Simple algebraic data types for C

Pieter Hartel and Henk Muller

April 6, 2010

Abstract

ADT is a simple tool in the spirit of Lex and Yacc that makes algebraic data types and a restricted form of pattern matching on those data types as found in SML available in C programs. ADT adds runtime checks, which make C programs written with the aid of ADT less likely to dereference a NULL pointer. The runtime tests may consume a significant amount of CPU time; hence they can be switched off once the program is suitably debugged.

1 Introduction

Brooks [2] advocates writing support tools to avoid repetitive and error prone work. A task that we have often encountered is the construction of the Algebraic Data Types (ADTs) 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, we present a simple tool called ADT, which, on the basis of a set of ADT specifications, generates C `struct` and function declarations that create and manipulate the corresponding concrete data structures. ADT is small and efficient, and it interworks well with Yacc.

2 Related work

The literature provides a large variety of tools that solve the problem at hand.

Firstly all declarative programming languages provide ADTs and pattern matching. Some object oriented languages, such as the Java derivative Pizza [10], offer the same powerful support for ADTs and pattern matching as found in declarative languages.

Secondly, it is possible to use ADTs and pattern matching in a standard language, usually C++. For example McNamara and Smaragdakis [8] focus on the polymorphic typing issues and Läufer [7] 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 [5]. 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 [4] and ASDL [14] focus on the development of intermediate representations that can be marshalled and unmarshalled in a variety of programming languages. The TTT tool [12] implements ADTs in C using runtime type checking, whereas our approach uses compile time type checking. The ApiGen tool [13] 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 [11] focuses on rewriting abstract syntax trees for the purpose of code renovation. The work of de Jong and Olivier [3] focuses on the maintainability of the code generated for ADTs. By contrast, ADT does not help with either code renovation or maintainability.

The fourth category does not propose a specific method to support building abstract syntax trees. For example Smart C [6] 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 purposes of ADT.

ADT is simpler and smaller than all of the above. ADT 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 the ADT the tool generates a `.h` and a `.c` file that provides the ADT user with the functionality needed to build and pattern match ADTs. There is no support for languages other than C, and there is no support for marshalling and unmarshalling, or automatied sharing. ADT does not perform type checking, this is left to the C compiler. ADT is thus simple.

3 ADT input

To illustrate how the ADT tool works we use the definition of the tool itself as a running example. The description of ADT in ADT is shown in Figure 1.

The specification introduces nine abstract data types, `input`, `deflist`, `def`, etc. The notation of ADT is inspired by functional programming languages, in particu-

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

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

deflist = [ def *def ];
def     = DEF(ident *ident, string header, adt *adt, string trailer);

adt     = ADT(sum *sum)
        | LIST(string header, product *product, string trailer
              %{ struct term_struct *_term; %});

sum     = [ term *term ];
term    = TERM(ident *ident, string header, product *product, string trailer);

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

ident   = IDENT(string ident);
```

Figure 1: The data type specification of the ADT tool.

lar Standard ML [9]. We focus on three of these type declarations in order to explain the notation.

First, the type `input` is a product type (i.e. a record in Pascal or struct in C). The *product* is over three fields that are separated by commas: `header`, `deflist`, and `trailer`. The types of the three fields are a `string`, a `deflist *` (i.e. a pointer to a `deflist`), and a `string` respectively. Most other tools reviewed in the related work section assume that all fields of a product type are pointers. For example in ASDL [14] the primitive types `identifier`, `int`, and `string` are represented by 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.

In Standard ML, the `input` data type would be declared as shown below (fields are separated by stars):

```
datatype input = INPUT of (string * deflist * string);
```

Second, the type `deflist` in Figure 1 is defined using square brackets to indicate that this type is a *list*. Inside the square brackets the types of the element is shown (a pointer to `def`) and the name of the element (`def`, separate name spaces denote types and fieldnames, so the same identifier may be used). Note that a list is strongly statically typed. The Standard ML equivalent would be:

```
type deflist = def list;
```

Third, the type `adt` in Figure 1 is a *disjoint sum* (tagged union) consisting of two product types, where the constructor names `ADT` and `LIST` are used to distinguish between the two summands. A vertical bar is used to indicate the summands. In Standard ML one would write:

```
datatype adt = ADT of (sum)
             | LIST of (string * product * string);
```

So compared to standard ML the fields must be named, since we want these fields to be accessible from standard C.

3.1 Escape mechanism

The definition of the `LIST` constructor in Figure 1 shows the use of an escape mechanism: the text `struct term_struct *_term;` will be include literally in the `struct` generated for the `adt` type. This can be used to attach information to a node, which is outside the control of the ADT tool. Another example of the escape mechanism is shown on top of Figure 1, where we specify which include files should be used in the generated C code.

3.2 Yacc grammar

The accompanying YACC grammar for the ADT tool is shown in Figure 2. This shows how the semantic actions `mkINPUT`, `mkDEFLIST`, etc generated by the ADT tool are used to create a parse tree. We note the uncluttered appearance of the semantic actions.

As an example consider the semantic actions for the non-terminal `deflist`, which creates a `NULL` terminated list of `def`. Lists are implemented as is usual in C, with a `NULL` terminated linked list of nodes.

```
input   : text deflist text
                { root = mkINPUT($1, $2, $3); }
      ;

text    : TEXT
        | /**/  { $$ = NULL; }
      ;

deflist : def   { $$ = mkDEFLIST($1, NULL); }
        | def ';'
                { $$ = mkDEFLIST($1, NULL); }
        | def ';' deflist
                { $$ = mkDEFLIST($1, $3); }
      ;

def     : ident '=' text adt text
                { $$ = mkDEF($1, $3, $4, $5); }


adt     : sum   { $$ = mkADT($1); }
        | '[' text product text ']'
                { $$ = mkLIST($2, $3, $4); }
      ;

sum     : term  { $$ = mkSUM($1, NULL); }
        | term '|' sum
                { $$ = mkSUM($1, $3); }
      ;

term    : ident '(' text product text ')'
                { $$ = mkTERM($1, $3, $4, $5); }
        | ident '(' text ')'
                { $$ = mkTERM($1, $3, NULL, NULL); }
        | ident { $$ = mkTERM($1, NULL, NULL, NULL); }
      ;

product : factor{ $$ = mkPRODUCT($1, NULL);}
        | factor ',' product
                { $$ = mkPRODUCT($1, $3); }
      ;

factor  : ident '*' ident
                { $$ = mkFACTOR($1, "*", $3); }
        | ident idents
                { $$ = mkFACTOR($1, "",  $2); }
      ;

ident   : TOKEN { $$ = mkIDENT($1); }
      ;
```

Figure 2: The Yacc specification of the ADT tool

```
tree   = LEAF(int val)
       | BRANCH(int val, tree *left, tree *right);
forest = [ tree *tree ];

fun sum (LEAF(val)) = val
  | sum (BRANCH(val, left, right))
                    = val + sum(left) + sum(right);
```

Figure 3: The SML data types and function used as specification of test programs.

4 ADT output

We now illustrate the code generated by ADT, using an ADT specification of a simpler example than Figure 1: the specification of a forest of binary trees.

4.1 Test program

Using ADT, we will implement the test program shown in Figure 3, which sums all the `val` fields in a `BRANCH` or a `LEAF`.

In the next section we will discuss the C type definitions generated by ADT for the `tree` and `forest` ADTs. This will then be followed by a description of the C functions and macros necessary to manipulate the `tree` in four different pattern matching styles.

4.2 C type definitions

The two `typedefs` below are generated for the type `tree`. No user defined code should access any of the fields directly; all access should be mediated by the interface functions discussed in the next section.

```
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 `struct` types. The ADT tool ensures that all tags are unique, which is handy for debugging. The `flag` field is use to avoid printing shared structures more than once (c.f. the `pr`... and `cl`... functions described below). As one of the main uses of the ADT tool is for building compilers, the `lineno`, `charno`, and `filename` fields can be used for generating error messages.

4.3 Functions

ADT generates two constructor functions for the `tree` data type. These are `mkLEAF` and `mkBRANCH` as shown below. The type and number of the arguments of the constructor functions correspond exactly to the type and number of the fields of the summands. The constructor functions use the `calloc` library function to allocate the data on the heap. Checks for heap overflow are compiled in automatically.

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

C does not provide garbage collection, hence there is a function to free the space allocated by the constructor function, and its descendants. In the case of the `tree` type this is the function `frtree`:

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

There are a number of debugging functions, such as `prtree`, which prints a nicely indented representation of the `tree` data structure. While the intended use of the data structure is a parse tree, it may sometimes be convenient to make it into a graph. Therefore the print function uses a flag in each node to remember whether the node has been printed. The family of clear functions, including `cltree` reset this flag.

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

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 fields in the product types there are get and set functions, i.e. `gtLEAFval` gets the `val` field of the product tagged with `LEAF`.

```
tree_tag gttreetag(tree *subject);
int      gtLEAFval(tree *subject);
void     stLEAFval(tree *subject, int value);
```

The main idea of the get functions is to make sure that any access to a NULL pointer is detected as early as possible, and to check that disjoint sum data types contain a value of the expected alternative. For example ADT generates for the fully protected access function `gtLEAFval` the following code:

```
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, but it will reduce the runtime (See Section 6).

4.4 Pattern matching functions

Pattern matching is supported by the functions with the `mt` and `pt` prefixes and the macro definitions with the prefix `in`.

```
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 `mt`... function compares both arguments, starting with the root of each. 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 on a match.

The patterns can be created on the heap by the family of `pt`... functions, or on the stack by the macros `in`...:

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

The `in`... macros are C99 style initialisers:

```
#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 `pt`... functions offer more flexibility, such as statically nested patterns, whereas the `in`... functions offer better performance.

Finally the ADT tool generates case style macros `cs`... as follows:

```
#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;
```

Four different usage models are provided for, which are discussed next.

4.4.1 Pattern matching in the stack

We can write C code that corresponds to the Standard ML code of Figure 3 by building two patterns on the stack that are matched against the tree. In the function `insum` below, the pattern `inLEAF(&val)` creates a `tree_struct` on the stack that stores the integer in a `LEAF` node in the local variable `val`. Similarly, when a match succeeds, the pattern `inBRANCH(&val, &leftbind, &rightbind)` stores the integer found in a `BRANCH` node in the local variable `val`, and it stores pointers to the left and right children in the local variables `left` and `right`, via the binders `intree(&left)`, and `intree(&right)`. Since the `in`... macros are implemented as initialisers, they cannot be nested syntactically, hence the rather verbose formulation at the beginning of the `else` branch.

```
int insum(tree *cur) {
  int val;
  struct tree_struct leaf = inLEAF(&val);
  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(&val,
      &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. A more elegant method stores a pattern on the heap.

4.4.2 Pattern matching on the heap

A pattern is stored on the heap by the `pt`... functions and freed again by the `fr`... functions. This makes it possible to write more elegant code. However, the run time cost is significant, as patterns have to be allocated and freed. (See Section 6).

```
int ptsum(tree *cur) {
  int val;
  tree *left, *right;
  tree *leaf = ptLEAF(ptint(&val));
  if(!mttree(leaf, cur)) {
    tree *branch = ptBRANCH(ptint(&val),
      pttree(&left), pttree(&right));
    if(mttree(branch, cur)) {
      val = val + ptsum(left) + ptsum(left);
    }
    frtree(branch);
  }
  frtree(leaf);
  return val;
}
```

The C function `ptsum` is reasonably elegant. It is possible, using the same pair of pattern and matching functions to create arbitrarily nested patterns. If this is not required, it is possible to produce even more elegant code, as we will see in the next section.

4.4.3 A pattern matching case statement

With the `cs`... macros it is possible to produce even more legible code, which is also the most efficient. However this only works on flat data structures since nested patterns are not possible:

```
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 canonical version of the sum function that uses access functions. This is the topic of the next section.

4.4.4 Using access functions

The straightforward access functions can be used instead, which avoids any notion of pattern matching and is more in style 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 above is a little cluttered with the many access functions, which can be avoided by pattern matching, as we have seen before. The previous version is, in our view, most comprehensible.

4.5 List functions

For the list ADTs, such as `forest`, the ADT tool generates two useful C functions. The first, `itforest`, iterates a function `f`, passed as an argument, over a list as shown below.

```
void itforest(void (*f) (void *, tree *),
              void *x, forest *subject) {
  while(subject != NULL) {
    f(x, subject->data._FOREST._tree);
    subject = subject->data._FOREST._next;
  }
}
```

This is the only place where `void *` pointers are used, to allow the iterate function to pass an extra argument (`x`) to the argument function `f`. This mechanism allows for the functionality of *currying*[1, Chapter 1] to be implemented.

An example use would be a function that calls `cssum` on all trees in a forest, and sums the result in an integer pointer. The calling function initialises the integer pointer to point to zero, and returns the result on completing the iteration:

```
void sum(void *x, tree *t) {
    *(int*)x += cssum(t);
}

int sumall(forest *f) {
  int s = 0;
  itforest(sum, &s, f);
  return s;
}
```

The second list processing function allows appending two lists such that the first list `subject` is copied, and the second list `object` is shared.

```
forest *apforest(forest *subject, forest *object);
```

4.6 Polymorphism

While it has been a deliberate design choice to avoid the use of `void *` as much as possible (the only exception is the `it`... function family), this has the drawback that no form of polymorphism can be supported directly by the ADT tool. However, it is possible to build C-style polymorphism on top of the facilities provided by ADT. For example creating a ‘polymorphic’ list could be done as follows:

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

This creates the full complement of functions and typedefs, for example:

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

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. However, the use of unique tags guarantees that at least at runtime, incorrect use of the elements of a void list is detected. Note that this list implements dynamic type checks and it can store elements of different types without the use of a disjoint sum type.

5 Usage

The ADT tool expects an input file with extension `adt`. The program has a number of options as follows:

-d prints a readable, indented parse tree on `stdout`.

-h outputs a header file with the typedefs and prototypes. The file name is the same as the input filename, but with the extension `h`.

-c outputs the function definitions to a file with the same as the input filename, but with extension `c`.

-t generates template traversal functions for the entire collection of ADTs in the input. The output is written to `stdout`. This is useful as a starting point for writing functions that traverse the parse tree. Unfortunately, any later changes in the ADT specification will have to be incorporated manually in the traversal functions.

-l Generates a latex document for the ADTs, with comments (represented by the token `text` in Figure 2) appropriately formatted.

legend	`insum`	`ptsum`	`cssum`	`gtsum`
time	152±1 ns	1717±2 ns	16.8±0.1 ns	17.7±0.1 ns
% time	8%	100%	0.98%	1.03%
elegance	-	+/-	+	+/-
nesting	+	+	-	-

Table 1: Performance and elegance of the four styles of pattern matching.

6 Experience

The ADT specification of Figure 1 comprises 20 lines (9 data types). This is expanded to 307 lines for the header file and 1244 lines (155 functions) for the code. The ADT tool actually only uses 30% of the generated functions.

We have also built a compiler from SystemC 1.0 to VHDL, which has an ADT specification of 103 lines (16 data types), expanding to 845 lines for the header file and 4125 lines for the code.

Both the SystemC compiler and the ADT tool have been written such that the only occurrences of the C "`->`" 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 have found this to be helpful, especially in developing our SystemC compiler.

The best way to use the ADT tool is with discipline, in the sense that the algebraic data types must be designed before any of the code can be developed. The design of the data type would naturally go hand in hand with the development of the Yacc grammar. 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.

The performance of the generated code depends on the style of pattern matching used. The average run time (on a 1.2 GHz PC, with Cygwin and GCC) in ns per call to the `sum` function of the four versions of the test program from Section 4.4 is shown in Table 1.

The third version `cssum`, which uses the `cs`... style macros is the most efficient, closely followed by the fourth version `gtsum`. The reason for the small difference in performance is that the former checks the tag only once, the latter checks the tag more than once. The versatility of nested patterns comes at a high price. The version `insum`, which allocates patterns on the stack, is about 10 times slower and the version `ptsum`, which allocates patterns on the heap is about 100 times slower than `cssum`.

The two versions `cssum` and `gtsum` can be made a little faster by switching off the checks for NULL pointers. The speed improvement depends on the optimisation settings of the GCC compiler. With `-O3` there is no significant difference due to the aggressive optimisations. With the default setting of the optimiser, `gtsum` is about twice as fast when the NULL pointer checking is turned off. Given that the performance penalty is low with the optimiser on, we believe that it is probably a good idea to keep NULL pointer checking activated.

7 Conclusion

ADT is simple and effective. It does not provide the rich functionality of the tools provided in related work, but we believe this to be a strength. ADT offers four styles of pattern matching. 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.

ADT offers a clean separation between specification of the algebraic data type and the implementation in C. Firstly, there is no need to include C code in the ADT specification. Secondly, the generated C code obeys a few simple rules, for example (1) for every algebraic data type the same family of constructor and access functions is generated, and (2) the signature a constructor function is exactly the same the signature of a summand.

ADT catches all unexpected NULL pointer dereferences.

ADT is available online from `http://eprints.eemcs.utwente.nl/17771`.

References

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

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

[3] 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: `http://dx.doi.org/10.1016/j.jlap.2003.12.002`.

[4] 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: `http://dx.doi.org/10.1007/3-540-53669-8_77`.

[5] G. Iannello. Programming abstract data types, iterators and generic modules in C. *Software: Practice and Experience*, 20(3):243–260, Mar 1990.

Available from: http://dx.doi.org/10.1002/spe.4380200303.

[6] 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: http://dx.doi.org/10.1109/SCAM.2006.28.

[7] 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: http://www.usenix.org/publications/library/proceedings/coots95/laufer.html.

[8] 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: http://doi.acm.org/10.1145/351240.351251.

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

[10] 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: http://doi.acm.org/10.1145/263699.263715.

[11] 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: http://dx.doi.org/10.1007/978-3-642-00434-6_8.

[12] 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.

[13] 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: http://dx.doi.org/10.1049/ip-sen:20041181.

[14] 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: http://www.usenix.org/publications/library/proceedings/dsl97/wang.html.