matching constructs and garbage collection. On the other hand it assumes that the ADTs are use only in a functional manner, hence typical C idioms that use destructive updates cannot be supported. By contrast the Adt tool allows destructive updates but requires the programmer to manage the storage.

The fourth category does not propose a specific method to support building abstract syntax trees. For example Smart C [11] is a type aware macro processor that can be used to rewrite fragments of C programs. This can be used to catch dereferencing of NULL pointers, which is one of the main advantages of the Adt tool.

The Adt tool is simpler and smaller than all of the above. The Adt tool provides only one thing, i.e. ADTs and pattern matching, and it does this in the simplest and most efficient way possible. From a specification of a set of ADTs the tool generates a `.h` and a `.c` file that provides the C programmer with the functionality needed to build and pattern match ADTs. There is no support for languages other than C, there is no support for marshalling and unmarshalling, automated sharing, or garbage collection. The Adt tool does not perform type checking, this is left to the C compiler. We believe that in the whole spectrum of possibilities, the Adt tool is as simple as possible.

8 Conclusions and Future work

The Adt tool is simple and effective. It does not provide the rich functionality of the tools discussed in related work, but it supports monomorphic, user-defined ADTs, pattern matching and polymorphic built-in types such as lists and trees. We have shown that the design choices that we have made lead to a simple, yet usable tool.

The Adt tool offers four styles of pattern matching on ADTs. The two styles that allow nested patterns are less efficient than the styles that allow flat patterns only. The most elegant style is also the fastest, but it does not offer nesting. The style that offers nesting with reasonable performance is the least elegant.

The Adt tool supports three polymorphic types that have been built into the tool. We argue that adding other polymorphic built-in types is probably as simple as implementing an STL container class for C++.

We describe the lessons learned while using the Adt tool to build a new compiler and to renovate an existing network intrusion detection system. The main conclusion is that C code developed with Adt can even be faster than traditional C code, and the code is likely to contain fewer errors than C code developed without the Adt tool.

The Adt tool is available online from `http://eprints.eemcs.utwente.nl/17771`. We welcome all contributions, but especially implementations of the C++ STL container classes in the form of polymorphic built-in types.

Acknowledgments

We thank the anonymous reviewers for their insights and helful comments. We are grateful to SecurityMatters BV for providing us with the data for the commercial NIDS case study.

References

[1] M. Barr and A. J. Massa. *Programming embedded systems: with C and GNU development tools.* O'Reilly Media, Inc., 2006. Available from World Wide Web: `http://oreilly.com/catalog/9780596009830`. 1

[2] R. S. Bird and P. L. Wadler. *Introduction to functional programming.* Prentice Hall, New York, 1988. 7

[3] F. P. Brooks. *The Mythical man month – Essays on software engineering.* Addison Wesley, Reading, Massachusetts, second edition, 1995. 1

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, Jun 2000. Available from World Wide Web: `http://dx.doi.org/10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H`. 11

[5] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer, third edition, 1987. 2

[6] H. A. de Jong and P. A. Olivier. Generation of abstract programming interfaces from syntax definitions. *J. of Logic and Algebraic Programming*, 59(1-2):35–61, Apr 2004. Available from World Wide Web: `http://dx.doi.org/10.1016/j.jlap.2003.12.002`. 12

[7] P. Deitel and H. M. Deitel. *C++ : How to Program.* Pearson Education, Upper Saddle River, New Jersey, seventh edition, 2010. Available from World Wide Web: `http://www.pearsonhighered.com/educator/product/C-How-to-Program/9780136117261.page`. 8

[8] J. Grosch and H. Emmelmann. A tool box for compiler construction. In *3rd Int. Workshop on Compiler Compilers (CC)*, volume 477 of *LNCS*, pages 106–116, Schwerin, Germany, Oct 1990. Springer. Available from World Wide Web: `http://dx.doi.org/10.1007/3-540-53669-8_77`. 12

[9] C. A. R. Hoare. Null references: The billion dollar mistake. In *Int. Sofware Development Conference (QCon)*, London, UK, Mar 2009. Unpublished. Available from World Wide Web: `http://www.qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake`. 3

[10] G. Iannello. Programming abstract data types, iterators and generic modules in C. *Software: Practice and Experience*, 20(3):243–260, Mar 1990. Available from World Wide Web: `http://dx.doi.org/10.1002/spe.4380200303`. 12

[11] M. Jacobs and E. C. Lewis. SMART C: A semantic macro replacement translator for C. In *6th IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 95–106. IEEE Computer Society, Sep 2006. Available from World Wide Web: `http://dx.doi.org/10.1109/SCAM.2006.28`. 12

[12] B. W. Kernighan and D. W. Ritchie. *The C programming language.* Prentice Hall, Englewood Cliffs, New Jersey, 1978. 3

[13] K. Laufer. A framework for Higher-Order functions in C++. In *Conf. on Object-Oriented Technologies (COOTS)*, page Article 8, Monterey, California, Jun 1995. Usenix Association. Available from World Wide Web: `http://www.usenix.org/publications/library/proceedings/coots95/laufer.html`. 12

[14] B. McNamara and Y. Smaragdakis. Functional programming in C++. In *5th ACM SIGPLAN Int. Conf. on Functional programming*, pages 118–129, Montréal, Canada, Aug 2000. ACM, New York. Available from World Wide Web: `http://dx.doi.org/10.1145/351240.351251`. 12

[15] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997. 2

[16] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *12th Int. Conf. on Compiler Construction (CC)*, volume 2622 of *LNCS*, pages 61–76, Warsaw, Poland, Apr 2003. Springer. Available from World Wide Web: `http://dx.doi.org/10.1007/3-540-36579-6_5`. 12

[17] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, Mountain View, California, 2007. Available from World Wide Web: `http://www.artima.com/shop/programming_in_scala`. 2, 12

[18] M. Odersky and P. L. Wadler. Pizza into Java: Translating theory into practice. In *24th Principles of programming languages (POPL)*, pages 146–159, Paris, France, Jan 1997. ACM, New York. Available from World Wide Web: `http://dx.doi.org/10.1145/263699.263715`. 12

[19] J. L. Overbey and R. E. Johnson. Generating rewritable abstract syntax trees - A foundation for the rapid development of source code transformation tools. In *A Foundation for the Rapid Development of Source Code Transformation Tools*, volume 5452 of *LNCS*, pages 114–133, Toulouse, France, Sep 2008. Springer. Available from World Wide Web: `http://dx.doi.org/10.1007/978-3-642-00434-6_8`. 12

[20] H. D. Owens, B. F. Womack, and M. J. Gonzalez. Software error classification using Purify. In *4th Int. Conf. on Software Maintenance (ICSM)*, pages 104–113, Monterey, California, Nov 1996. IEEE. Available from World Wide Web: `http://dx.doi.org/10.1109/ICSM.1996.564994`. 11

[21] A. Savidis. The implementation of generic smart pointers for advanced defensive programming. *Software: Practice and Experience*, 34(10):977–1009, Aug 2004. Available from World Wide Web: `http://dx.doi.org/10.1002/spe.600`. 4

[22] W. J. Toetenel. TTT - A simple type-checked C language abstract data type generator. In *R. Nigel Horspool*, volume IFIP Conf. Proceedings 117, pages 263–276, Berlin, Germany, Feb 1998. Chapman & Hall. 12

[23] M. van den Brand, P.-E. Moreau, and J. Vinju. Generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings: Software*, 152(2):70–78, Apr 2005. Available from World Wide Web: `http://dx.doi.org/10.1049/ip-sen:20041181`. 12

[24] M. G. T. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, Mar 2000. Available from World Wide Web: `http://dx.doi.org/10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y`. 12

[25] D. C. Wang, A. W. Appel, J. L. Korn, and C. S. Serra. The Zephyr abstract syntax description language. In *Conf. on Domain-Specific Languages (DSL)*, page Paper 17, Santa Barbara, California, Oct 1997. Usenix Association. Available from World Wide Web: `http://www.usenix.org/publications/library/proceedings/dsl97/wang.html`. 9, 12

Appendix

The two functions in Figure 6 represent the implementation of the polymorphic built-in type `tree`. The first function builds an ADT from an instance of a tree, and the second function generates the extra function with the `tr` prefix. Both functions are compact because they make heavy use of the functions generated by the Adt tool for the ADT of 4. The second function consists almost entirely of `printf` statements, indicating that code generation is simple indeed.

```
void instantiate_adt_tree(char *def_nm, adt *cur) {
  product *product1 = gtADTparameters(cur);
  ident * ident1 = mkIDENT(flipcase(def_nm, "leaf"));
  summand *summand1 = mkSUMMAND("", ident1, product1);

  factor *left2 = mkFACTOR("", FACTOR_STRUCTURAL, mkIDENT(def_nm), "*", mkIDENT("left"));
  factor *right2 = mkFACTOR("", FACTOR_STRUCTURAL, mkIDENT(def_nm), "*", mkIDENT("right"));
  product *product2 = func_approduct_3(gtADTparameters(cur), mkPRODUCT(left2, NULL), mkPRODUCT(right2, NULL));
  ident * ident2 = mkIDENT(flipcase(def_nm, "branch"));
  summand *summand2 = mkSUMMAND("", ident2, product2);

  sum *sum = mkSUM(summand1, mkSUM(summand2, NULL));
  stADTsum(cur, sum);
}

void high_level_instance_tree_traverse(char *def_nm, product *prod, adt *cur) {
  char *constr1_nm = gtIDENTident(gtSUMMANDident(gtSUMsummand(gtADTsum(cur))));
  char *constr2_nm = gtIDENTident(gtSUMMANDident(gtSUMsummand(gtSUMnext(gtADTsum(cur)))));
  product *combined = func_approduct(prod, gtADTparameters(cur));

  printf("void tr%s(void (*f) (void *, ", def_nm);
  formal_product(combined);
  printf("), void *x, %s *subject) {\n", def_nm);
  printf("\tcheck_ptr(subject, \"tr%s\");\n", def_nm);
  printf("\tswitch(subject->tag) {\n");
  printf("\tcase %s :\n", constr1_nm);
  printf("\t\tf(x, ");
  actual_product(constr1_nm, combined);
  printf(");\n");
  printf("\t\tbreak;\n");
  printf("\tcase %s :\n", constr2_nm);
  printf("\t\ttr%s(f, x, subject->data._%s._left);\n", def_nm, constr2_nm);
  printf("\t\tf(x, ");
  actual_product(constr2_nm, combined);
  printf(");\n");
  printf("\t\ttr%s(f, x, subject->data._%s._right);\n", def_nm, constr2_nm);
  printf("\t\tbreak;\n");
  printf("\t}\n");
  printf("}\n\n");
}
```

Figure 6: The complete support for polymorphic built-in trees.