Industrial Strength Compiler Construction with Equations

Lutz H. Hamel

BKS Software GmbH
Guerickestr. 27
1000 Berlin 10
Germany

usenet: lhh@bksbln

Abstract

Industrial strength compilers need not only be robust and efficient but also understandable, maintainable, and extensible. To make compilers more manageable there has been a strong trend towards the high-level specification of the different phases of the compilation process. However, the high-level specification of one phase - the semantic analysis - has not been very successful in terms of production quality standards. In this paper we propose the use of a language based on equations for the specification of the semantic analysis phase of compilers.

1. Introduction

Industrial strength compilers need not only be robust and efficient but also understandable, maintainable, and extensible. In the past compilers in industrial production have been notorious in that respect. Implementations which were difficult to read have hidden the actual functionality of the compilers. This was mainly due to the fact that the traditional implementation languages lacked the features needed to elegantly express transformations and algorithms routinely needed in compilers. Thus, important aspects of the compilation were buried in unimportant implementation details.

There has been a strong trend towards the high-level specification of compilers to make them more manageable. Over the recent years tools such as *lex* (Lesk, 1975) and *yacc* (Johnson, 1975) have been used to specify the lexical and syntactical analysis phases of compilers, respectively. Code generation has been generally recognized as a bottom-up tree pattern matching problem (Aho, 1985, Glanville, 1977, Hatcher, 1986) and various tools have been proposed for the construction of efficient code generators from high-level specifications (Fraser, 1988, Hatcher, 1988) . Some of these tools have successfully been used to construct commercial compilers. However, the high-level specification of one phase - the semantic analysis - has not been very successful in terms of production quality standards. Most of the attempts to propose such a high-level specification (Gordon, 1979, Knuth, 1968, Lee, 1989, Paakki, 1991) have failed to yield tools that generate compilers efficient enough for industrial applications or have simply proved too unwieldy to use.

In this paper we address this issue by proposing the use of a high-level specification language based on equations. Due to their declarative nature and their clean semantics, equational specifications produce compilers which are easy to understand and maintain, hence easy to extend. We also observe that a compiler implemented with our equational notation is still comparable in efficiency with hand coded compilers. We have used this notation to implement a commercial compiler which translates a superset of C++ into ANSI 2.0 C++.

The rest of the paper is organized in the following way: in section 2 we try to informally motivate the use of equations. Section 3 takes a brief look at UCG-E, an equational specification language. The implementation of our compiler using UCG-E is discussed in section 4. We close with some general remarks and observations in section 5.

2. Why Equations?

We were attracted to equations as a specification language due to their declarative nature and their extremely simple semantics when viewed as directed rewrite rules (Herman, 1991, Huet, 1980) : a term which matches the left hand side of an equation is simply replaced by the term on the right hand side of that equation. The rewrite process continues until no further matches can be identified.

Formalizing this a little bit; an *equational logic program* (Hatcher, 1991, O'Donnell, 1985) consists of a set of equations of the form

```
X = Y
```

where X and Y are well-formed terms over a ranked alphabet and a set of variable symbols. Input to an equational logic program is a well-formed term of the ranked alphabet only. Execution of an equational program consists of exhaustively applying the equations as rewrite rules to the input term. Any subterms (including previously rewritten terms) which match the term X are replaced by term Y. Rewriting halts when no further subterms of the form X can be identified.

The following is an equational logic program which appends an item to a list of items. We show the program with a sample input, the corresponding trace, and the resulting output term.

Program:

```
append(item, nil) = cons(item, nil)
append(item, cons(head, tail)) = cons(head, append(item, tail))
```

Input Term:

```
append(3, cons(1, cons(2, nil)))
```

Trace:

```
append(3, cons(1, cons(2, nil)))
cons(1, append(3, cons(2, nil)))
cons(1, cons(2, append(3, nil)))
cons(1, cons(2, cons(3, nil)))
```

Output Term:

```
cons(1, cons(2, cons(3, nil)))
```

We also believe that equations and equational rewriting match very well the intuitive view a programmer has of the semantic analysis phase: a successive rewriting of the abstract syntax tree (as generated by the parser) into an intermediate form - the "semantic" tree, which we view as the final result of the semantic

analysis. The recursive nature of the parse tree is handled implicitly by the equations, since they apply to any term appearing anywhere in the tree. Consider the following examples.

```
plus(constant(val1), constant(val2)) = constant(val1 + val2)
```

The three node pattern in the abstract syntax tree is replaced by a single constant node of the right hand side of the equation. If the rewritten `plus` term happens to be a subtree of another `plus` term, the equation would simply be applied again given that the second `plus` node has now two constant children. The following rewrite sequence illustrates this: `3 + 2 + 5 => 3 + 7 => 10`. Transformations of this kind are usually part of the constant folding mechanism in compilers.

Typically compilers have to do some form of type checking as part of the semantic analysis. Type information about subtrees is propagated throughout the semantic tree. This can conveniently be expressed with equations:

```
(1) plus(int(left), int(right)) = int(int_plus(left, right))

(2) plus(float(left), int(right)) =
            float(float_plus(left, int_to_float(right)))

(3) plus(int(left), float(right)) =
            float(float_plus(int_to_float(left), right))

(4) plus(float(left), float(right)) =
            float(float_plus(left, right))
```

Equation `(1)` states that a `plus` node with two integer subtrees itself becomes an integer subtree with the addition being an integer addition. Analogously, equation `(4)` states that a `plus` node with two float subtrees is itself a float subtree with a floating point addition. On the other hand, equations `(2)` and `(3)` state that a `plus` node with an integer subtree as well as a float subtree turns into a float subtree with the addition being a floating point addition and its integer subtree explicitly converted into a float subtree.

It is precisely this terseness of the specification of rather complex problems that makes equations attractive for the construction of compilers.

3. The UCG-E Specification Language

The UCG-E specification language (Hamel, 1991) is a descendant of the UCG system (Hatcher, 1991) developed at the University of New Hampshire for the generation of code generators. The UCG-E system takes an equational specification and translates it into C++ code. The generated code consists of a lookup table and a set of C++ function definitions. Each specification has two parts: a *declaration section* containing the alphabet and variable declarations and an equational or *rule section*.

A specification is treated as a rewrite system. Each equation has the form

```
M := N
```

and represents a directed rewrite rule where the term N replaces the term matched by M. Two restrictions apply to the form of these rewrite rules. Let Q be a well-formed term and V(Q) be the set of variables in term Q, then

(1) The term M must be linear, i.e. each variable in M may appear only once.

(2) $V(N) \subseteq V(M)$.

The first restriction insures that if a rule matches an input term the variables in M are uniquely instantiated. The second restriction insures that instead of having to do a global unification of the variables we may copy the values of the variables from the left hand side to the right hand side during rewriting if necessary.

As a concrete example, here is the type checking example from the previous section rewritten in the UCG-E specification language.

```
/* Declaration section. */

/* Define the alphabet. */

%unary int;
%unary float;
%unary int_to_float;

%binary plus;
%binary int_plus;
%binary float_plus;

/* Define variables. */

%var TERM left;
%var TERM right;

%%

/* Rule section. */

plus(int(left), int(right)) := int(int_plus(left, right));

plus(float(left), int(right)) :=
      float(float_plus(left, int_to_float(right)));

plus(int(left), float(right)) :=
      float(float_plus(int_to_float(left), right));

plus(float(left), float(right)) :=
      float(float_plus(left, right));
```

Two features make UCG-E especially attractive for the use in industrial software production. These are the *term generating functions* and the *user action functions*. Both of these features essentially provide an interface from UCG-E to other programming environments and thus allow the integration of the UCG-E rewrite system in software systems written in C++.

UCG-E builds a term generating function for each symbol in the alphabet. The user may call these functions to build terms labeled by symbols from the alphabet. In addition of generating a term these functions make the new term known to the UCG-E term rewriting mechanism which attempts to apply any of the given rules to this newly generated term. In our specific application, a compiler, we use these term generating functions in the parser to build the abstract syntax tree.

The user action functions, on the other hand, allow the use of side effects in the rewrite rules. These side effects could take on the form of I/O, a symbol table mechanism, or any other action which lies outside the realm of efficient equational processing. User action functions are distinguished symbols in the alphabet definition of the specification and may only appear on the right hand side of the rewrite rules.

In general, term rewriting systems worry not only about the final answer, but also that the same answer is obtained via any competing rewrite sequences given the same input term. This is known as *confluency*. Rather than imposing any further restrictions on the form of the equations, UCG-E assumes the equational specification to be confluent. Currently this is not checked. Since UCG-E allows side effects to be used in its rewrite rules, the rewrite sequence is as much part of the solution as is the final answer and therefore a deterministic selection of the rewrite sequence is important. UCG-E selects rules by the order in which they appear in the specification. It guarantees that rules appearing earlier in the specification have a higher priority than rules appearing later in the specification.

The programming and debugging of equational specifications is greatly facilitated by the interactive debugger supported by UCG-E. The debugger allows the user to single-step rewrite sequences one rule at a time and to browse current state information.

4. Implementation of an Industrial Strength Compiler

Our compiler translates a superset of C++ (C++ extended by the notion of persistent objects) into ANSI 2.0 C++. It had a number of requirements which ultimately led to the decision to use UCG-E as the implementation tool rather than another equational specification language such as the ACT-System, OBJ2, or OBJ3 (Hansen, 1988, Futatsugi, 1985, Goguen, 1988) .

(1) One of the platforms for the compiler is the MS-DOS operating system, therefore code size was important. The UCG-E system generates a lookup table and a set of relatively small C++ routines which satisfied our space requirements.

(2) It was important to be able to interface to the C++ programming environment in order to take advantage of such things as parsers, lexical analyzers, and symbols table mechanisms written in C++. UCG-E provides two interface mechanisms, the term generating functions and user action functions.

(3) At the time of the design of the compiler target platforms had not yet been defined, therefore portability was critical. Since UCG-E translates the equational specification into C++, we felt that it was the obvious choice for portability.

(4) Compilation speed was critical. UCG-E casts a pattern recognition engine in C++ for fast rewriting, rather than interpreting the equations at runtime.

The UCG-E specification consists of roughly 250 equations which also includes a code generator. (We implemented the code generator with UCG-E rather than with a separate tool, since our compiler generates C++ code and does not have to deal with optimal instruction selection and other problems related to machine code generation.) The lexical analysis phase as well as the parser were generated using *lex* and *yacc*, respectively. The symbol table mechanism and the type system were implemented in C++. The overall structure of the compiler follows the traditional model of lexical analysis, syntax analysis, semantic analysis, and finally code generation (Aho, 1986) .

We decided to compare the compile time of our compiler to that of Free Software Foundation's GNU C++ compiler, an extremely efficient compiler. This comparison was somewhat contrived, since the two compilers share neither the same source language nor the same target language, but it gave us a good idea of the efficiency of our compiler. Our measurements were done on a NeXT Computer running the Mach operating system with 16Mbyte of memory. The numbers cited reflect only the runtimes of the compilers proper without any pre-processing.

```
Our Compiler: 230 lines/sec
GNU C++:      750 lines/sec
```

These numbers indicate that we are currently running about three times slower than the GNU C++ compiler. Analyzing the execution time of our compiler even further we found that the total time breaks down in the following way:

```
Type System/Symbol Table Management:  34%
Equational Processing:                30%
Memory Allocation/Deallocation:       28%
Misc. (I/O, strcmp, etc.):             5%
Syntax Analysis:                       2%
Lexical Analysis:                      1%
```

Contrary to our own intuition we found that the equational processing is not the only major bottleneck in our compiler. We feel that this is a rather positive result and leaves plenty of opportunities for work to make our compiler more efficient. It should be mentioned again that the equational processing includes not only the semantic processing but also code generation.

5. Conclusions and other Observations

We have used the equational specification language UCG-E to construct a commercial compiler. During the construction of this compiler the semantics of the extended C++ source language changed many times, but we were able to keep up with these changes within the compiler mainly because of the easy extensibility of the semantic analysis phase of the compiler due to the high-level equational notation.

The compiler now runs on radically different platforms such as MS-WINDOWS, MS-DOS, and various flavors of UNIX. Porting to these diverse platforms was greatly facilitated by the use of the UCG-E specification language. Instead of changing the specification of the compiler to suit a particular system, we adapted the code generator of UCG-E to the target system, thus completely avoiding system dependencies in the specification of the compiler, keeping it clean and readable.

The development of the equational specification for our compiler was greatly eased by the UCG-E debugger. Rather than taking the traditional debugging approach of "dump the intermediate representation and see what went wrong", UCG-E essentially enabled us to single-step the compiler at a very high level, allowing errors to be traced relatively easily.

In conclusion, equations proved to be an extremely powerful implementation device for industrial strength compilers. The efficiency of the generated compilers is comparable to that of hand coded compilers. We also feel that the equational notation produces compilers which are easier to understand, maintain, and extend.

Looking into the future, we envision that a high-level specification language such as UCG-E could be used to define parts of a compiler completely system independently, making it sharable in much the same way as definitions for parsers and lexical analyzers are sharable today.

Acknowledgements

I thank Phil Hatcher for many inspiring discussions and for providing us with the source code of the original UCG system. Many thanks also to Jonathan Robie for patiently reading earlier drafts of this paper and providing many helpful hints and comments.

References

Aho, 1985.
Aho, A. and Ganapathi, M., "Efficient Tree Pattern Matching: an Aid to Code Generation.," *12th Annual ACM Symposium of Principles of Programming Languages*, pp. 334-340 (1985).

Aho, 1986.
Aho, Alfred A., Sethi, Ravi, and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, Massachusetts (1986).

Fraser, 1988.
Fraser, C. and Wendt, A., "Automatic Generation of fast optimizing Code Generators," *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 79-84 (1988).

Futatsugi, 1985.
Futatsugi, K., Goguen, J., Jouannaud, J., and Meseguer, J., "Principles of OBJ2," *Proceedings 12th ACM Symp. on Principles of Programming Languages*, pp. 52-66 Association for Computing Machinery, (1985).

Glanville, 1977.
Glanville, R., "A Machine independent Algorithm for Code Generation," *PhD thesis*, University of California at Berkeley, (1977).

Goguen, 1988.
Goguen, J., Kirchner, C., Kirchner, H., Megrelis, A., Meseguer, J., and Winkler, T., "An Introduction to OBJ3," *Lecture Notes in Computer Science* **308** pp. 258-263 Springer-Verlag, (1988).

Gordon, 1979.
Gordon, Micheal J. C., *The Denotational Description of Programming Languages,* Springer-Verlag, New York (1979).

Hamel, 1991.
Hamel, Lutz H., "UCG-E Language Definition and User's Guide," *Technical Report*, BKS Software GmbH, (1991).

Hansen, 1988.
Hansen, H., "The ACT-System: Experiences and Future Enhancements.," *Lecture Notes in Computer Science* **332** pp. 113-130 Springer-Verlag, (1988).

Hatcher, 1986.
Hatcher, P. and Christopher, T., "High-Quality Code Generation via bottom-up Tree Pattern Matching," *13th Annual ACM Symposium on the Principles of Programming Languagues*, pp. 119-130 (1986).

Hatcher, 1988.
Hatcher, P. and Tuller, J., "Efficient retargetable Compiler Code Generation.," *IEEE International Conference on Computer Languages*, (1988).

Hatcher, 1991.
Hatcher, P., "The Equational Specification of Efficient Compiler Code Generation," *Comp. Lang.* **16(1)** pp. 81-95 Pergamon Press, (1991).

Herman, 1991.
Herman, M., Kirchner, C., and Kirchner, H., "Implementations of Term Rewriting Systems," *The Computer Journal* **34** pp. 20-33 (January 1991).

Huet, 1980.
Huet, G. and Oppen, D, "Equations and Rewrite Rules: A Survey," *Formal Languages Theory*, pp. 349-405 Academic Press, (1980). Edited by R. Book

Johnson, 1975.
Johnson, S., "YACC - Yet Another Compiler Compiler," *Computer Science Technical Report #32*, Bell Telephone Laboratories, (1975).

Knuth, 1968.
Knuth, D. E., "Semantics of Context-Free Languages," *Math. Systems Theory* **2** pp. 127-145 (1968).

Lee, 1989.
Lee, Peter, *Realistic Compiler Generation,* The MIT Press, Cambridge, Massachusetts (1989).

Lesk, 1975.
Lesk, M., "LEX - Lexical Analyzer Generator," *Computer Science Technical Report #39*, Bell Telephone Laboratories, (1975).

O'Donnell, 1985.
O'Donnell, M., *Equational Logic as a Programming Language,* The MIT Press, Cambridge, Massachusetts (1985).

Paakki, 1991.
Paakki, J., "Prolog in Practical Compiler Writing," *The Computer Journal* **34(1)** pp. 64-72 (1991).