The Term Processor *Kimwitu*: A system for generating language-based software

Peter van Eijk   Axel Belinfante   Henk Eertink
pve@cs.utwente.nl
University of Twente, Fac. Informatics
7500 AE Enschede, NL

Abstract

We present a new system to support the construction of language-based software. Its major innovation is open multi-paradigm programming. This allows such software to be described in an arbitrary variety of styles, and also facilitates integration of language based tools. A large application and the impact on productivity of our system is discussed.

1 Introduction

The construction of language-based software, such as compilers and tools for the analysis of programs or specifications, is a rich semantic domain. There are a number of formalisms in use to describe the functionality of such software. One sensible approach to implementing such software is to use a very high level programming language, of which the semantic model is closely related to the formalism in which the functionality is described. Examples are attribute grammars and functional languages. This works fine for prototypes of the software, but does not always result in satisfactory space and time performance. Another approach is to use a programming language that allows total control over the space and time consumption of the software, such as C. Its disadvantage is that the gap between the description of the functionality and the implementation of the functionality is large, implying a long, tedious, and error-prone implementation effort.

Our system attempts to blend the advantages of both approaches, and bridge this gap. In essence, we allow open multi-paradigm programming. Multi-paradigm programming allows to express each part of the implementation in the most appropriate language. It is a 'best of both worlds' approach, where one uses a high-level language where possible, and a low-level language where necessary. 'Open' in this context means that escape hatches are provided to other implementation techniques. A basic common concept in the programming languages for language-based software is trees, or terms. This forms the basis of our system, so all formalisms operate on the same kind of structures. It turns out that such an approach also facilitates tool *integration*.

The contents of the rest of this paper is as follows. In Section 2 we analyse some existing formalisms for the construction of language based software and present the concept of a *term processor*. In Section 3 a more technical discussion is given of our particular system, which is followed in Section 4 by a presentation of some existing applications. Section 5 discusses the strengths and weaknesses of our approach, and in Section 6 we draw some conclusions.

2 Constructing Language-Based Software

In this section we discuss a variety of systems that are used for the construction of language-based software. Basic tools of course are Yacc and Lex, which are based on context-free grammars and regular expressions. They do not support any data type construction and allow functions to operate in a one-pass bottom-up way only. NewYacc[PC89] is a derivation of `Yacc` that retains the parse tree, and allows treewalks and unparsing definitions to be described. It does not allow attribute evaluation specifications or direct access to the parse tree.

On the other side of the spectrum there are fully syntax-driven systems like the SG[TR89] and Linguist[Far84], which are based on attribute grammars, and CENTAUR[B^+88], based on natural semantics. These systems are suitable for a wide range of realistic applications, e.g. [vE89], but one cannot easily circumvent or escape their formalisms when that is necessary, e.g. for performance reasons.

IDL (Interface Definition Language)[Sno89] was initially developed for the description of interfaces between ADA tools. It provides a large number of constructs to describe annotated trees, but little mechanism for the definition of functions over these trees.

High-level programming languages such as PROLOG and ML have also been demonstrated to be suitable for some realistic language-based tools. Rewrite formalisms have been used for code generation [FW88]. Again, in each of these formalisms there are things that cannot be easily or efficiently expressed.

There is some commonality in these formalisms, though. All of them, in one way or another, allow trees (or terms) as a fundamental data structure. To some extent, all these systems can be called *term processors*. Our system distinguishes itself by allowing a variety of formalisms.

3 The *Kimwitu* System

The *Kimwitu*[1] system is a *term processor*. The basis of our language is a notation to describe a term algebra, which defines a set of terms and operations to construct terms. Computations over these terms can then be described through a variety of mechanisms, as explained below. Terms can be manipulated in a tool as well as exchanged between tools. In this way tool integration is facilitated because the same term algebra describes both the internal as well as the external representation of values.

The input of the *Kimwitu* compiler is an abstract description of terms, annotated with implementation directives, plus a description of functions on these terms. Examples of the latter are simple function definitions and rewrite and unparse rules. The output consists of a number of C files, containing data structure definitions for the terms, a number of standard functions on the terms, and a translation (in C) of the function definitions and rewrite and unparse rules in the input. The standard functions can be used to create terms, compare them for equality, read and write them to files and do manipulations like list

[1] *Kimwitu* (pronounced 'kee-mweetu') is pidgin-Swahili for 'language of trees'.

concatenation. All generated functions contain assert statements that check dynamically for NULL pointers and type mismatches. Type mismatches, by the way, can also be detected by Lint. Additionally, analysis and statistics gathering functions are generated.

The functions that read and write terms from and to a file use the 'structure' file format that is used by editors generated by the SG. These functions can be used to pass terms between *Kimwitu* generated tools and SG-generated editors if the phyla of the exchanged term are defined in the involved tools.

3.1 Terms

The input structure of terms is borrowed from the SG Specification Language SSL. Among the predefined phyla are int and string. An example declaration is the following.

```
expr: Plus( expr expr )
    | Const( int ) ;
exprlist: list expr;
```

This declares two *phyla*, or types, or nonterminals, depending on your viewpoint. Each of these denote a set of terms. As shown, there are two ways of constructing a phylum. One is by enumerating its variants, each of which is an *operator* applied to a list of phyla. The other is by declaring it as a list phylum, which has, apart from its brevity, the advantage that it instructs the system to generate additional, list-specific functions. The example (`exprlist`) is effectively equivalent to the following right-recursive phylum.

```
exprlist: Nilexprlist()
    | Consexprlist( expr exprlist ) ;
```

Attributes can be added to the phyla. In our system, attributes are only a way to decorate trees with values. The actual decoration is up to the user, who can use any tree walking scheme for decoration. In this example an integer attribute `value` is added to the phylum `expr`.

```
expr: { int value = 0; };
```

The type of the attribute can be any C type, including one that results from a phylum declaration. Attributes can be initialised by giving the initial value in the attribute declaration. Attributes do not appear in structure files.

For each phylum, *Kimwitu* generates a C data type with the same name and C functions to construct and otherwise manipulate objects of such a type. Technically, the data structure is a pointer to a structure containing the attributes, a variant selector (cf. the operator), and a union of structures. Note that this scheme allows type checking by Lint over C programs to check if a term is constructed from the correct phyla.

An additional generated data type is YYSTYPE, technically a union of all phyla, which can be used in Yacc-generated parsers to construct terms. This makes it relatively easy to combine the use of Yacc, Lex and *Kimwitu*.

By default, each operator 'application' just yields a new 'memory cell' containing pointers to the arguments of the operator, with initialised attributes. However, if the storage option 'uniq' is specified for a phylum, the system guarantees that if an operator is called once with a certain set of arguments, each additional call with the *same* arguments will yield a pointer to the cell that was created by the first call. As a result, common subterms are automatically shared, resulting in a directed acyclic graph. Hashing is used to implement this uniqueness of representation property. A property of uniquely represented terms is that checking for structural equivalence of two terms, is equivalent to pointer comparison. The unique storage property, in combination with attributes, can be used to implement a symbol table, or more generally function caching[Pug88], without much programming effort.

3.2 Pattern Matching

Although the generated data types have a fairly regular structure and can be accessed by arbitrary C code, it is more convenient to make explicit use of structure induction in the definition of functions. *Kimwitu* gives the user flexibility in the manipulation of terms through the use of patterns over terms in several ways: in function definitions, rewrite rules and unparse rules.

3.3 Function Definitions

The syntax of the body of a function definition is C, extended with with-statements in which pattern matching can be expressed. As an example, the following function returns the value of its argument expression.

```
int get_value(e) expr e;
{   with( e ) {
        Const( i ): { return i; }
        Plus( e, e ): { return (2*get_value(e)); }
        Plus( e1, e2 ): { return (get_value(e1) + get_value(e2)); }
}   }
```

The function does a case analysis on the argument of the with-statement. For each pattern, a piece of C code is given between curly brackets, in which the variables bound in the pattern can be used.

An alternative notation for pattern matching is useful and more concise for certain special cases. If a parameter (e.g. of a function) is prefixed with a dollar symbol, an implicit 'with' can be used, as in the following example. The components of a production are then referenced as `$i`. The previous example then translates into the following.

```
int get_value($e) expr e;
{       Const: { return $1; }
        Plus: { return (get_value($1) + get_value($2)); }
}
```

In addition to the with-statement, there is also a *foreach*-statement through which one can enumerate over lists. To illustrate this, suppose we add the following production to `expr` to denote a sum of expressions.

```
| Sigma( exprlist )
```

The additional case in body of get_value can then be written as follows.

```
Sigma(e2): { int sum=0;
             foreach(e3; exprlist e2) {sum +=get_value(e3);}
             return sum;}
```

The parameters of the foreach clause include, in that order, the for variable, and the type and name of the list value that the body cycles through.

A variant of the foreach clause has a pattern in the place of the for variable, which acts as a filter on the list elements. The body will only be applied on the list elements that match the pattern.

3.4 Rewrite rules

Rewrite rules can also be used to specify computations over terms. An example rewrite rule is:

```
Plus(Const(i), Const(j)) -> Const(cplus(i,j));
```

The left-hand side of a rewrite rule is a pattern, which binds variables that can be used in the right-hand side. The right-hand side is a term, consisting of operators (e.g. `Const`), and C function calls (e.g. `cplus`). In a pure rewrite rule, the right-hand side consists of only operators.

From the collection of rewrite rules, for each phylum a function `rewrite_phylum` that returns the normal form of the argument, is generated. The currently implemented rewrite strategy is left-most inner-most.

3.5 Unparse rules

Unparse rules describe treewalks that can be used to derive textual representations of terms. Each unparse rule consists of a pattern, a list of views and a list of unparse items. The patterns are the same as those in function definitions and rewrite rules. Views can be used to specify several textual representations for the same term. An unparse item can be any of the following: a string denotation, a piece of arbitrary C code in which pattern variables can be used, a pattern variable or an attribute of a pattern variable. From the collection of unparse rules, for each phylum a function `unparse_phylum` is generated. These functions take three arguments: the phylum that will be unparsed, a (void) printer function (to be supplied by the user) that will be applied to each string denotation, and the view to be used. Each unparse item defines a part of an `unparse_phylum` function. The relation between unparse items and generated code is as follows: a string denotation is mapped to an invocation of the printer function, the C code is copied, and a pattern variable, or attribute, maps to a call of the corresponding `unparse_phylum` function.

The following example demonstrates how pattern matching can be used to handle list separators. The number of separators is one less than the number of list elements.

```
Nilexprlist() -> [:];
Consexprlist( ex, Nilexprlist() ) -> [: ex ];
Consexprlist( ex, rest ) -> [: ex ", " rest ];
```

4 Experiences

Our main experience with *Kimwitu* is from the work on LOTOS [ISO88] tools. This work was either carried out at the University of Twente or elsewhere in Lotosphere, an Esprit II project involving 16 partners in 7 countries.

In the LotosPhere project an integrated toolset is being built for LOTOS. Every tool in this toolset works on a central object, which is a representation of a LOTOS specification. This object is formally described in 525 lines of *Kimwitu* input. *Kimwitu* generates data structures and I/O routines from this description. This makes changes to the structure of the interface object rather easy — in most cases programs only have to be recompiled. The fact that the specification of the central object is used for both the external and the internal representation simplifies the production of tools that work on the central object. In one case, a person with no experience in C or *Kimwitu* produced a conversion tool in one week.

A compiler for equational systems into code for specialised abstract term rewriting and narrowing machines [Wol90] has been produced using *Kimwitu* in a three man-months, which was about half of the planned development time. This system is described in 2900 lines of *Kimwitu* input. In particular the automatic management of symbol tables proved very helpful. The speed of the resulting program is comparable with earlier versions, which were written in C. The interpreters for the abstract machines were written in 2600 lines of C-code.

A full LOTOS simulator (called Smile) has been built in 6 man-months. This simulator does extensive manipulation of complex data structures. The size of *Kimwitu* code is about 4000 lines (112 Kb) with an additional 1200 lines of C-code for the X-Window based user-interface. These numbers do not include the abstract data type part mentioned previously.

A previous system, Hippo [vEVD89], built by different persons, was implemented in 20,000 lines of Yacc, Lex and C, of which 5000 lines are devoted to the abstract data type part. Its development took 18 man-months. It is now hard to extend and maintenance on it has been stopped.

Generally speaking, Smile has more functionality than Hippo: it is fully symbolic and its abstract data type part is much more advanced. The memory consumption of Smile is less and the execution speed is better (both by a factor of at least 2 to 3), on a comparable run.

The following gives an indication of the performance of the generated code. A 3195 line LOTOS specification results in a structure file of 780 Kb, containing approximately 200,000 operator applications. Reading this object, initialising the simulator, and compiling the abstract data types takes 18 seconds of cpu-time on a SparcStation.

5 Discussion

In this section we discuss why we are pleased with the system and what its weaknesses are. The system improves productivity, is relatively easy to learn, and produces efficient code.

Software productivity appears to be related to the number of lines written per month, independent of the notation. This implies that a more compact notation, in principle, allows a certain amount of functionality to be produced in less time. Our notation is compact for two reasons. Some code does not have to be written at all, e.g. C structures and generated functions, and there is flexibility in choosing the most appropriate (=more compact) paradigm. Furthermore, productivity is improved if less time is spent on debugging. Lines that do not have to be written cannot contain errors. The generated code also contains the assert statements which programmers are usually too lazy to write. Since *Kimwitu* automates the process of interfacing between tools, hardly any integration problems occur.

Our experiences confirm this expectation. It is always hard to compare two competing approaches to software construction. A controlled experiment is methodologically hard to do and very expensive. There is some experimental result, though, if the remake of a tool has more functionality and is developed in less time, by different persons. For example, in comparison with Hippo, Smile has more functionality, better performance, and was produced in half the time, using half the number of lines of code.

Can one afford to use *Kimwitu*? One aspect of this is the learning effort required. The entire manual [vEB90], including cookbook style examples, contains only 32 pages. This is partly due to the care we took in designing a small language. Another aspect of affordability is the performance of the generated programs. As we have shown earlier, this is quite acceptable.

What are the weaknesses? We have not been able to find, despite repeated attempts, a way to describe polymorphic functions that can be mapped easily to C. This is partly due to the C language. Furthermore, the system generates a number of functions for each phylum. Not all these functions are used in a typical application, which results in executables that contain superfluous code. A global analysis at link-time would be able to remove this.

6 Conclusion

The novelty of our approach is to allow a variety of formalisms to be used in the construction of language-based software. We do not claim novelty in the formalisms used, but rather in their combination. We believe that our system is a significant tool in the implementation of programming and specification languages.

We would like to thank the users that gave us feedback on the system, in particular Robert Elbrink of PTT Research Netherlands, Dietmar Wolz of the Technical University of Berlin, and Eric Madelaine of INRIA Sophia-Antipolis. Albert Nijmeijer made helpful comments on this paper.

References

[B^{+}88] P. Borras et al. CENTAUR: the system. *SIGPLAN*, 24(2):14–24, Nov 1988.

[Far84] Rodney Farrow. Generating a production compiler from an attribute grammar. *IEEE Software*, 1(4):77–93, Oct 1984.

[FW88] C. W. Fraser and A. L. Wendt. Automatic generation of fast optimizing code generators. *SIGPLAN*, 23(7):79–84, July 1988.

[ISO88] ISO. IS 8807 information processing systems - Open Systems Interconnection - the definition of the specification language LOTOS, 1988.

[PC89] James J. Purtilo and John R. Callahan. Parse-tree annotations. *CACM*, 32(12):1467–1477, Dec 1989.

[Pug88] William Pugh. *Incremental Computation and the Incremental Evaluation of Function Programs.* PhD thesis, Cornell University, Ithaca, N.Y., 1988.

[Sno89] Richard Snodgrass. *The Interface Description Language: Definition and Use.* Computer Science Press, Rockville, MD, 1989.

[TR89] T. Teitelbaum and T. W. Reps. *The Synthesizer Generator - A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1989.

[vE89] Peter van Eijk. LOTOS tools based on the cornell synthesizer generator. In H. Brinksma, G. Scollo, and C. A. Vissers, editors, *Proceedings of the ninth international symposium on protocol specification, testing and verification*, Amsterdam, 1989. North-Holland.

[vEB90] Peter van Eijk and Axel Belinfante. The termprocessor *kimwitu*, manual and cookbook. Technical Report INF-90-45, University of Twente, Enschede Netherlands, 1990.

[vEVD89] P.H.J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS - results of the ESPRIT/SEDOS project.* North-Holland, Amsterdam, 1989.

[Wol90] D. Wolz. Design of a compiler for lazy pattern driven narrowing. In Unknown, editor, *Proc. of the 7th international workshop on specifications of abstract data types, Wusterhausen-Dosse*, pages ?–?, Berlin, 1990. Springer-Verlag. Lecture Notes in Computer Science ???