IJETST- Vol.||03||Issue||06||Pages 4068-4073||June||ISSN 2348-9480 2016

Open access Journal International Journal of Emerging Trends in Science and Technology

Impact Factor: 2.838 **DOI:** http://dx.doi.org/10.18535/ijetst/v3i06.04

Extensible Compiler

Authors

Sushama[1], Mrs Reema[2]

[1]M-Tech student of Department of Computer, Science and Engg, Sat Kabir Institute of Technology and Management Bahadurgarh,Haryana-124507

Email: *Sushama.angel@gmail.com*

[2]A.P in CSE Dept, Sat Kabir Institute of Technology and Management Ladrawan, Bahadurgarh-Haryana-124507

Email: *arorareema@live.com*

ABSTRACT

A compiler is a computer program or a set of program which converts the data from source code to object code, source code mean human understandable form whereas object means machine understandable form i.e.binary language. The compilers made till know are used to transform the a specific language and provides only the features which are added in it while designing the compilers, no other features are supported by the compiler other than those mentioned while designing it. This problem can be solved using extensible compilers. The basic idea used here is to extend a programming language by by adding new syntax, features etc. through adding extension modules which act as plug-ins for the compiler. Certain challenges are faced while building such compiler like creation of extensible that are simultaneously powerful, to allow effective extensions, convenient to make these extensions easy to write; and composable ,to make it possible to use independently-written extensions together.

In this paper, I have tried to make such a compiler which can act as a plug-in and extend a compiler by adding features and syntax to it .

Keywords: *Source code, Object code, syntax*

INTRODUCTION

Extensible programming is a term used in computer science to describe a style of computer programming that focuses on mechanisms to to extend the programming language, compiler and runtime environment. This paper is also about compilers and programming languages. A good compiler is necessary to do productive programming,a good programming language and compiler improves the productivity of the programmer.

Different languages are made in this regard,some of them excel at symbolic manipulation, some at numeric computation ,while some others in more specific domains. Programmers often find it useful to build custom language extensions that add abstractions or checking to the task of hand.

For example, James Gosling's made ace preprocessor which makes building specialized graphics operators an easy task. Kohler's Click router uses a special language to describe outer configurations and module properties. Krohn's tame preprocessor provides convenient syntax for Mazières's libasync. Engler's Magik allows userspecified checks and code transformations. Holzmann's Uno enables user specified flow-sensitive checks. Torvalds's Sparse adds function and pointer annotations

so it can check function pairings and address space constraints. These tools duplicate the parsing and semantic analysis required of any C

compiler. Engler built Magik by editing the GNU C compiler; most of the other tools duplicate the work of a compiler without being derived from one. Both approaches—editing an existing compiler or starting from scratch—require significant effort, so only the highly motivated tend to write these tools. These tools would be easier to build if the compiler had been more readily extensible and reusable.

The basic idea used here is that the compilers can and should allow programmers to extend programming languages with new syntax, features, and restrictions by writing extension modules that act as plugins for the compiler. We call such compilers extension-oriented. Here I have proposed extension-oriented syntax trees (XSTs) as a mechanism for building extension-oriented compilers and then evaluated them in the context of an extension-oriented compiler for the C programming language.

A compiler structured around XSTs makes it possible to implement derived languages as a collection of small, mostly independent extension modules. Thus, the target user of an extension-oriented compiler is a would-be language implementor who lacks the time or expertise to write a compiler from scratch. Extensibility via plug-in modules is the dominant extensibility mechanism in today's software—operating systems, web browsers, editors, multimedia players, and even games use plug-ins—and I believe that compilers will eventually adopt this model.

RELATED WORK

A number of techniques are there for making extensible compilers. Hence, to carry out the work, large number of papers had to be surveyed. Lots of information was collected. All these technique are used for giving high ranking to their web pages. Programmers have been exploring ways to extend programming languages for as long as they have been programming. We six main threads of language extension research:

1. macros
2. extensible languages
3. attribute grammars
4. term rewriting systems
5. modular compilers
6. extensible compilers.

Macros

Macros were perhaps the earliest programmer-controlled way to raise the level of abstraction of a language. In 1959, McIlroy was one of the first to use macros to raise the level of abstraction of a compiled source language [40]. Although his paper gives an example implementation for Algol, the bulk of the text is concerned with applying a macro system to an assembly language. This system, like most since, expanded macros by text substitution into templates.

Extensible languages

Macros demonstrated the utility of programmer-defined portions of a language, and by the 1970s, extensible languages were a popular idea. The term ''extensible'' is problematic: it usually means ''more flexible than normal,'' so it only has a concrete meaning in context. For example, Algol 68 may have been responsible in part for kick-starting interest in tensibility, but the term meant something different than what today's programmers would mean

Attribute grammars and term rewriting systems

Knuth introduced attribute grammars as a formalism for defining the semantics of context-free languages; they describe computations on the parse tree. More recent systems, such as Silver have built extensible language translators around attribute grammars. Term rewriting systems are a very different formalism based on syntactic substitution, like macros. They differ from macros in that rewrite rules apply not just to the original text but also to rewritten output. The ASF+SDF and Stratego systems are recent language translation systems based on term rewriting.

Attribute grammar systems and term rewriting systems suffer from the same limitation: because both formalisms are Turing-complete, systems

based on them typically do not provide any other computation mechanism, making the systems elegantly simple for some computations and frustratingly awkward for others. Attribute grammars easily express type checking and other local analyses. Term rewriting works best for problems already framed as repeated rewriting, like peephole optimization or lambda calculus evaluation . Neither approach is particularly convenient for nonlocal program manipulations or algorithms like data flow analysis.

The work described in this dissertation adopts the idea of attributes as a mechanism for allowing extensions to interact and for structuring computations like type checking.

It also adopts and refines the convenient pattern-based syntax manipulations of term rewriting systems. Importantly, this work does not require either as the only computation mechanism. Instead, it relies on a general-purpose programming or ML for the implementation of functionality that is most naturally expressed in such a language.

Modular compilers

Many people have built clean, modular compilers for research and teaching. One example, targeted at research, is the SUIF compiler infrastructure. SUIF was focused primarily on modularity in the back end, to facilitate research on issues related to code generation and performance, not on front end issues like syntax extensions or type checking.

Another example, targeted at teaching, is Sarkar et al.'s nanopass compiler framework . The nanopass framework defines each translation pass using a notation similar to Scheme's macro patterns, along with grammars describing the input and output grammars for the translation pass.The nanopass infrastructure uses the grammars to create a parse tree that is accessed only via pattern matching. Unlike this work, the nanopass infrastructure makes no attempt at extensions or composition of extensions. The compiler writer threads the individual nanopasses together explicitly to create the overall compiler. The success of the nanopass approach for teaching a compiler course suggests that the XST interfaces should also be easy to use for programmers who are not compiler experts.

Extensible compilers and compiler toolkits

Necula's C Intermediate Language (CIL) was a step closer to a compiler with an extensible front end. CIL provides a simple IR-like representation for C programs and makes it easy to write new programs that use the CIL interface to transform existing C programs. CIL's extensions can do type analyses and make semantic language changes, but they cannot add new syntax to the language.

Nystrom's Polyglot and Grimm's xtc are extensible compiler toolkits for Java and C, respectively. Both provide the power targeted by extension-oriented compilers, but they lack composability of extensions. They are toolkits for writing compilers rather than extensible compilers themselves. Both also require intimate knowledge of the compiler internals, which our work avoids. For example, both systems require extension writers to learn the Java data structure rpresentation of the input programs, while extension-oriented syntax trees allow extension writers to manipulate input programs in terms of the already-familiar concrete syntax.

PROPOSED WORK

In this paper three main artifacts are described.

1) The first is a set of language interfaces that enable the creation of extension-oriented compilers. Collectively, these interfaces provide access to a data structure called an extension-oriented syntax tree(XST).
2) The second artifact is an extension-oriented compiler for C, written using XSTs; this compiler is called xoc.
3) The third is a collection of extensions to xoc. The central challenge in creating an extension-oriented compiler is to design and expose an extension interface that:

1. is powerful enough to implement actual extensions.
2. does not require extension writers to be compiler experts.

3. allows independently-written extensions to be used together.

A single solution to all three challenges is to structure the entire compilation process around syntax trees that are first-class language objects with well-integrated interfaces.

These syntax trees are called extension-oriented syntax trees or XSTs. The four key interfaces to XSTs are extensible grammars, which define the conversions from text to

XSTs; syntax patterns, which allow allow manipulation of XSTs using concrete syntax; canonicalizers, which transform XSTs into a canonical representation; and attributes, which provide a mechanism for structuring and sequencing computations and analyses on XSTs.

We have implemented these interfaces by modifying a small language we designed called zeta. Zeta has a small, simple implementation, making it an ideal test bed for experimenting with XSTs. Even so, the ideas behind XSTs are in no way tied to zeta: XSTs could be added, with perhaps more effort, to any standard programming language.

Our Approach

The challenge in building an extension-oriented compiler is to create an extension mechanism that is powerful enough to implement the extensions people want to write, but at the same time to keep the extension mechanism limited enough that extensions can be written without detailed knowledge of the compiler and that multiple, independently-written extensions can be used simultaneously. In short, the challenge is

to build an extension mechanism in which extensions are powerful, simple to write, and composable. To meet this challenge, this dissertation proposes the use of extension-oriented syntax trees (XSTs). An XST is a conventional data structure—a parse tree—used via four unconventional interfaces: extensible grammars, which define the conversions from text to XSTs; syntax patterns, which allow manipulation of XSTs using concrete syntax; canonicalizers, which transform XSTs into a canonical representation; and attributes, which provide a mechanism for structuring and sequencing computations and analyses on XSTs. Extensible grammars, syntax patterns, and canonicalizers make extensions powerful yet simple to write. Attributes provide the connective glue that allows extensions to reuse the compiler core and to cooperate with each other to carry out computations like type checking an expression that combines features from multiple extensions. The XST runtime support can be implemented by a library, but the four interfaces require syntactic and semantic changes to the language the compiler is written in.

Implementation

This paper describes three main artifacts: first, a set of language interfaces that provide support for XSTs; second, an extension-oriented compiler for C called xoc, implemented using XSTs; and third, a variety of extensions written using xoc, including recreations of the functionality of Alef and Sparse. Collectively, these artifacts that demonstrate that XSTs can be used to create an extension-oriented compiler that meets the three goals above: extensions are powerful, simple to write, and composable.

XSTs provide the interface that connects the extensions to xoc itself.

Adding XSTs to a programming language requires more than just writing a library: XST support is tightly integrated into the language itself, with its own syntax and semantics. For our implementation, we added the XST interfaces to a simple new language we designed called zeta. Using our own small language made it easy to experiment with and refine the ideas behind XSTs, but XSTs are not tied to zeta: one could add support for them to any standard programming language.

CONCLUSION

Compilers can and should allow programmers to extend programming languages with new syntax, features, and restrictions by writing extension modules that act as plugins for the compiler. In

such a system, which we have named an extension oriented compiler, the extension mechanism must be powerful enough to implement the extensions programmers want. At the same time, it must be simple enough that extensions are short and do not require detailed knowledge of the base compiler. Finally, extensions need to be composable, so that a programmer can use multiple, independently-written extensions together in a single program.

ACKNOWLEDGMENT

I would like to thank my guide Mrs. Reema Sachdeva for her indispensible ideas and continuous support ,encouragement ,advice and understanding me through my difficult times and keeping up my enthusiasm, encouraging me and building up my confidence during the completion of this work. A special thanku note to my parents for their infinite supply of love which kept me going. They always inspired me to follow my instincts.

REFERENCES

1. Andrew W. Appel. Modern Compiler Implementation in C. Cambridge University Press.
2. Apple Computer, Inc.. Dylan Reference Manual..
3. Ken Arnold, James Gosling, and David Holmes. The Java Programming Language. Prentice Hall,
4. Jonathan Bachrach and Keith Playford. The Java Syntactic Extender (JSE). Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 31-42, Tampa Bay, Florida, United States.
5. Jason Baker and Wilson C. Hsieh. Maya: Multiple Dispatch Syntax Extension in Java. Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pages 270-281, Berlin, Germany.
6. Alan Bawden. Quasiquotation in Lisp. Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM '99), pages 4-12, San Antonio, Texas, United States.
7. Thomas Anthony Bergan. Typmix: A Framework For Implementing Modular, Extensible Type Systems. Master's thesis, University of California Los Angeles
8. Computer History Museum. Fellow Awards | 1997 Recipient Dennis Ritchie. Available online at http://www.computerhistory.org/fellowawards/index.php?id=71.
9. William Clinger and Jonathan Rees. Macros that Work. Proceedings of the 1991 ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages, pages 155-162.
10. Coverity. Linuxbugs. http://linuxbugs.coverity.com Offline. Available via archive.org at http://web.archive.org/web/*/http://linuxbugs.coverity.com
11. Brad J. Cox and Andrew J. Novobilski. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley.
12. Glen Ditchfield. An Overview of Cforall. Available online at http://plg.uwaterloo.ca/~cforall.
13. Eelco Dolstra and Eelco Visser. Building Interpreters with Rewriting Strategies. Proceedings of the 2002 Workshop on Language Descriptions, Tools, and Applications, Grenoble.
14. Gabriel Dos Reis and Bjarne Stroustroup. Specifying C++ concepts. Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 295-308.
15. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. Lisp and Symbolic Computation 5(4), pages 295-326.
16. Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as Inconsistent Behavior: A General Approach to

Inferring Errors in Systems Code. Proceedings of the 2001 Symposium on Operating Systems Principles, pages 57-72, Banff.

17. Dawson R. Engler. Incorporating Application Semantics and Control into Compilation. Proceedings of the USENIX Conference on Domain-Specific Languages (DSL '97), October 1997.
18. Bob Flandrena. Alef User's Guide. Plan 9 Programmer's Manual: Volume Two.
19. Bryan Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. Proceedings of the 2004 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy.
20. James Gosling. Ace: a syntax-driven C preprocessor, 1989. Available online at http://swtch.com/gosling89ace.pdf