Processing ASTs in C++: `maketea`

Edsko de Vries*, John Gilbert, and David M Abrahamson

Trinity College Dublin

Abstract. We present `maketea`, a tool which generates a C++ infrastructure for processing ASTs based on an object oriented context free grammar. The generated code includes a class hierarchy for storing ASTs with support for cloning, equality checking, pattern matching, a general visitor API and a transformation API. The tool is available under an open source license and can be downloaded from `www.maketea.org`.

1 Introduction

Given the definition of an abstract syntax tree in the form of a context free grammar, `maketea` creates a C++ class hierarchy for storing AST instances and an API for processing them. This is extremely useful when developing compiler front ends, and indeed `maketea` was developed with exactly such an application in mind [1]. Manually maintaining the data structures and corresponding operations on them in the face of an evolving grammar is both time consuming and error prone. In this paper we highlight the most important features of `maketea` and discuss some of the challenges encountered during its design.

The grammar formalism used by `maketea` is an extension of what is sometimes referred to as an "object oriented context free grammar" [2]. The key principle with such grammars is that there be a one-to-one mapping from the productions to the class definitions. Only two types of rules are permitted:

- $a ::= b \mid c \mid \ldots$, with $b, c, \ldots$ single (terminal or non-terminal) symbols
- $a ::= b^m\, c^n \ldots$, with $b, c, \ldots$ single symbols, and $m, n, \ldots$ multiplicities

A rule of the first form (a disjunction) models the *is-a* relation and gets mapped to an abstract class. The classes that correspond to the symbols in the body of the disjunction will all inherit from this class (multiple inheritance is allowed and frequently useful). On the other hand a rule of the second form (a sequence) gets mapped to a concrete class, with class data members for each of the terms in the body of the rule. We distinguish between five different multiplicities: single, optional (?), list ($*$), optional list ($*?$) and list of optional elements ($?*$), where lists are implemented using C++ STL lists.

All classes generated by `maketea` support deep equality checking and deep cloning. Moreover, the grammar specification can include C++ code fragments which can override any generated functionality. Finally, to support interoperability amongst tools, maketea generates an XML schema that can be used to validate serializations of the AST in XML format and a factory method which can be used to construct DOM based XML parsers.

* Supported by the Irish Research Council for Science, Engineering and Technology

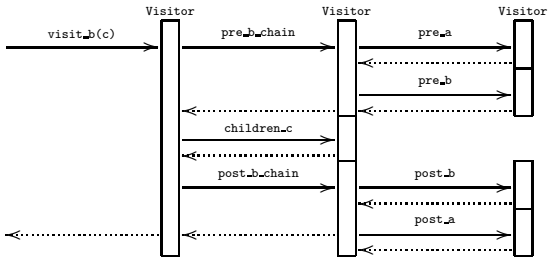

Fig. 1. Sequence Diagram for the Visitor API

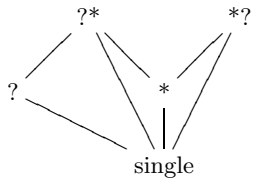

Fig. 2. Partial ordering on multiplicities

2 Visitor API

The visitor API is a generalization of the well-known visitor pattern. We create an identity visitor that can traverse arbitrary ASTs and calls different methods for the various types of nodes it encounters. Specific traversals are defined by inheriting from the identity visitor and overriding these methods. The visitor API generated by `maketea` differs from the basic pattern in the following aspects:

- The API simultaneously supports pre-order and post-order traversals: before the children of a node are visited, a "pre-visit" method is invoked; and after the children have been visited, a "post-visit" method is invoked.
- The pre-visit and post-visit methods for a node invoke a *chain* of methods, one for each of its superclasses. Thus generic visitors can be defined that for instance visit all "statements" in an AST, where "statement" is the disjunction of "if", "while", etc. Fig. 1 shows an example sequence diagram for a call to `visit_b` on a node of a type B that inherits from A.
- All of the visitor's functionality is embedded in the visitor class rather than in the AST classes. This makes it possible to change the behaviour of the visitor on a *per visitor* basis. For example, it is possible to override the order in which the children of a particular node type are visited, or the order in which the visitor methods of the superclasses of a node type will be invoked.

Finally, the visitor API has explicit support for unparsers. For example, rather than omitting a call to a visitor method for NULL nodes, we invoke a special `visit_null` method. One application where this is useful is XML serialization, where the absence of an optional element must be explicitly indicated.

3 Pattern Matching

Pattern matching is a variation on deep equality checking where parts of the tree that we are comparing against are left unspecified using wildcards. Pattern matching provides a mechanism for recognizing larger parts of the tree of a specified form. For example, in an AST denoting expressions we can easily define a pattern to match the action of adding two constants. To support pattern matching, `maketea` introduces a new class `Wildcard` which is defined as

```
#template<class C>
class Wildcard : public virtual C, public __WILDCARD__ {
    C* value;
    // ...
}
```

Since `Wildcard<X>` inherits from `X`, a node of type `Wildcard<X>` can be used in place of a node of type `X`. Pattern matching proceeds like a deep equality check, but whenever a node of type `__WILDCARD__` is encountered the `value` field of the wildcard is made point to the subtree it matched. After pattern matching completes, the values of the wildcards are easily accessed and since the pattern itself has not been modified it can be reused in a later match.

4 Transform API

The visitor API has one important disadvantage: visitors cannot easily modify the structure of the tree. Suppose we wanted to apply constant folding. A visitor can visit all binary operator nodes in a tree, but cannot replace an operator by a different node. Instead, it would have to visit all nodes *containing* binary operators (the parent nodes), and do the constant folding there. This means that an optimization that should only have had to be defined for a single type of node (binary operators) must now be defined for a large numbers of nodes: all nodes that contain binary operators.

To overcome these difficulties, `maketea` introduces a transformation API alongside the visitor API. The basic setup is very similar: we generate an identity transformation, which traverses the tree and invokes transformation methods for each node. Specific transformations are implemented by inheriting from the identity transformation and overriding the appropriate transformation methods.

The API constrains the type of node B a node of type A can be replaced with. Using a process called *context resolution*, `maketea` will find the most general type B such that it is always correct to replace a node of type A by one of type B while keeping the tree consistent with the grammar. This proceeds in two steps

1. In the first step an initial set of contexts is extracted from the grammar.
   (a) When a concrete symbol c is used with multiplicity m (in the body of a sequence rule) it yields a context (c, c, m). That is, we can only replace a c by a c, but the multiplicity determines how many cs we can replace a single c with.
   (b) When an abstract symbol c is used with multiplicity m, it yields contexts (c', c, m) for all symbols c' that inherit from c.
2. In the second step we reduce the initial set of contexts so that there is a single context for every symbol in the grammar. For contexts (c, d, m) and (c, d', m') in the initial set we construct a new context (c, d'', m'') where d'' is the unique[1] common subclass of both d and d' that is also a superclass of c, and m'' is the greatest lower bound of m and m' using the partial order shown in Fig. 2.

[1] Such a class must always exist since, in the worst case, it is c itself.

For example, if this is part of a grammar:

```
start ::= A b? b ;
b     ::= C | D ;
```

then the derived context for `A` will be `(A,A,single)` because there is an explicit reference to `A` in the grammar (step $1a$), while two contexts will be inferred for `C` (step $1b$): `(C,b,?)` and `(C,b,single)`, which get reduced to `(C,b,single)`.

5 Conclusions

We presented `maketea`, a tool which generates a C++ infrastructure for representing and processing ASTs based on an object oriented context free grammar. Although many other tools such as `classgen`, `ANTLR`, `JastAdd`, `jjtree`, `GENTLE`, `MEMPHIS` and `Zephyr` provide similar functionality, they are less versatile than `maketea` in several respects. The grammar formalism may be more restrictive; for example, it may not support multiple inheritance or accept multiplicities. While most tools provide a visitor API, they usually cover only the basic pattern without support for the more advanced features described in Section 2. Few support pattern matching, and we are not aware of any tools that support a transformation API comparable to the one we provide.

Because `maketea` is written in Haskell, the tool is relatively small (just over 3000 lines of code) despite the comprehensive API it provides. The small code base makes it feasible to quickly adapt `maketea` for individual projects when the functionality provided out of the box is insufficient.

We designed `maketea` with a specific project in mind (`phc` [1]), and believe we saved a significant amount of time in the development of that project even allowing for the time spent developing `maketea` itself: `phc` currently includes two `maketea` specifications which together account for some 900 lines of code. When run through `maketea` these expand to approximately 32,000 lines of C++ code. Maintaining that C++ code by hand during the development of `phc` while the grammars were not yet completely fixed, would have been nearly impossible. Various versions of `maketea` have been used during the development of `phc` since early 2006, and it has now reached a point where it is suitably robust for use in other projects.

References

1. de Vries, E., Gilbert, J.: Design and implementation of a PHP compiler front-end. Dept. of Computer Science Technical Report TR-2007-47, Trinity College Dublin
2. Koskimies, K.: Object-orientation in attribute grammars. In: Proceedings on Attribute Grammars, Applications and Systems, London, UK, Springer-Verlag (1991) 297–329

Appendix: Demonstration

The `maketea` demonstration will proceed as outlined below[2].

High level introduction to maketea Briefly describe–with aid of slides–the problem `maketea` solves, and introduce the functionality it generates: the Visitor, Transformation and Pattern matching APIs. This will provide preliminary background for the rest of the demonstration.

Grammar formalism Show by way of demonstration the format of a `maketea` specification. The running example introduced is a small programming language called *Chai*, defined by the `maketea` grammar shown below:

```
program    ::= statement* ;
statement  ::= assignment | while | print ;
assignment ::= literal | add ;
while      ::= VAR statement* ;
literal    ::= lhs:VAR INTEGER<long> ;
add        ::= lhs:VAR left:VAR right:VAR ;
print      ::= VAR ;
```

The following *Chai* program is then introduced, which will be used as an example input to all subsequent visitors and transformations in the tutorial:

```
y := -1;
x := 3;
while x {
  print x;
  x := x + y;
}
```

Visitor API: Pretty Printer A unified pre-order/post-order traversal of the AST will be demonstrated. This is useful for unparsing `while` statements (curly braces can be problematic without this unified approach). This is implemented in Fig. 5, and simply reproduces the input program. This shows our first deviation from the standard Visitor design pattern which tends to support either pre-order or post-order traversal, but not both simultaneously.

Visitor API: XML Unparser The second difference between the canonical Visitor pattern and our implementation is the ability to use a *chain* of methods while processing each concrete node in the AST. This is useful for writing generic traversals as will be demonstrated in our XML unparser (Fig. 3), which is easily implemented. The result of running this Visitor on our sample *Chai* program is shown in Fig. 6.

Visitor API: Evaluator Our visitor implementation does not embed the traversal logic within the concrete AST nodes, but rather within the Visitor itself. This gives great flexibility, as demonstrated in our Evaluator example (Fig. 7), where the body of a *while* node can be repeatedly visited until its condition fails. The output of evaluating our sample *Chai* program is as follows:

[2] Note that the figures are not shown in order to optimize the available space in this overview.

```
x: 3
x: 2
x: 1
```

Transformation API: Debugger Our transformation API is demonstrated by instrumenting the simple *Chai* program with *print* statements after each *add* statement (Fig. 8). Although simple, this transformation demonstrates the issue of resolving contexts for the Transform API, and shows in practical terms the reduction in code required to implement the transformation when compared with implementing the same task using a Visitor. If a Visitor were used to implement the transformation, both *program* and *while* nodes would need to be dealt with separately, while a Transform requires the implementation for *add* nodes only. The result of the transform is as follows:

```
y := -1;
x := 3;
while x {
  print x;
  x := x + y;
  print x;
}
```

```
class XML_unparser : public AST_visitor {
  void pre_node(AST_node* node) {
    cout << "<" << demangle(node) << ">" << endl;
  }

  void post_node(AST_node* node) {
    cout << "</" << demangle(node) << ">" << endl;
  }

  void pre_var(Token_var* var) {
    cout << *var->value << endl;
  }

  void pre_integer(Token_integer* integer) {
    cout << integer->value << endl;
  }
};
```

Fig. 3. XML unparser

```
// statement ::= assignment | while | print;
class AST_statement : virtual public AST_node
{
public:
  AST_statement();
public:
  void visit(AST_visitor* visitor) = 0;
  void transform_children(AST_transform* transform) = 0;
public:
  int classid() = 0;
public:
  bool match(AST_node* in) = 0;
public:
  bool equals(AST_node* in) = 0;
public:
  AST_statement* clone() = 0;
public:
  void assert_valid() = 0;
};

// while ::= VAR statement* ;
class AST_while : public AST_statement
{
public:
  AST_while(Token_var* var,
            List<AST_statement*>* statements);
protected:
  AST_while();
public:
  Token_var* var;
  List<AST_statement*>* statements;
public:
  void visit(AST_visitor* visitor);
  void transform_children(AST_transform* transform);
public:
  static const int ID = 2;
  int classid();
public:
  bool match(AST_node* in);
public:
  bool equals(AST_node* in);
public:
  AST_while* clone();
public:
  void assert_valid();
};
```

Fig. 4. Generated C++ classes for `statement` and `while`

```
class Unparser : public AST_visitor
{
protected:
  string indent;

public:
  void post_literal(AST_literal* literal)
  {
    cout << indent
         << *literal->lhs->value << " := "
         << literal->integer->value << ";" << endl;
  }

  void post_add(AST_add* add)
  {
    cout << indent
         << *add->lhs->value << " := "
         << *add->left->value << " + "
         << *add->right->value << ";" << endl;
  }

  void post_print(AST_print* print)
  {
    cout << indent
         << "print " << *print->var->value << ";" << endl;
  }

  void pre_while(AST_while* wh)
  {
    cout << indent
         << "while " << *wh->var->value << " {" << endl;
    indent.push_back('\t');
  }

  void post_while(AST_while* wh)
  {
    indent = indent.substr(1);
    cout << indent << "}" << endl;
  }
};
```

Fig. 5. Unparser (pretty printer)

```
<AST_program>
<AST_literal>
<Token_var>
y
</Token_var>
<Token_integer>
-1
</Token_integer>
</AST_literal>
<AST_literal>
<Token_var>
x
</Token_var>
<Token_integer>
3
</Token_integer>
</AST_literal>
<AST_while>
<Token_var>
x
</Token_var>
<AST_print>
<Token_var>
x
</Token_var>
</AST_print>
<AST_add>
<Token_var>
x
</Token_var>
<Token_var>
x
</Token_var>
<Token_var>
y
</Token_var>
</AST_add>
</AST_while>
</AST_program>
```

Fig. 6. XML output from the visitor in Fig. 3

```
class Eval : public AST_visitor {
  map<string, long> env;

  void pre_literal(AST_literal* literal) {
    env[*literal->lhs->value] = literal->integer->value;
  }

  void pre_print(AST_print* print) {
    cout << *print->var->value << ": "
         << env[*print->var->value] << endl;
  }

  void pre_add(AST_add* add) {
    env[*add->lhs->value] =
      env[*add->left->value] + env[*add->right->value];
  }

  void children_while(AST_while* wh) {
    while(env[*wh->var->value])
      AST_visitor::children_while(wh);
  }
};
```

Fig. 7. Evaluator

```
class Add_prints : public AST_transform {
  void pre_add(AST_add* in, List<AST_statement*>* out) {
    out->push_back(in);
    out->push_back(new AST_print(in->lhs->clone()));
  }
};
```

Fig. 8. Insert instrumentation for debugging