The Term Processor Kimwitu++

Martin v. LÖWIS

and

Michael PIEFEL
Institut für Informatik, Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany

ABSTRACT

Compiler builders have for a long time used tools that automate the frontend process of scanning and parsing the input. However, tools to handle the result of the parse are rarely found. Kimwitu++, the term processor, is an extension to the C++ languages which eases the production of programs that operate on trees or terms, and as such is a tool which is well suited for this purpose.

This paper presents an overview of Kimwitu++ with its different techniques such as rewriting and unparsing as well as some sample applications where we put it to use.

Keywords: Compiler, Abstract Syntax, Unparsing, Rewriting

1. INTRODUCTION

Kimwitu++ is a system that supports the construction of programs that use trees as their main data structure. It allows you to define, store and operate on trees with typed nodes. Each type of node has a specific number of children, and expects these children to have specific types. A very popular example of such trees are syntax trees as used in compiler generation.

The nodes are defined in a fashion quite similar to the definitions in Yacc. The constructed tree can be unparsed (i. e. treewalk) and rewritten (i. e. term substitution). Kimwitu++ gives you powerful yet simple to write pattern matching for specifying unparse and rewrite rules.

Kimwitu++ is an extension to C++. It introduces abovementioned Yacc-like node definitions, the unparse and rewrite rules, and extensions for pattern matching within C++-functions. It will translate its input files into pure C++. To build the tree you might use a parser generated with Bison [5], but you are free to use other tools.

We have used Kimwitu++ and its predecessor, Kimwitu, extensively in different compiler projects, and found that it simplifies the code generation process significantly. The Kimwitu++ compiler is written in the Kimwitu++ language itself.

2. HISTORY OF KIMWITU++

The Kimwitu system was originally developed by Peter van Eijk and Axel Belinfante, and described in [1]. It allows to use a high-level language to develop compilers, while still preserving the efficiency and portability of C when executing the compiler.

While working with Kimwitu for the SITE project [2], we found that the restriction to C as the back-end language is sometimes limiting. While Kimwitu allows for use C++ within the self-written code parts, and generated programs generally compile cleanly with C++ compilers, the following problems remain:

- Kimwitu allows to embed C code. This implies the usual problems of type safety. Specifically, it is common to reference all phyla through `void` pointers.
- While Kimwitu creates a new data type for trees, adding operations on these types is difficult. These types can be extended only with new data members.
- When creating and transforming large trees, memory management becomes an issue.

  Standard C has only restricted support for automatic memory management; Kimwitu adds support for a specific usage pattern by means of storageclasses. With C++, more automatic options become available, such as reference counting and smart pointers.

To remove these restrictions, we initiated the Kimwitu++ project, starting with Kimwitu 4.4 (see [1]). Later versions of Kimwitu, up to 4.6, have been incorporated into Kimwitu++, thus the functionality of Kimwitu++ is a superset of that of plain Kimwitu.

The migration from Kimwitu to Kimwitu++ is a relatively easy one, although it is slightly more complicated than going from C to simple C++. In the process of developing Kimwitu++ we simultaneously applied it in the project SDLC (see [3] and [4]), and ported Kimwitu itself to Kimwitu++. Today, Kimwitu++ is used in six different projects in our group.

3. KIMWITU++ SYNTAX AND SEMANTICS

A typical Kimwitu++ specification consists of the following parts:

- A definition of the abstract syntax
- A set of rewrite rules, to transform terms into a different term according to certain patterns
- A set of unparse rules, traversing a tree
- Supporting C++ functions (where the function body allows a few Kimwitu++-specific extensions to C++).

Since the output of Kimwitu++ is just C++, it is feasible to combine it with pure C or C++. For instance, often the component constructing the initial tree will be a parser, for instance one written in Bison, with another Flex frontend.

Abstract Syntax

The abstract syntax uses a Yacc-like notation which was originally inspired by the Synthesizer Specification Language SSL used by the Synthesizer Generator [6]. Here, a node type (or phylum) is defined by means of a set of alternatives (or operators). For example, to describe the abstract syntax of expressions, the following fragment could be used:

```
expr:
    Number( integer )
    | Variable ( casestring )
    | Plus ( expr expr )
    | Mul( expr expr )
    | Minus( expr expr )
    | Div ( expr expr )
    ;
```

In this example, expr is a node type (phylum) with six different alternative forms (operators). Each of the operators has a sequence of typed parameters, for example, the Mul operator requires two parameters that are of the type expr.

Some phylum types are predefined, such as integer and casestring, representing integer numbers and case-sensitive character strings, respectively.

In the next fragment we want to represent a number of assignments to variables (e. g. to take them into consideration before evaluating the expr). We define a phylum to represent the assignment itself and another one to hold a list of those:

```
assignment:
    Assignment(expr casestring);

assignments:
    list  assignment;
```

Here, assignments is a list of assignment phyla. The special construct list implicitly introduces two operators for the end of the list and another for a member of the list linking to the next node. Also list phyla have a number of additional methods to operate on them, such as reverse, map, filter etc. The definition for assignments is roughly equivalent to:

```
assignments:
     Consassignments(assignment assignments)
   | Nilassignments();
```

The above is sufficient for specifying arbitrary abstract grammars. For further processing, for instance attribute grammars, the phyla can be extended with additional attributes. To add a variable to hold the synthesized value of an expression, we use:

```
expr:
    {
        int val;
    };
```

In Kimwitu++, each phylum and each operator translates to a C++ class. Kimwitu++ allows the specification of additional members and methods for each generated class. By using C++ classes, automatic construction and destruction of members is guaranteed by the C++ runtime semantics.

An equivalent, but more flexible, way to add the val attribute, is to use the %member construct; this also allows to add to phylum types and to write an initializer:

```
%member int expr::val = 0;
```

Adding methods is even simpler

```
int expr::base64(ostream &o) { /* some processing */ };
```

Here, the phylum type expr (internally represented by a C++ class) is extended with a method called base64; the required declarations will automatically be inserted into the class definition.

Rewrite

Rewrite rules allow the transformation of one tree into another. For example, given the syntax above, the distributive law could be expressed as

```
Mul(a, Plus(b, c))        -> <expand:
    Plus(Mul(a, b), Mul(a, c)) >;
```

Each rewrite rule is restricted by a pattern and a view. The pattern is given before the -> sign. It may be arbitrarily complex, and multiple patterns can be given separated by commas. The most specific pattern that matches will then cause its rule to execute.

A Kimwitu++ program can contain many different rewriting strategies, giving a view name to each. In the example, the view is named expand, indicating that the expression will expand under this specific application of the law of distribution. It is the application's choice to apply a certain rewrite view to a term, yielding different rewrite results for different purposes.

Here are two other rules which will eliminate some superfluous subterms:

```
Mul(Number(0), *),
Mul(*, Number(0))          -> <simplify:
    Number(0) >;

Mul(Number(1), t),
Mul(t, Number(1))          -> <simplify:
    t >;
```

Rewrite rules are applied repeatedly until none of the patterns in the current view matches anymore. In the above example, once the rewriting finishes in the view simplify, there will be no terms left where 0 or 1 are part of a multiplication.

Note that, since rewriting will only stop when no rules match, all rules must produce a new term which is closer to some kind of fixed point. You should avoid cycles, for instance stating commutative relations is dangerous. This is not a limitation of Kimwitu++ in particular, but rather that of any rewriting system.

Unparsing

While rewriting traverses the tree with the specific goal of constructing a new one (and giving a convenient notation to do so), unparsing allows more general traversal of the tree. Similar to rewriting, named views determine the control flow of the the unparsing process.

Although any treewalk can be accomplished easily with unparse rules, their original purpose (and origin of their name) is to reverse the process of parsing terms into syntax trees by generating a text representation. For example, to produce a prefix notation of an expression on standard output, the following unparse rules could be part of the unparsing specification:

```
Plus(a, b)                 -> [prefix:
    "+␣" a "␣" b ];

Mul(a, b)                  -> [prefix:
    "*␣" a "␣" b ];

Minus(a, b)                -> [prefix:
    "-␣" a "␣" b ];

Div(a, b)                  -> [prefix:
    "/␣" a "␣" b ];
```

In this fragment, again each unparsing rule starts with a pattern. If the pattern and the view name matches, the unparsing body is executed and each unparse item is processed in succession. Unparse item can be:

literal A plain string is printed.

C++ code A block of C++ code is denoted by surrounding curly braces. It is copied verbatim into the generated code and will be executed.

variable Everything else is recursively unparsed.

Kimwitu++ provides default unparsing rules, which traverse the parameters of each operator, from left to right. The predefined phyla (i. e. integer or casestring) are printed to standard output in their text representation.

Unparse rules are made more flexible by another two features: Printers and view change. A printer is simply a functor taking the string that is to be printed—it does not necessarily have to print it, though. It can apply further modifications (like indentation) or simply do nothing if all necessary actions have been taken in the C++ blocks within the unparse rules.

Every unparse item can have a view attached. Normally, any item is unparsed with the current view, i. e. the one that was active when the current rules was entered. Attaching a view name will unparse the item with the named view instead. This allows to reuse views in different contexts and offers modularization.

C++ code, and integration into the main program

Kimwitu++ recognizes blocks of C++ code, either incorporating them into its output files, or processing them to provide syntax extensions. In particular, Kimwitu++ provides the with keyword, which allows to apply pattern matching inside of a C++ function.

Extended C++: To work efficiently with complex terms or trees it is necessary to do pattern matching. The same powerful pattern matching which is available in the head of rewrite and unparse rules is also usable in C++ code through the use of additional keywords. There is a with keyword for pattern matching, and foreach keyword to facilitate matching over all elements of a list.

A (contrived) fragment to find the sign of an expression e could be written as follows:

```
with (e) {
    Minus(a, b):      {
        if (a->val < b->val)
            return NEG;
        else if (a->val == b->val)
            return ZERO;
        else
            return POS;
    }
}
```

An example for the special list processing that foreach allows, let us determine the number of (expensive) division operations in a list expressions containing expr phyla:

```
foreach (Div (*, *);  exprlist  expressions) {
    count++;
}
```

Alternatively, to count both divisions and multiplications on the one hand and additions and subtractions on the other hand, foreach can also be used like this:

```
foreach ($e; exprlist expressions) {
    Mul (*, *),  Div  (*, *) : {   mul_count++; }
    Plus (*, *),  Minus (*, *) : {  add_count++; }
}
```

Interfacing non-Kimwitu++ code: To write a complete compiler, Kimwitu++ needs to interact with the parser. The most common technique is to combine Kimwitu++ with Bison [5], putting actions to create nodes of the abstract syntax into the reduce actions of the parser. For example, a parser for a reverse-polish notation may have the production

```
term:
            NUMBER
            { $$ = Number(mkinteger(atoi($1))); }
          | IDENT
            { $$ = Variable(mkcasestring($1)); }
          | term term '+'
            { $$ = Plus($1, $2); }
          | term term '*'
            { $$ = Mul($1, $2); }
          | term term '−'
            { $$ = Minus($1, $2); }
          | term term '/'
            { $$ = Div($1, $2); }
          ;
```

Furthermore, a typical compiler application is a stand-alone program whose actions are controlled by the main() function. This function needs to interact both with the parser and the rewrite and unparse actions. For this example, the main function may read

```
int main()
{
    yyparse();
    TheInput = TheInput−>rewrite(kc::expand);
    TheInput−>unparse(printer, kc::prefix);
}
```

In this application, it is assumed that the Bison start production assigns the top-level term to the global variable TheInput. This term is, in turn, rewritten using the expand view, and unparsed using the prefix view.

4. APPLICATIONS

Kimwitu++ is used in a number of projects at Humboldt-Universität. We use it to build different compilers for languages such as SDL, ASN.1 and CIDL. We will give a short overview of two of them here.

SDLC

One particularly interesting project—which is the result of [3, 4]— is SDLC. In this project we build tools that automatically generate a compiler from SDL's specification [7].

The generated compiler transforms SDL to ASM [8] (which in turn can be executed). This compiler is meant to be a *reference compiler*, meaning it does not necessarily produce efficient programs, but instead always programs which are correct and adhere to SDL's very complex semantics.

The SDL Standard: The standard document for SDL is a text file. Using the macro functions of the word processor plain text versions of the crucial parts were extracted. These contain two parts: the static semantics with the different grammars of SDL (a concrete one and two increasingly abstract ones), and consistency rules; and the dynamic semantics which define SDL in terms of ASM.

The compiler needs to transform a concrete program firstly from its concrete syntax representation into the abstract one, then in turn expressing this with ASM.

Also the abstract syntax was not fixed at the start of the project, and tools were built to transform the concrete grammar to a draft of the abstract grammar.

The Tool Chain: The static semantics of SDL consist of the consistency rules expressed in predicate calculus and rewrite rules to transform a program from its concrete form into the abstract forms.

In SDLC, the consistency rules are expressed as boolean functions which operate over complex terms. These are written in Kimwitu++ using the abovementioned extensions to C++ such as the with statement.

The rewrite rules of the SDL standard are a natural match for the rewrite rules of Kimwitu++. This makes for a simple one-to-one relationship.

The dynamic semantics are defined over the terms of the abstract syntax. For such terms, ASM program fragments are given. In SDLC, unparse rules are employed to yield a complete ASM program from the abstract syntax tree of the compiled SDL program.

The tools that generate the SDL compiler from the plain text extracts of the SDL standard are also written in Kimwitu++.

CIDL

The CORBA Component Model [9] employs a new form of extended IDL for its specifications. However, in order to stay compatible with established and well-tested tools, the language bindings were not altered. This makes it necessary to transform IDL 3 to IDL 2. This mapping is defined in the standard.

The IDL 3 to IDL 2 compiler is a very typical Kimwitu++ application. The processing is done in the following steps.

The compiler starts by scanning and parsing the original IDL 3 specification, it thereby builds a tree representation of the input. This is done using standard techniques using Flex and Bison as described above.

This tree containing new IDL 3 features is then rewritten into a (larger) tree which consists only of the basic IDL 2 features.

Finally, the result tree is unparsed. The name is suitable here, as this is the exact opposite of parsing. Since the original white-spacing is lost, new style rules are applied. This way the compiler can also act as a pretty-printer on basic IDL 2.

5. SUMMARY AND CONCLUSIONS

Using higher-level languages to construct compilers has proven to give a significant increase in the productivity of developers. Having such tools generate object-oriented code, e. g. by using C++, provides even more convenience by giving access to more powerful data structures and feature-rich libraries.

Kimwitu++ is such a high-level language. It is well-suited for all kinds of processing typed trees. Though it has been developed in a compiler building environment, it is not restricted to that domain.

Kimwitu++ has been successfully employed in a number of projects. We are now in possession of meta-tools for all stages of the compiler building process.

6. ACKNOWLEDGEMENTS

We would like to thank Axel Belinfante (the author of Kimwitu) for the fruitful conversations and prompt reactions to our needs.

Ralf Schröder and Andreas Prinz provided valuable feedback. Harald Böhme and Toby Neumann greatly improved Kimwitu++ by actually using all of the new features that we put into the program.

7. REFERENCES

[1] Peter van Eijk and Axel Belinfante. *The Term Processor Kimwitu: Manual and Cookbook* University of Twente, Enschede, Netherlands, 1992

[2] Ralf Schröder. *SDL Integrated Tool Environment* Humboldt-Universität zu Berlin, Germany, 2002 < http://www.informatik.hu-berlin.de/SITE >

[3] Michael Piefel. *Ein automatisch generierter SDL-Compiler* Humboldt-Universität zu Berlin, Germany, 2000

[4] Andreas Prinz. *Formal Semantics for SDL – Definition and Implementation* Humboldt-Universität zu Berlin, Germany, 2001

[5] Charles Donelly and Richard Stallman. *GNU Project parser generator (yacc replacement)* Info documentation for version 1.34, Free Software Foundation, 2001

[6] Grammatech Inc. *The Synthesizer Generator* Grammatech Inc., 2001 < http://www.grammatech.com/products/sg/ >

[7] ITU-T Recommendation Z.100. *Specification and Description Language (SDL)* International Telecommunication Union (ITU), Geneva, Switzerland, 2000

[8] Jim Huggins *Abstract State Machines* University of Michigan, Ann Arbor, USA, 2002 < http://www.eecs.umich.edu/gasm >

[9] Dr. Phillipe Merle (editor) *CORBA 3.0 New Components Chapter* Object Management Group TC Document ptc/2001-11-03, Boston, USA, 2001