TWEAST: A Simple and Effective Technique to Implement Concrete-Syntax AST Rewriting Using Partial Parsing

Akim Demaille Roland Levillain[*] Benoît Sigoure
EPITA Research and Development Laboratory (LRDE)
14/16, rue Voltaire, F-94276, Le Kremlin-Bicêtre, France
akim@lrde.epita.fr roland@lrde.epita.fr sigoure@lrde.epita.fr

ABSTRACT

Abstract Syntax Trees (ASTs) are commonly used to represent an input/output program in compilers and language processing tools. Many of the tasks of these tools consist in generating and rewriting ASTs. Such an approach can become tedious and hard to maintain for complex operations, namely program transformation, optimization, instrumentation, etc. On the other hand, *concrete syntax* provides a natural and simpler representation of programs, but it is not usually available as a direct feature of the aforementioned tools. We propose a simple technique to implement AST generation and rewriting in general purpose languages using concrete syntax. Our approach relies on extensions made in the scanner and the parser and the use of objects supporting partial parsing called Texts With Embedded Abstract Syntax Trees (TWEASTs). A compiler for a simple language (Tiger) written in C++ serves as an example, featuring transformations in concrete syntax: syntactic desugaring, optimization, code instrumentation such as bounds-checking, etc. Extensions of this technique to provide a full-fledged concrete-syntax rewriting framework are presented as well.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Parsing, Translator writing systems and compiler generators*

General Terms

Design, Languages

Keywords

C++, Parsing, Concrete Syntax, Program Transformation, Rewrite Rules, Compiler Design

[*]Also with Université Paris-Est, LABINFO-IGM, UMR CNRS 8049, A2SI-ESIEE, Cité Descartes, BP 99, FR-93162 Noisy-le-Grand Cedex, France

1. INTRODUCTION

Compilers typically feature three parts: the *front-end* analyzes the input program, the *middle-end* is independent of the source and target languages, and the *back-end* generates the output program. This paper focuses on the front-end. It deals with an *Abstract Syntax Tree (AST)*, a tree-structured representation of the source. Inner nodes correspond to the various language constructs (`while` loops, assignments, etc.) and leaves are "meaningful" terminal symbols (e.g., the literals, but not punctuation, comments or other layout details). In Object-Oriented Programming (OOP), ASTs are implemented as a simple class hierarchy (Figure 1).

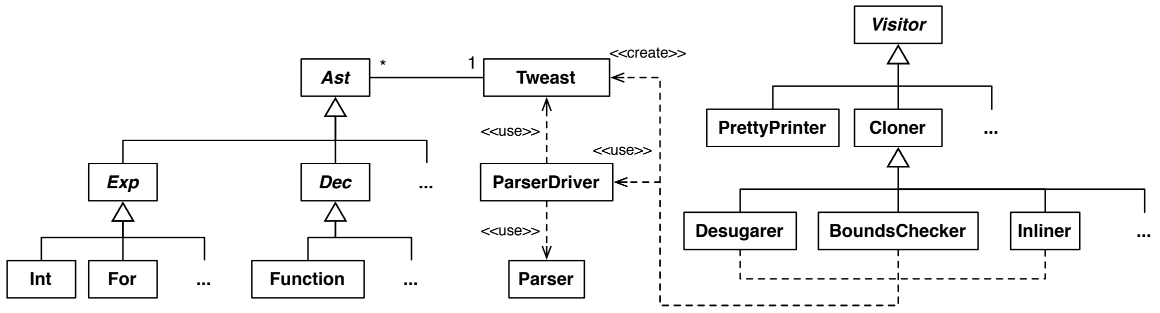
Transformations are often difficult to implement: the rewrite rules are expressed in abstract syntax — the syntax of the transforming language, not the transformed one. Consider the following production rule from the grammar formatted for the Bison [4] parser generator, corresponding to the Boolean expression '*exp* `&` *exp*'.

```
/* Boolean and operators.   We code 'A & B' as
   'if  A then B <> 0 else 0'.   */
exp: exp "&" exp
  {
    /* '$1' and '$3' hold the semantic values associated to
       respectively the first and second 'exp' symbols of the
       right-hand side of the production, i.e. ASTs corresponding
       to the operands of '&'.  '$$' is the result.  */
    $$ = new If($1,
                new Op($3, Op::NotEqual, new Int(0)),
                new Int(0));
  };
```

Instead of building an AST using a dedicated node (e.g., `new And($1, $3)`), this rule uses other (existing) nodes of the abstract language : `If` ("functional" `if-then-else` expression), `Op` (binary comparison/arithmetical expression) and `Int` (literal integer value). Operators like `&` can indeed be considered as *syntactic sugar* on top of the *core language*, a subset of the full language composed of primitive constructs. By translating these complex constructs into more primitive ones, we remove the syntactic sugar or *desugar* the program to the core language.

Put differently, the semantic action of the previous production is a *program transformation* directly expressed in the parser: a substitution of an abstract syntax subtree pattern (its leaves being replaced by labels called *metavariables*), by another subtree pattern, as depicted on Figure 2.

However, despite the desugaring approach is worthy, the previous implementation is tedious, error prone and makes the compiler or processing tool hard to maintain: abstract syntax clutters the transformation. In this paper, we show how run-time *concrete-syntax* rewrite rules can be implemented in general purpose languages (such as C++) using



Front-end tasks dealing with an AST are implemented as visitors. The VISITOR design pattern [7] is a convenient implementation of traversals that decouples the algorithm (the traversal) from the data (the AST). They all inherit from the *Visitor* abstraction. Concrete syntax is written in visit methods thanks to Tweast objects holding both Tiger sentences and sub-ASTs. The PARSERDRIVER is then invoked to produce an AST.

Figure 1: Some of the classes involved in transformations in run-time concrete syntax.

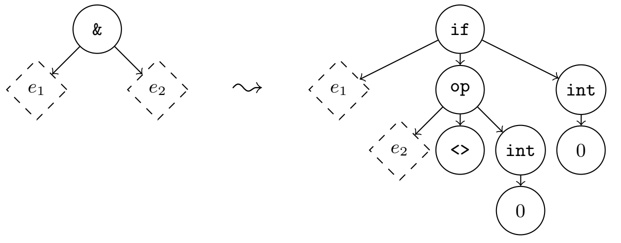


Figure 2: An abstract syntax transformation of an & (Boolean and) expression. e_1 and e_2 are metavariables representing expressions.

regular tools. Run-time concrete syntax is supported thanks to a new idiom which we name Text With Embedded Abstract Syntax Trees (TWEAST). This method relies on extensions in the scanner and parser to embed existing ASTs in concrete syntax which is parsed at run-time.

As a proof of concept, we implemented the techniques presented here in a pedagogic compiler [5], written in C++, for the Tiger language [1, 6]. We re-implemented various transformations of our Tiger compiler in order to simplify and shorten the code, so as to reduce the maintenance cost and the potential number of bugs.

This paper is structured as follows: Section 2 introduces the concept of TWEAST and partial parsing; implementation details are exposed in Section 3; Section 4 presents several applications of our technique, and results and related work are discussed in Section 5. Section 6 reports on-going work and extensions of TWEAST and Section 7 concludes.

2. TWEASTS AND PARTIAL PARSING

In language manipulation tools, conversion from concrete syntax to abstract syntax (AST) is performed by a syntactic analyzer or *parser*. Conversely, abstract-to-concrete transformation is the task of a *pretty-printer*. A first step toward using concrete syntax in rewrite rules is to make use of the parser and the pretty-printer to avoid explicit AST node instantiations and manipulations to generate the rewritten tree. For instance the transformation from Section 1 can be rewritten as:

```
exp: exp "&" exp
  {
    std::ostringstream s;
    // Sub−ASTs '$1' and '$3' are printed back to concrete
    // syntax, and concatenated to the other strings into 's'.
    // Spaces are needed not to transform 'a&b' into 'ifathenb ...'.
    s << "if " << $1 << " then " << $3 << "<> 0 else 0";
    // Then the whole string is parsed.
    $$ = parse(s.str ());
  };
```

Here, we rely on the AST pretty-printer to turn each sub-AST into a string (each AST node is equipped to support its conversion to concrete syntax with `operator<<`). This approach is algorithmically inelegant, inefficient (some parts of the sources are possibly parsed many times), and error-prone (care must be taken with spaces, operator precedences and so forth). Besides, annotations such as bindings, types, etc. are lost.

A solution avoiding the extra costs of parsing and pretty-printing while preserving the concrete syntax feature is to use an object recording both the string (which may include *metavariables*, labeled placeholders for sub-ASTs), and the sub-ASTs.

In other word, this object is used to represent a state of partial parsing: in the previous code sample, sub-ASTs '`$1`' and '`$3`' are the product of previous parsings while the "piecewise" string '`"if" (...) "then" (...) "<> 0 else 0"`' is not-yet-parsed text. We name this object *Text With Embedded Abstract Syntax Trees (TWEAST)*. It can be used like the stream of the previous example:

```
exp: exp "&" exp
  {
    // Sub−ASTs are used as−is (they are not printed, then parsed,
    // as in the previous example). Spaces are no longer critical.
    $$ = parse(Tweast() <<
           "if" << $1 << "then" << $3 << "<> 0 else 0");
  };
```

Concrete-syntax rewriting can be used in several places:

Within the parser. As seen on the examples of this section, rewriting can be turned into a *generation* mechanism to replace explicit and unreadable AST instantiations by concrete-syntax patterns.

After an AST node or tree has been matched. This is the case when an AST pattern is searched in a whole tree, to be rewritten to something else. Our Tiger compiler features many AST traversals implemented as VISITORS [7]. One of them, the `Desugarer`, removes the syntactic sugar. For instance, operator-based string comparisons are translated into calls to the function `strcmp`:

```
void Desugarer::operator() (const Op& e)
{
  // Desugar 'a < b' as 'strcmp(a, b) < 0', and so on.
  if (e.op().kind() == Comparison
      && e.lhs().type() == String)
    {
      result_ = parse(Tweast() <<
        "strcmp(" << recurse(e.lhs()) << ","
                  << recurse(e.rhs()) << ")"
                  << e.op() << "0");
    }
  // ...
}
```

(In Tiger, comparison operators are overloaded.) See Section 4 for more examples.

After a *concrete-syntax pattern* is matched. In the previous examples, only the *built* pattern of a rewrite rule uses concrete-syntax features; the *matched* pattern either relies on a production rule of the parser or on an abstract-syntax pattern. A full concrete-syntax rewrite engine uses the concrete syntax for *both* the building *and* the matching parts of the rewriting. Such a facility allows the expression of rules in a concise and simple manner, as in this example:

$$\underbrace{x\ \overbrace{\texttt{::string}}^{\text{type annotation}} < y\ \overbrace{\texttt{::string}}^{\text{type annotation}}}_{\text{match pattern}} \rightarrow \underbrace{\texttt{strcmp}(x, y)\ \texttt{< 0}}_{\text{build pattern}}$$

The implementation of this technique is beyond the scope of this article, and will not be covered here, however Section 6.2 gives some information on the subject of full concrete-syntax matching and rewriting.

3. IMPLEMENTATION OF TWEASTS

This part details prominent aspects of the design of the concrete-syntax rewrite engine. We present the main actors of the system, depicted on Figure 1: the `Tweast` class first; `ParserDriver`, which abstracts the task of parsing and scanning, the input being either text, or a composition of text and metavariables (`Tweast` objects); the parser and the scanner themselves; and cloner-based VISITORS, processing an abstract `Ast` recursively.

This method can be implemented in any general purpose programming language since it only requires a couple of extensions in the scanner and parser and a TWEAST container that is queried by the parser to fetch ASTs. The parser and scanner used must be re-entrant in order to use TWEASTs within the semantic rules of the parser to desugar constructs directly at the parsing stage.

3.1 TWEAST

In the example of translation of `&` into `if-then-else` of Section 2, it is not the pretty-printed sub-ASTs which are stored into the `Tweast`, rather the ASTs *themselves* are stored. When parsing TWEASTs, textual parts are analyzed and embedded ASTs are attached where appropriate. Figure 3 shows how data are stored in the TWEAST object.

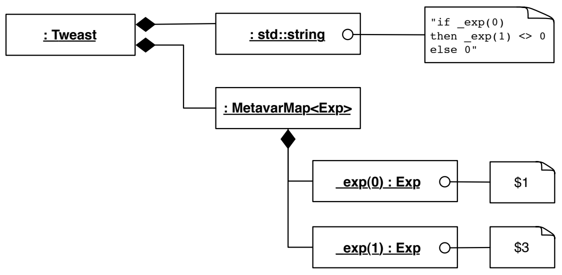


Figure 3: A TWEAST object holding the data of the desugared & expression from Section 2.

The class `Tweast` aggregates a growing string, and several typed arrays for sub-ASTs — expressions, l-values, declarations... We overload the `operator<<` to append standard material to the string, but to act differently for sub-ASTs:

```
Tweast& Tweast::operator<< (Exp* e)
{
  exp_[count_] = e;
  return *this << " _exp (" << count_++ << ") ";
}
```

In other words, the AST is kept in an array (say as '`exp_[42]`'), and `" _exp (42) "` is appended to the text. When parsing a `Tweast` the driver aggregates it, and both the scanner and parser are extended to recognize these syntactic constructs (see Section 3.3). The parser can then query the TWEAST to retrieve the given expression. Because in Tiger no user identifier may start by '_', new keywords such as '`_exp`' cannot introduce ambiguities in the language.

3.2 Parsing Driver

The whole parsing operation is complex and involves several small components: the scanner, the parser, incoming data (e.g., debugging flags, sub-ASTs...) and outgoing data (the resulting AST). We propose the PARSERDRIVER design pattern, which we can only sketch here. As a special case of the FACADE design pattern [7], it aggregates all the above components and coordinates them. It is the only public interface to the `Parse` module. It conducts the scanner and parser, and handles incoming (text, library path, options, etc.) and outgoing (ASTs) data. It implements recursive parsing invocations, keeping track of scanner states and opened files (to detect cycles). `ParserDriver` features a `parse(Tweast&)` entry point to build an AST from a `Tweast` object.

3.3 Parser and Scanner

TWEAST expects some support from the parser and the scanner:

- special codes like '`_exp`' must be recognized as valid tokens in the scanner, representing metavariables;
- metavariables must be accepted by the parser as valid right-hand sides of the corresponding non-terminal ('`_exp(42)`' must be a valid '`exp`').

Our implementation uses the Flex scanner generator [11] and the GNU Bison parser generator [4], but this is not a requirement to implement TWEASTs. The aforementioned changes are straightforward to implement with these tools. For each non-terminal (e.g., '`exp`') replaceable by a metavariable:

- a new token definition must be added to the parser definition file:

```
%token EXP "_exp";
```

- the scanner must be equipped to recognize this token:

```
"_exp"          return token::EXP;
```

- the parser must be augmented with a production accepting this token, and extracting the AST corresponding to the metavariable:

```
exp: "_exp" "(" INT ")" { $$ = driver.tweast->_exp[$3]; }
```

3.4 Visitors

As mentioned in Section 2, visitors are clients of `Tweasts` objects: as they traverse ASTs, they can be used to match one or several node(s), and trigger a rewriting process. Our transformations rely on the duplication of the AST. Hence, most transformation-related visitors derive from a common `Cloner` class to factor the code duplicating the AST (see Figure 1). Section 4 shows several visit methods of these visitors.

4. APPLICATIONS OF TWEASTS

TWEAST-based transformations allow many operations, providing a simpler, more concise and efficient rewriting framework than bare abstract-syntax manipulations. This section is a short catalog of some transformations using our concrete-syntax proposal.

4.1 Simplification by Syntactic Sugar Removal

TWEAST has been primarily designed with desugaring in mind. Reducing the size of a language to be further processed by removing syntactic sugar reduces the number of "active" AST nodes, as well as the number of addressed cases in traversals (visit methods of visitors).

Simple syntactic sugar can be removed at the parsing stage. In our Tiger compiler, this is the case for Boolean operators (`&` and `|`, see Section 1 and 2), and for the unary minus operator:

```
/* '-E' is translated as '0 - E'. */
exp: "-" exp
  {
    $$ = parse(Tweast() << "0 - " << $2);
  };
```

Even very simple TWEAST uses like these save the burden of implementing extra AST nodes (`UnaryMinus`, `And`, `Or`) and all the corresponding visit methods (`operator()`) in every visitor, which is a real benefit as far as maintenance is concerned.

More complex desugaring patterns can be expressed after the whole syntactic analysis has taken place, or even after the semantic analysis. This is the case of the string comparison de-overloading from Section 2. Another desugaring easy to implement is the translation of `for`-loops into `while`-loops, which subsequently allows the deletion of all the code processing `For` nodes after this desugaring pass.

```
void Desugarer::operator() (const For& e)
{
  // Relaunch the visitor on the children of 'e'.
  Exp* lo   = recurse(e.vardec_get().init_get());
  Exp* hi   = recurse(e.hi_get());
  Exp* body = recurse(e.body_get());
  // Symbols are fly-weight character strings [7].
  const Symbol& var = e.vardec_get().name_get();
  result_ =
    (parse(Tweast() <<
      " let"
      "   var _lo := " << lo <<
      "   var _hi := " << hi <<
      "   var " << var << " := _lo"
      " in"
      "   if _lo <= _hi then"
      "     while 1 do"
      "       ("
      "         " << body << ";"
      "         if " << var << " = _hi then"
      "           break;"
      "         " << var << " := " << var << " + 1"
      "       )"
      " end"));
}
```

The desugar approach can even be used to translate the input language into a simpler one. This is the strategy we choose to augment the Tiger language with object-oriented constructs (classes, objects, attributes, methods, inheritance). Instead of altering the whole compiler to support OOP, we only extended the front-end up to the semantic analysis, then added an object-desugar step translating the OOP dialect into the original object-less language. The following excerpt of our object-desugar visitor shows how method calls are turned into regular function calls[1].

```
/* Translate:
      o.m(a1, a2)     [with 'o' having static type 'c']
   as:
      _method_c_m(o, a1, a2)  */
Tweast
ObjectDesugarer::method_call(const Symbol& class_name,
                             const Symbol& method_name,
                             const std::string& target,
                             const Formals& formals)
{
  Tweast res;
  res << "_method_" << class_name << "_" << method_name
      << "(" << target;
  // Pass arguments.
  foreach (const ast::VarDec* arg, formals)
    res << ", " << arg->name_get();
  res << ")";
  return res;
}
```

4.2 Code Instrumentation

A task easily performed by program transformation is to add new statements to instrument the compiled code. Such additions can be used to

- add run-time checks of array accesses,
- trace the execution of the program by logging events like function entries and exits, memory allocation, etc.,
- record run-time information (time elapsed in functions, memory consumption) for profiling purpose.

4.3 Optimization

Term rewriting is also a good strategy to implement high-level language optimizations (i.e., performed in the front-end), while keeping the code very readable. Example of such optimizations include

- inlining of function bodies,
- loop unrolling (when the bounds are statically known),
- partial evaluation (when some or all of the terms of an expression are statically known).

[1]The `foreach` statement is an alias for the `BOOST_FOREACH` macro.

We implemented a simple inlining pass able to translate the following code

```
let function add(x : int, y : int) =
  x + y
in
  add(42, 51)
end
```

into

```
let var x := 42
    var y := 51
in
  x + y
end
```

Additional visitors can statically replace `x` and `y` by their value, then evaluate the sum and replace the binary `+` expression by its result.

```
void Inliner::operator() (const Call& e)
{
  // Get the AST node corresponding to this function call.
  const Function* fundec = e.definition_get();
  // A recursive function cannot be inlined.
  if (recursive_functions_set.has(fundec))
    // Don't inline, simply clone the function call.
    return clone(e);
  Tweast input;
  input << "let";
  // Introduce temporaries to evaluate formal arguments once.
  foreach (const Exp* exp, e.args_get())
    {
      Symbol v = symbol::fresh();
      Type* type = exp->type_get();
      Exp* init = clone(*exp);
      input << "var" << v << " : " << type << " := " << init;
    }
  // Inlined call.
  input << "in"
        << recurse(*fundec->body_get())
        << "end";
  result_ = parse(input);
}
```

5. DISCUSSION AND RELATED WORK

The TWEAST approach is correct since it does not pretty-print and re-parse the ASTs variables involved in concrete syntax constructs, which would lose the various existing annotations and would require special care with respect to operator precedences. However, this is inefficient compared to the other traditional approach that transforms concrete syntax in abstract syntax at compile-time of the compiler, since the parser is re-invoked multiple times at the execution of the compiler.

For instance, the Stratego/XT tool set [3] allows one to implement any kind of standalone program transformation system. It relies on a modular Syntax Definition Formalism (SDF) [12] and a Scanner-less Generalized LR (SGLR) parser [13] to analyze any context-free language. All sorts of analyses or transformations can be written in the Stratego programming language [15], which features concrete-syntax [14]. The advantage of our method compared to Stratego is that no special language is needed: it can be directly implemented in any compiler written in a general-purpose programming language.

Other alternatives to our proposal include

- functional languages such as Haskell or Caml, featuring matching capabilities that considerably simplify the implementation of rewrite rules (the VISITOR pattern is made worthless);
- language extensions such as Prop [9] or Tom [2], augmenting one or several general-purpose languages with pattern matching and abstract (not concrete) syntax rewrite rules;
- AST classes and traversal generation tools, like Treecc [16], which takes a similar approach to ours. (The project seems to be no longer maintained though.) Some parser generators automate the AST generation [10], and some even derive it from the SDF grammar [13]. JJForester [8] can derive a set of visitors from the same SDF grammar.

Table 1: Code reduction thanks to concrete syntax.

Measure	Abstract Syntax	Concrete Syntax	Gain
C++ `new` expressions	146	1	99%
Non-whitespace characters	995	671	32%
Words	886	340	61%

In practice the performance penalty introduced by TWEASTs is most likely unnoticeable. When concrete syntax is used in all the key places of a compiler where it greatly enhances the code and maintainability, the overall slowdown is negligible. In our Tiger compiler we observed that, for the changes in Table 1, TWEASTs account for 1.5% of the run-time of the front-end and less than 0.1% of the run-time of the entire compiler. Given the enormous code reduction entailed by concrete syntax, the trade-off is worth the small run-time penalty.

Errors in the concrete syntax are not reported until the compiler is run, thus losing most of the static safety guaranteed by languages such as C++. This, however, is the price to pay to have a system of concrete syntax without extending the host language (that in which the compiler is written) and without requiring an extra pre-processing step to expand concrete syntax in abstract syntax. In our experience, these limitations are outweighed by the advantages of the system.

6. EXTENDING THE TWEAST CONCEPT

This section presents current work on TWEAST to improve both their efficiency and usability.

6.1 Static TWEAST

A great part of the cost from using `Tweast`s comes from the parsing of the string they contain. However, the parsed string is always the same: the changing parts are only the value of the metavariable. As their contents does not influence the parsing itself, `Tweast` objects can be turned into `static` objects, and use a memoization strategy: parse the string they hold the first time they are used, keep the resulting AST, and assemble sub-ASTs at metavariable locations; then re-use the saved AST on subsequent `Tweast` uses.

Applying this ideas to the first desugaring example of Section 1 and 2 results in the following code.

```
exp: exp "&" exp
  {
    /* Build a static TWEAST from a build pattern (with
       typed metavariable) only. */
    static Tweast bool_and("if %exp:1 then %exp:2 <> 0 else 0");
    /* "Apply" it on sub-ASTs $1 and $3 to create the
       final AST, using 'operator%'. */
    $$ = exp(bool_and % $1 % $3);
  };
```

6.2 Full Concrete-Syntax Rewriting

One of the weakness of our approach is that the concrete syntax is only used to *produce* ASTs, not to *match* them. We still have to rely on abstract syntax to identify a match-pattern. With a concrete-syntax matcher, one could write transformations fully expressed in the compiled language:

```
// A rewrite rule translating '0 + e' as 'e'.
RewriteRule r("0 + %exp:1", "%exp:1");
Ast* ast = r("0 + 42");  // Rewritten as '42'.
```

Concrete-syntax matching requires several additions to the technique proposed here:

- a means to write match-patterns in concrete-syntax. This can be easily achieved by a small extension of `Tweast`;
- a MATCHER taking a pattern and an AST as input, recording every location where the pattern matches in the AST;
- a PRODUCER, using a set of locations produced by a MATCHER, a build-pattern and an AST, rewriting the latter using the formers;
- a REWRITER taking a match- and a build-pattern as input, as well as an AST, and coordinating a MATCHER and a PRODUCER objects to perform bottom-up or top-down rewriting on the whole tree.

We have added a simple implementation of this extension to our compiler to carry experiment on full concrete-syntax rewriting. First results are promising, and our compiler is able to apply rules like the one desugaring `for`-loops to `while`-loops (Section 4.1), fully in concrete syntax.

7. CONCLUSION

We described a set of techniques to implement concrete-syntax rewrite rules within stock C++. Concrete-syntax is featured by a new facility, Text With Embedded Abstract Syntax Trees, which reuses the parser at run-time and embeds existing stubs of AST. Experiments were conducted on our compiler for Tiger, a small and simple language.

The rewrite rules are used within a compiler or language-processing tool to easily desugar various constructs down to simpler ones, perform optimizations or instrument the code of the compiled program. This requires a minimum of equipment in the front-end (in particular in the parser and scanner) but offers a unique chance to the user to write his own rules.

The advantages of these techniques are manifold. They make the compiler much more maintainable by fostering stepwise desugaring with concrete syntax rewrite rules. They are also generic and easily portable to other general-purpose languages.

We also presented improvements of TWEAST, both from the performance and usability point of view: static TWEASTs use memoization to avoid repetitive run-time parsing costs, while concrete-syntax matching complete the system to provide a full concrete-syntax engine. The latter extension offers interesting evolutions: as our rewriting technique takes place during the execution of the compiler, concise concrete-syntax rewrite rules could be passed as command-line options or even better, embedded in the source code itself.

Acknowledgments

We thank Raphaël Poss for his proofreading of this paper.

8. REFERENCES

[1] A. W. Appel. *Modern Compiler Implementation in C, Java, ML*. Cambridge University Press, 1998.

[2] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16. Components for transformation systems. In *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, Charleston, South Carolina, January 2006. ACM SIGPLAN.

[4] R. Corbett, R. Stallman, and P. Hilfinger. Bison: GNU LALR(1) and GLR parser generator, 2003. http://www.gnu.org/software/bison/bison.html.

[5] A. Demaille. Making compiler construction projects relevant to core curriculums. In *Proceedings of the Tenth Annual Conference on Innovation and Technology in Computer Science Education (ITICSE'05)*, pages 266–270, Universidade Nova de Lisboa, Monte da Pacarita, Portugal, June 2005.

[6] A. Demaille and R. Levillain. *The Tiger Compiler Reference Manual*. EPITA Research and Development Laboratory (LRDE), 14-16 rue Voltaire, FR-94270 Le Kremlin-Bicêtre, France, 2007. http://www.lrde.epita.fr/~akim/ccmp/tiger.pdf.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[8] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Science of Computer Programming*, 47(1):59–87, 2003.

[9] A. Leung. Prop: A C++-based pattern matching language. Technical report, Courant Institute of Mathematical Sciences, 1996.

[10] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software, Practice and Experience*, 25(7):789–810, 1995.

[11] V. Paxson, W. Estes, and J. Millaway. *The Flex Manual*. The Flex Project, September 2007. http://flex.sourceforge.net/manual/index.html.

[12] E. Visser. A family of syntax definition formalisms. In M. G. J. van den Brand et al., editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.

[13] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[14] E. Visser. Meta-programming with concrete object syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[15] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.

[16] R. Weatherley. Treecc, the Tree Compiler-Compiler. http://www.southern-storm.com.au/treecc.html, 2002.