An Extension-Oriented Compiler

by

Russell Stensby Cox

A.B., Computer Science
Harvard College, 2001

S.M., Computer Science
Harvard University, 2001

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

September 2008

Author ______________________________
Department of Electrical Engineering and Computer Science

Certified by ______________________________
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Certified by ______________________________
Eddie Kohler
Associate Professor of Computer Science, UCLA
Thesis Supervisor

Accepted by ______________________________
Terry P. Orlando
Chair, Department Committee on Graduate Studies

An Extension-Oriented Compiler

by

Russell Stensby Cox

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The thesis of this dissertation is that 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 compilers designed around this idea *extension-oriented*. The central challenge in designing and building an extension-oriented compiler is the creation of extension interfaces 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.

This dissertation proposes and evaluates extension-oriented syntax trees (XSTs) as a way to meet these challenges. The key interfaces to XSTs are grammar statements, a convenient, composable interface to extend the input parser; syntax patterns, a way to manipulate XSTs in terms of the original program syntax; canonicalizers, which put XSTs into a canonical form to extend the reach of syntax patterns; and attributes, a lazy computation mechanism to structure analyses on XSTs and allow extensions to cooperate. We have implemented these interfaces in a small procedural language called zeta. Using zeta, we have built an extension-oriented compiler for C called xoc and then 13 extensions to C ranging in size from 16 lines to 245 lines.

To evaluate XSTs and xoc, this dissertation examines two examples in detail: a reimplementation of the programming language Alef and a reimplementation of the Linux kernel checker Sparse. Both of these examples consist of a handful of small extensions used in concert.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Thesis Supervisor: Eddie Kohler
Title: Associate Professor of Computer Science, UCLA

“See this page of paper? It’s blank,” Scull said. “That, sir, is the most frightening battlefield in the world: the blank page. I mean to fill this paper with decent sentences, sir—this page and hundreds like it. Let me tell you, Colonel, it’s harder than fighting Lee. Why, it’s harder than fighting Napoleon.”

Larry McMurtry, *Comanche Moon*

Contents

Acknowledgements

I did not write this dissertation alone. I have had the good fortune to enjoy seven years at MIT working with and learning from a seemingly inexhaustible supply of amazing people. I cannot list them all, but I am grateful to everyone who has crossed my path.

Frans Kaashoek and Eddie Kohler have been incredibly supportive as thesis advisors. They pushed me and my research farther than I thought possible. I am proud to say that much of the work described in this dissertation was done with help from Tom Bergan and Austin Clements, both of whom contributed immense amounts of careful thought, hard work, and raw energy.

Frans, Eddie, Robert Morris, and John Guttag served on my thesis committee. I am indebted to them for their efforts to make this dissertation better.

Frans and Robert created an excellent research environment and encouraged me to dive into a new research area that was way over my head.

Finally, my parents and my wife, Sara O'Brien, helped keep me afloat while I learned to swim.

I gratefully acknowledge financial support from the Fannie and John Hertz Foundation and from Nokia Research Cambridge Center.

Thank you, everyone.

Introduction

This dissertation is in the area of compilers and programming languages. A good compiler and programming language can substantially improve programmer productivity. Some languages excel at symbolic manipulation, others at numeric computation, still others in more specific domains. Programmers often find it useful to build custom language extensions that add abstractions or checking tailored to the task at hand.

For example, Gosling's ace preprocessor makes it easy to build specialized graphics operators [20]. Kohler's Click router uses a special language to describe router configurations and module properties [35][36]. Krohn's tame preprocessor [37] provides convenient syntax for Mazières's libasync [39]. Engler's Magik [17] allows user-specified checks and code transformations. Holzmann's Uno [30] enables user-specified flow-sensitive checks. Torvalds's Sparse [55] 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 thesis of this dissertation is that 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*. This dissertation proposes extension-oriented syntax trees (XSTs) as a mechanism for building extension-oriented compilers and then evaluates 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. The target reader of this dissertation, however, is a compiler writer who wants to create an extensible compiler that the would-be language implementor can use. 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 we believe that compilers will eventually adopt this model. This dissertation shows one possible way to do that.

The rest of this chapter details the motivation for this work and defines what we mean by extension-oriented compilers and also extension-oriented syntax trees. It then gives an outline of the rest of the dissertation.

1.1. Motivation

Programmers are never fully satisfied with their programming languages. Once that dissatisfaction reaches a critical level, an energetic programmer will often modify the language or set out from scratch to create a new one. As an example, consider the venerable C programming language [32]. C is a popular language yet it has also spawned a large number of offspring, ranging from whole new languages like C++ [57], C# [38], Objective C [11], and Java [3] to variants adding individual features like the ones mentioned above. These systems represent a substantial amount of effort to adapt a popular language to make it better for one class of programmers or another.

For concreteness, consider these two extensions to C: the Alef language [69] and the Sparse program checker [55].

1.1.1. Alef

In the 1990s at Bell Labs, Phil Winterbottom was unhappy with C as a systems programming language and created a new language called Alef [69]. To a first approximation, Alef was C with extensions for the systems programming problems faced in the development of Plan 9 [48]. The main extensions were syntax and runtime support for multithreading, communication, exceptions, and polymorphism. Alef was used for a few large programs in Plan 9 and many smaller ones. The 1995 release of Plan 9 contained 30,000 lines of Alef source code [49]. In 1997, Dennis Ritchie was made a Computer History Museum Fellow; in an interview given at the time he named Alef as his favorite programming language [8].

The most commonly used extension in Alef was its support for typed communication channels and syntax for send and receive operations on those channels. For example, in this snippet from a real Alef program, the `alt` statement chooses between receiving (the <- operator) on two different channels, depending on which can proceed [49]:

```
Timer *timer = timerstart(dt);
alt{
case <-(timer->c):
     timerstop(timer);
     break;
case mouse = <-cmouse:
     task timerwaittask(timer);
     break;
}
```

Channels in Alef are typed, so that receiving from `cmouse`, a `chan(Mouse)`, evaluates to a `Mouse` structure.

By 1999, Winterbottom had moved on to other projects, and the remaining Alef users didn't want to maintain the compiler, so they rewrote their Alef programs in C. They used a library to provide the threading and communication functionality and gave up the other features along with the convenient syntax. In the end, they discarded a 26,000 line compiler and replaced it with a 3,000 line library and a clumsier interface. The above example now reads [50]:

```
Alt alts[3];
Timer *timer = timerstart(dt);
alts[0].c = timer->c;
alts[0].v = nil;
alts[0].op = CHANRCV;
alts[1].c = cmouse;
alts[1].v = &mouse;
alts[1].op = CHANRCV;
alts[2].op = CHANEND;
switch(alt(alts)){
case 0:
	timerstop(timer);
	return;
case 1:
	taskcreate(timerwaittask, timer, STACK);
	return;
}
```

In addition to losing the convenient syntax, this example also lost its type safety: the assignment to mouse is done by writing through the void* pointer alts[1].v; this is a common source of errors among programmers new to the library.

1.1.2. Sparse

The second example begins at SOSP in 2001, where Dawson Engler's research group from Stanford presented work that inferred implicit conventions in the Linux kernel source code and then identified places where the conventions were not followed [16]. These techniques found many serious bugs in the Linux kernel and eventually led to a company named Coverity. Coverity checked the Linux kernel for free, both as a service to the Linux community and as good publicity [10].

The Linux developers were grateful for the bug reports, but Linus Torvalds, the lead developer, eventually decided that stating the conventions explicitly would be easier and more accurate than inferring them. At the time, he said that inferring the conventions "is really manly. But it's manly in a very stupid way" [53]. He added explicit annotations to the kernel source, like these:

```
void __iomem *in_port;  /* Inbound port address */
void __iomem *out_port; /* Outbound port address */

void __user *in;        /* Data to be transferred in */
void __user *out;       /* Data to be transferred out */

void spin_lock(spinlock_t *lock)       __acquires(lock);
void spin_unlock(spinlock_t *lock)     __releases(lock);
```

The __iomem tags declare in_port and out_port to point at I/O memory; similarly, in and out point at user memory. The __acquires and __releases annotations say that spin_lock acquires lock and spin_unlock releases it. Any function that is not explicitly tagged must have no net effect on lock, so in an untagged function, there must be a balanced number of spin_lock and spin_unlock calls on every possible control flow path.

Torvalds then wrote a new C checker and compiler called Sparse to check that the code behaved according to the annotations. Sparse is implemented in 24,000 lines of C [55].

1.2. Goals

In both examples, an expert programmer wanted to extend C with features specific to his problem domain, and the best way he knew how was to write a whole new compiler that was tens of thousands of lines of code. This sets a high bar for language extension.

The goal of this work is to lower that bar significantly, so that language extensions can be created by programmers who are not compiler experts and by writing only tens or hundreds of lines of code. Equally important, this work aims to construct a framework in which multiple, independently-written extensions can be used simultaneously, so that programmers can pick and choose language extensions the same way they do code libraries and web browser plugins. We call such a compiler an extension-oriented compiler.

We envision that an extension-oriented compiler would be invoked by naming one or more extensions on its command line, as in

```
xoc -x alef -x sparse file.c
```

to compile `file.c` after loading both the `alef` and `sparse` extensions. Even if the two extensions were written independently by two programmers who did not coordinate, it should be possible to use both together to compile `file.c` as long as the two extensions do not make conflicting changes to the language (for example, redefining the same construct to mean different things).

Extensions in an extension-oriented compiler need to be powerful, able to change the parts of the language that programmers want to change; they must be simple, not requiring programmers to be experts in every detail of the base compiler; and they must be composable, so that different extensions that make compatible language changes can be used together.

Ideally, an extension-oriented compiler would be able to determine, for any pair of extensions, whether they make conflicting changes. In practice, if extensions are implemented using a general-purpose programming language, most of these questions are undecidable. This work is focused only on making composability of extensions possible, not on checking or guaranteeing it.

1.3. Alternate approaches

To understand the challenges in creating an extension-oriented compiler, it is useful to examine approaches that have been taken in the past. This section considers first an illustrative but imagined example, a so-called patch-oriented compiler, and then the real example of extensible compiler toolkits.

1.3.1. Patch-oriented compiler

Consider the following system, a patch-oriented compiler. A patch-oriented compiler is a wrapper around a traditional compiler; call the traditional compiler `tcc`. In the absence of patches, the patch-oriented compiler compiles an input file by running `tcc` on that input file. Patches change the behavior of the patch-oriented compiler by describing modifications to `tcc`'s source code: insert these lines, delete these, replace these. When invoked with options specifying one or more patches, as in

```
poc -p patch1 -p patch2 file.c
```

the patch-oriented compiler first applies `patch1` and `patch2` to `tcc`'s source code and recompiles `tcc`. Then it invokes the resulting modified `tcc` on the input `file.c`.

Patches written for the patch-oriented compiler have complete flexibility; they have the power to change any line of code in the compiler and thus to implement any imaginable and implementable change to the compiler's input language. Thus patches achieve the first goal for extensions, but they fall short of the other two. Patch writers need to understand the entire base compiler (or at least large portions of it) in order to write their patches, so patches are not easy to write. Also, patches may make semantically compatible changes to the language being compiled but still not be usable together: one patch may change the line numbers or lines depended on by a second patch. The patch-oriented compiler would not make a good extension-oriented compiler: its patches are certainly powerful, but they are neither simple nor composable.

1.3.2. Extensible compilers

A real approach with similar motivation to ours is that of extensible compiler toolkits, an old idea exemplified recently by the Polyglot compiler for Java [44] and the xtc compiler for C [28]. The main piece in an extensible compiler toolkit is a base compiler for a known language (let's say C) written in an object-oriented language (let's say Java). To build a language variant, an extension writer creates new Java classes that subclass and extend existing functionality; then he links them with the base compiler to create a new compiler. Extensible compiler toolkits usually provide convenient, simple interfaces to the core compiler functionality.

Extensible compiler toolkits make extensions powerful and simple, but they are not composable: each extension creates a new compiler. There is no way, as in our example above, to change the set of extensions being used at each invocation of the compiler. In most toolkits, there is not even a clear way to combine two extensions into a single compiler binary. Depending on the extensions, it may be more work to combine two existing extensions than to reimplement them from scratch in a single new extension.

Part of the reason that Polyglot and xtc have trouble handling multiple extensions is that the act of composing multiple extensions does not fit naturally into Java's object-oriented type system. In response to this, Polyglot's authors created J&, a Java extension that adds "nested intersection types" to model composition of compiler extensions [45]. A J&-based Polyglot requires less work to compose two extended compilers than the original Polyglot, but it still requires some manual effort. This thesis shows how to achieve general composability using a new approach.

1.4. 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. For example, the form of a syntax pattern depends on the language being compiled. In a traditional compiler, if `x` is a parse tree node, one might write

```
if(x->op == ExprAdd && x->left->op == ExprMul)
```

to look for an addition of a multiplication expression. Syntax patterns allow one to write instead:

```
if(x ~ ‘C.expr{\a*\b+\c})
```

Here, ~ is the pattern match operator, `‘C.expr` begins a pattern for the `C` grammar's `expr` symbol, and `\a*\b+\c` is the pattern itself, a C expression with wildcard slots `\a`, `\b`, and `\c`. Implementing this notation requires explicit support in the language the compiler is written in.

1.5. Implementation

This dissertation 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. Figure 1 illustrates these artifacts. 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

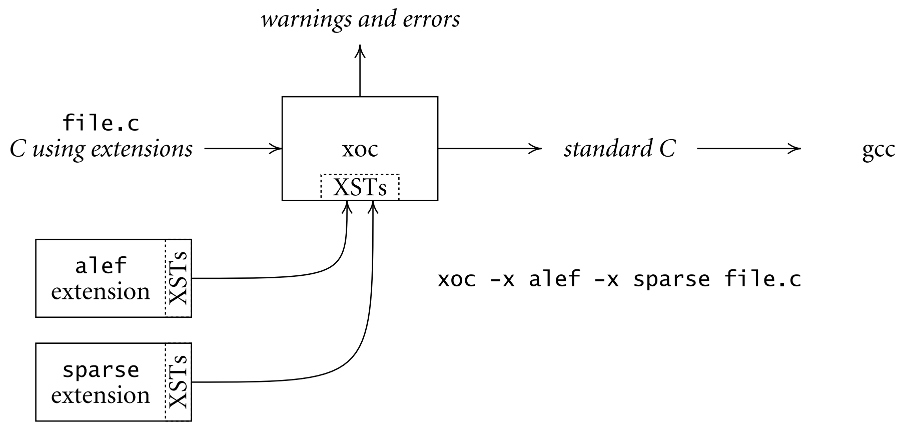

Figure 1. An illustration of the sample xoc command line `xoc -x alef -x sparse file.c`. Xoc loads the two extensions `alef` and `sparse` before processing `file.c`, which contains C source code that makes use of the changes to C introduced by the two extensions. Extension-oriented syntax trees (XSTs) provide the interface between xoc and the extensions. The output of xoc is a set of compiler warnings and errors along with a translation of the input file into standard C. Xoc then invokes the GNU C compiler to translate the standard C into an object file.

and semantics, as Chapter 3 explains. 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.

1.6. Contributions

This dissertation introduces extension-oriented compilers as an approach to programming language extension and as a research goal, and then it makes the following contributions toward that goal. The first and primary contribution is the identification and development of XSTs and their interfaces as a mechanism for building extension-oriented compilers. The second contribution is zeta, xoc, and implementations of the functionality provided by Alef and Sparse as extensions; collectively, these demonstrate the viability of XSTs as a base for building extension-oriented compilers. The third contribution is an evaluation of XSTs, both quantitatively, examining their performance considerations, and qualitatively, examining the overall ease of use. We hope that the XST interfaces described in this dissertation will influence compiler writers who want to add support for extensions to their own compilers.

1.7. Limitations

This work attempts to create an extension mechanism powerful enough that careful extension writers can, with minimal effort, create compiler extensions that can be used together to implement language extensions. The focus is on creating powerful extensions, not on protecting extension writers from that power. In particular, this work attempts only to make composability possible; it does guarantee composability in all cases, nor does it identify when two extensions are not composable. This work makes no effort to check the correctness of extensions: it is possible to write an extension that causes the compiler to loop forever or to crash or to generate an incorrect binary. Of course, programmers are not required to use a particular extension and would likely avoid a buggy compiler extension just as they avoid buggy code libraries.

There are also limitations specific to the xoc prototype. First, the generality of xoc comes at a cost: the xoc prototype runs significantly slower than hand-tuned traditional tools like the GNU C compiler and Sparse. Second, xoc is only a compiler front end: it compiles its input (C with extensions) into standard C and invokes a standard C compiler. This structure means that it cannot provide access to hardware features, like special instructions, that the C compiler does not. Removing both of these limitations would be interesting future work.

1.8. Outline

This dissertation is structured around the contributions. Chapter 2 examines previous approaches to programming language extension. Chapter 3 introduces extension-oriented syntax trees and their interfaces, implemented in the language zeta. Chapter 4 describes the implementation of an extension-oriented compiler for C using XSTs. Chapter 5 describes extensions to that compiler that recreate the functionality of Alef and Sparse as well as other extensions. Chapter 6 examines the performance of XSTs, xoc, and the extensions, and Chapter 7 discusses qualitative aspects of the system. Chapter 8 concludes. For completeness, two appendices give additional detail about aspects of the system: Appendix A gives a definition of the zeta language, and Appendix B lists the full source code for the extensions we have written.

Related Work

Programmers have been exploring ways to extend programming languages for as long as they have been programming. This dissertation draws on six main threads of language extension research: macros, extensible languages, attribute grammars, term rewriting systems, modular compilers, and extensible compilers. This chapter describes these areas to highlight this dissertation's intellectual debt as well as its contributions.

2.1. 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.

Lisp macros, introduced by Hart in 1963 [26], were one of the earliest non-textual approaches. Since Lisp already had a fixed representation of programs as data—S-expressions—Hart's macro system simply passed a macro invocation's arguments to the function responsible for computing the macro expansion. Macros implemented this way quickly became an essential part of the Lisp environment, though the exact details of the implementations continued to be refined for the next twenty years. Steele and Gabriel [56] give a detailed summary of the history. Quasiquotation—the use of backquote expressions like `‘(+ ,x 1)`—was developed in the mid-1970s, making macros much easier to write and understand. Many arcane details took another decade or so to work out fully [6]. Graham's *On Lisp* [21] is the canonical showpiece for the power of Lisp macros. The success of macros in Lisp inspired adaptations in Dylan [2], Java [4] [5], and many other languages.

The text-based and Lisp macro systems described so far suffer from the problem of unintended capture, in which a variable referenced in a macro body happens to share its name with an unrelated local variable defined and used around the macro invocation. For example, the imagined macro:

```
define swap(x, y) = begin local t; t = x; x = y; y = t; end;
```

will not work properly if one of its arguments is named `t`.

Hammer [22], working in Project MAC at MIT, was interested in macro processing as a method of syntax extension and suggested parsing each macro body only once and recording the parse structure. To instantiate the macro, his system would simply run the corresponding parse rules, without reparsing the entire token stream. Hammer's approach avoids the problem of unintended capture, although he does not mention this in his paper. Indeed, it seems that in 1971 when Hammer's work was published, unintended capture had not yet been recognized as a problem to be solved. For whatever reason, Hammer's approach did not catch on, perhaps because it is so tightly integrated with the parser. Even so, it is the same approach xoc takes to implement the syntax patterns described in Chapter 3. Unintended capture later became a cause célèbre in the Scheme macro community [9][15][34].

Macro systems automate code generation—a small amount of notation can be macro-expanded into a large amount of code—but they provide no access to the underlying compiler, making it difficult or impossible to check the types and syntaxes of macro arguments or even to check that the arguments make sense in the macro's code templates. As a result, macro systems are famous for the inscrutable error messages produced when an invalid argument passed to a macro results in an invalid code expansion; the compiler errors complain about code far removed from the actual mistake. C++'s templates are a popular example of this problem [14].

Macro systems have other, lesser drawbacks as well. For example, macros cannot affect code that does not invoke them. This makes macros unsuitable for implementing restrictions or semantic changes in existing programs.

Extension-oriented compilers aim to provide the power of macros but also expose existing compiler functionality, so that extensions can integrate well into the language and do better checking and handling of their arguments. Syntax patterns, described in Chapter 3, were inspired by Lisp's quasiquotation operator.

2.2. 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 extensibility, but the term meant something different than what today's programmers would mean. Tanenbaum's Algol 68 tutorial, published in 1976, gives a typical explanation [59]:

> Another principle, related to that of orthogonality, is the principle of extensibility. Algol 68 provides a small number of primitive data types, or modes, as well as mechanisms for the user to extend these in a systematic way. For example, the programmer may create his own data types and his own operators to manipulate them.

That is to say, Algol 68 had equivalents to C's `struct` and `union` or, more accurately, ML's product and sum types. Extensible language research waned as new languages with explicit support for data abstraction and defining data types gained traction: CLU, ML, Simula, and Smalltalk, just to name a few.

The exact features being added by extensibility are less important, for our purposes, than the mechanism by which they are added. The flurry of extensibility research in the 1970s focused on linguistic approaches, building extensibility features into the language. This dissertation argues that to go beyond the extensibility of data abstraction and abstract data types, it is more effective to implement extensibility at the level of the compiler, which already provides all the terms and concepts needed to explain how to change the language.

2.3. 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 [33]. More recent systems, such as Silver [61][62] 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 [60] and Stratego [64] 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 [63] or lambda calculus evaluation [13]. Neither approach is particularly convenient for non-local 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 language similar to C, Java, or ML for the implementation of functionality that is most naturally expressed in such a language.

2.4. Modular compilers

Many people have built clean, modular compilers for research and teaching. One example, targeted at research, is the SUIF compiler infrastructure [67]. 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 [51]. 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. Like the work described in this dissertation, 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.

2.5. Extensible compilers and compiler toolkits

Necula's C Intermediate Language (CIL) [43] 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 [44] and Grimm's xtc [28] are extensible compiler toolkits for Java and C, respectively. Both provide the power targeted by extension-oriented compilers, but they lack composability of extensions, as discussed in Chapter 1. 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 representation of the input programs, while extension-oriented syntax trees allow extension writers to manipulate input programs in terms of the already-familiar concrete syntax.

Inspired by Polyglot's lack of composability, Nystrom *et al.* created a Java variant J& that adds a type system feature called nested intersection, which enables composability of extensions in an object-oriented type system [45]. A new version of Polyglot, written in J&, provides composability of extensions but still requires intimate knowledge of compiler internals, such as the form of the abstract syntax tree.

2.6. Summary

There is a rich history of work related to extension-oriented compilers, but none of these attempt the combination of broad extension scope, simplicity of extension writing, and composability of extensions that we want to provide. Macro systems are a powerful code-rewriting technique, but they do not allow access to other parts of the compiler. The extensibility provided by languages in the 1970s resulted in the data abstraction now taken for granted in today's languages. Attribute grammars and term rewriting systems are both powerful tools that excel in some areas but make others more difficult. The work described in this dissertation takes the most convenient features and mixes them into a general-purpose language. The various extensible compilers built so far require their users to be intimately familiar with the base compiler and do not allow composability of extensions. These compilers go to great lengths to expose their internals. In contrast, the extension-oriented compiler described in this dissertation goes to great lengths to hide its internals, exposing instead the powerful but restricted interfaces described in the next chapter.

Extension-Oriented Syntax Trees

This dissertation describes three main artifacts. 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). The second artifact is an extension-oriented compiler for C, written using XSTs; this compiler is called *xoc*. The third is a collection of extensions to xoc. This chapter focuses on XSTs and their interfaces, which enable extension-oriented compilers. This chapter uses some small fragments of xoc as concrete examples to illustrate the XST interfaces. Chapter 4 focuses on xoc in more detail, and Chapter 5 focuses on the extensions. Although the examples in this chapter are taken from an extension-oriented C compiler, the XST interfaces described here are not specific to C and should work just as well to build extension-oriented compilers for other languages.

As discussed in Chapter 1, 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.

3.1. Overview of extension-oriented syntax trees

XSTs are the central data structure in an extension-oriented compiler, used both to represent the initial parse of a program as well as the result of any transformation steps during compilation. This section summarizes their general semantics. Subsequent sections discuss the key interfaces—extensible grammars, syntax patterns, canonicalizers, and attributes—in detail.

Zeta programs define the structure of an XST by giving a context-free grammar describing how to parse the corresponding input. Extensions can augment an existing grammar, adding new grammar symbols and rules or adding new rules for existing symbols. For example, an extension might define a new kind of expression, adding a rule to a pre-existing `expr` symbol. After defining the grammar, a zeta program can create an XST by asking zeta to parse a given piece of text (for example, an input file). Zeta also uses the grammar to define a rich type system for XSTs. For example, if the language grammar distinguishes between statements and expressions, then the XST type system also distinguishes them as different subtypes of XST; it enables both static type annotations and dynamic type checks.

The parser returns a pointer to an XST, but there the similarity to traditional compilers ends. Zeta programs manipulate and examine an XST not as a traditional data structure but in terms of the corresponding input syntax. For example, in a traditional compiler written in C, checking whether an XST denotes an addition of a multiplication might be written `x->op == ExprAdd && x->left->op == ExprMul`. Using syntax patterns, the same check is `x ~ '{\a*\b+\c}`. Extensions create new XSTs using almost identical notation. Zeta also allows the definition of canonicalizers, which are applied to both XSTs and syntax patterns to put syntax into a standard form before matching. For example, in a language with infix expressions, a canonicalizer might remove parentheses from the syntax tree so that `(1*2)+3` and `1*2+3` both match the pattern above.

For generic traversal and transformation of XSTs, zeta provides two more traditional functions: `xstsplit(x)` returns a list of the children of the XST node `x`, and `xstjoin(x, kids)` creates a copy of `x` but with a potentially different set of child nodes. This example illustrates an important point about XSTs: they are immutable. It is not possible to overwrite or edit an XST node. Instead, zeta programs create new XSTs that reuse existing subtrees.

To structure the analysis and compilation of XSTs, zeta provides attributes. An attribute is a function taking a single XST node as its argument. Often, an attribute is specific to a certain grammar subtype. For example, an XST for an expression might have an attribute `type` giving the type of the expression (`int`, `void`, `float`, and so on). An attribute function is invoked by referring to the attribute value using dot notation: if `x` is an XST, then the first reference to `x.type` invokes the function computing `x`'s `type` attribute. Zeta caches the computed value: future references to `x.type` will use the cached value instead of recomputing it. The fact that XSTs are immutable makes caching computed attribute values reasonable.

To make it possible for an XST node to find its place in a larger XST (for example, a variable reference `x` needs to locate the corresponding declaration of `x`), each XST node has a built-in attribute named `parent`. Creating a new XST that reuses a node from another XST would result in that node's `parent` attribute being ill-defined: the node would not have one parent, but two. Instead of allowing this situation, any time a new XST is created using a node that is already in another XST, that node is not used directly; instead, a complete copy (a so-called deep copy) is made, and the copy is used in the new XST. This ensures that any XST node has a unique parent and thus a well-defined `parent` attribute. It is often useful to examine the original context of a copied XST node. The built-in attribute `copiedfrom` exposes this relationship to zeta programs. A naïve implementation of these copying semantics would be quite expensive; as discussed in Chapter 5, zeta simulates such copying by making

(a)
```
extensible grammar C99
{
    ...
    %left Add Shift;
    %priority Add > Shift;
    ...

    expr: name
        | expr "+" expr [Add]
        | expr "-" expr [Add]
        | expr "<<" expr [Shift]
        | expr ">>" expr [Shift]
        ...
}
```

(b)
```
grammar Rotate extends C99
{
    expr: expr "<<<" expr [Shift]
        | expr ">>>" expr [Shift];
}
```

(c)
```
grammar AlefIter extends C99
{
    %right AlefIter;
    %priority Shift > AlefIter
                    > Relational;
    expr: expr "::" expr [AlefIter];
}
```

Figure 1. A grammar and two extensions: (a) a fragment of the grammar for C99 showing a few expression rules; (b) an extension adding new operators <<< and >>> as syntax for bitwise rotate; and (c) an extension adding a new operator `::` as syntax for iterator expression as in Alef. The `%left`, `%right`, and `%priority` lines define operator precedences, and the `[Add]`, `[Shift]`, and `[AlefIter]` annotations attach precedences to specific rules. Alef defines its iterator operator to occupy a new level between the shift and relational operators.

XSTs copy-on-reference. Copies of XSTs start with no cached attributes: they do not inherit any cached attributes from the original, because the attribute values might depend on the placement of the XST in the overall program.

Not all computations on XSTs are functions of a single XST node. For example, a compiler might need to decide whether one type can be automatically converted to another, a decision that is a function of two XST nodes. To accommodate these cases, zeta also provides *extensible functions*, which are a non-lazy generalization of attributes.

The rest of this chapter describes each of these aspects of XSTs in detail.

3.2. Extensible grammars

Extensible grammars define the relationship between input text and XSTs. As an example, Figure 1(a) shows a few rules concerning expressions in a grammar for the 1999 version of ANSI C. Figure 1(b) shows one possible extension, grammar rules that an extension could use to add new <<< and >>> operators intended as bitwise rotate. Figure 1(c) shows a second possible extension, grammar rules that add an iterator operator `::` in the style of the C-like language Alef [69].

The grammar rules define the structure of the XSTs that zeta builds. For example, the XST for an `expr` node corresponding to the `expr + expr` rule has two children, both `expr`s themselves.

The grammar rules also define a subtype hierarchy among XSTs, based on which grammar rules are used in a particular XST. An XST tree that uses only rules from the C99 grammar has type C99. Similarly, an XST that uses only rules from C99 and AlefIter has type `C99+AlefIter`. Types are independent of the order in which extensions are added to the base grammar: `C99+AlefIter+Rotate` and `C99+Rotate+AlefIter` are the same type, describing XSTs using only rules from C99 or the two extensions AlefIter and Rotate. Since any XST using only C99 rules is also an XST that uses only C99 and AlefIter, any XST of type C99 is also an XST of type C99+AlefIter. That is, C99 is a subtype of C99+AlefIter. Similarly, C99+AlefIter and C99+Rotate are both subtypes of C99+AlefIter+Rotate.

In an extension-oriented compiler, extensions must be prepared to handle XSTs that use grammar rules specific to other, unknown extensions. The wildcard ? denotes all extensions to a base grammar, including extensions not yet loaded or even written. C99, C99+AlefIter, and C99+AlefIter+Rotate are all subtypes of C99+?. Because ? denotes all extensions, C99+AlefIter+? is just a longer name for the type C99+?.

The particular set of grammars being used by a given XST defines one axis of the subtype hierarchy. The particular grammar symbol the XST represents defines a second axis. For example, a C99 XST whose root node is an expr has type C99.expr; similarly, a stmt node has type C99.stmt. Both C99.stmt and C99.expr are subtypes of C99, but neither is a subtype of the other. Xoc uses this grammar symbol-based subtyping heavily. For example, the function that computes the type of an expression is defined to accept a (C99+?).expr, not a C99+?, to statically detect mistakes like asking for the type of a statement.

Allowing extensions to introduce new grammar rules creates the possibility that the resulting grammar will be ambiguous for a given input: there might be multiple ways to parse the input. For example, one extension might introduce a >>> b as syntax for a bitwise right rotate, while a second might introduce the same notation for a logical right shift (like in Java). Using both extensions together to parse a program containing such an expression would be ambiguous. Zeta detects such situations during parsing and reports the ambiguity.

3.3. Syntax patterns

The grammar interface defines a constructor for XSTs. Syntax patterns provide the interface for accessing XSTs (discovering their contents and traversing them). In traditional compilers, the abstract syntax tree is represented as a data structure in the host language (for example, a C struct, or a Java class, or an ML record). This is certainly a powerful interface, but it requires that extension writers learn the mapping between input syntax and internal representation. Worse, giving extensions such low-level access to the representation makes it difficult for an extension to manipulate syntax trees created by another, unknown extension. Zeta addresses both of these by hiding the low-level XST representation and providing an interface via syntax patterns.

A syntax pattern is a fragment of code in the input source language with placeholders called *slots* denoted by backslashes. For example, `C99.expr{\a + \b} is a syntax pattern for an expression that is itself the combination of two other expressions using the expr: expr + expr rule. (Semantically, it is a pattern for an addition of two other expressions.) Syntax patterns can be used for two different purposes. One is to pattern match, and thus to traverse, existing XSTs. If x is an XST matching that pattern, then x ~ `C99.expr{\a + \b} evaluates to true and binds the new variables a and b to the two children of x. The second use is to create new XSTs. After the match just considered, evaluating the expression `C99.expr{\b + \a} would create a new XST like x but with its children in the opposite order. The use in the first example is a *pattern match*; the use in the second example is a *pattern constructor*. Since

XSTs are immutable, pattern constructors (which create new XSTs) substitute for the mutation of a traditional interface.

Syntax patterns are not limited to the single rule applications demonstrated by the example just given. As a more complex example, a code snippet like

```
if(x ~ ‘C99.expr{\a << \b << \c})
     x = ‘C99.expr{\a << (\b+\c)};
```

could be used to simplify shift expressions. In the most extreme case, some xoc extensions use multi-line pattern constructors for code generation.

Other systems that have used syntax patterns, like ASF+SDF [60] or Weise and Crew's syntax macros for C [66], have required that the pattern slots be annotated with grammar symbol information. Where zeta uses a pattern like `‘C99.expr{\a + \b}`, those other systems would require a pattern more like `‘C99.expr{\a::expr + \b::expr}`. Instead, zeta tries all possible symbol types for both `\a` and `\b`. Only a few choices will produce successful parses. From those, zeta chooses the one that places the slots highest in the parse tree. For example, in `‘C99.expr{\a + \b}`, choosing to make `a` and `b` both `expr`s produces a successful parse, but so does making either or both of them `num`s (integer constants), because there is a grammar rule `expr: num`. If `a` is chosen to be a `num`, however, its slot is positioned one level deeper in the tree than when `a` is an `expr`. Thus zeta chooses to make `a` (and, for the same reason, `b`) an `expr`. The end result is that zeta chooses the simplest interpretation of the pattern, which is usually the one the extension writer has in mind. If zeta and the extension writer disagree, the rich type system for XST nodes usually catches the disagreement. If an extension writer expected `b` to be a `C.num` but zeta decided it was a `C.expr`, later attempts to use `b` as a `C.num` (for example, passing it to a function expecting a `C.num` or evaluating an attribute only defined for `C.num`s) would produce a type error when compiling the extension itself.

Disambiguation is needed mainly for slots in pattern matches. Slots in pattern constructors can use type information instead of inference: if `a` is statically typed as `C99.expr` (or even `C99.num`), the meaning of `\a` is unambiguous without any annotations.

The benefit of not needing annotations is slight in these examples, but in larger examples, especially pattern constructors, it significantly helps the readability of the patterns not to have the input syntax littered with annotations. The search required to infer annotations may seem expensive, but zeta implements it efficiently using a GLR parser, as discussed in Chapter 6.

As a smaller notational savings, a pattern match can omit the type following the `‘` if the variable being matched is statically known to have that type; for example, if `x` has static type `C99.expr`, `x ~ ‘C99.expr{\a + \b}` can be shortened to `x ~ ‘{\a + \b}`.

3.4. Canonicalizers

Most programming languages provide more than one way to write certain inputs. Canonicalizers are functions that transform an XST into a canonical form, to make pattern matching work on equivalent but differently-written expressions. To illustrate canonicalizers, this section considers the simple example of removing parentheses from infix expressions: excluding slightly different handling by the parser, `i+j+k` and `(i+j)+k` have identical meanings, but an XST for `(i+j)+k` would not match the syntax pattern `‘C99.expr{\a+\b+\c}`. To address this shortcoming in syntax patterns, zeta introduces canonicalizers, functions that convert an XST into its "canonical" form. Zeta applies canonicalizers to the result of any pattern constructor as well as to syntax patterns used in pattern matches.

For example, a compiler could implement a canonicalizer that converted parenthesized expressions into unparenthesized ones. With that canonicalizer in place, the XST for `(i+j)+k` would be identical to the XST for `i+j+k`, and both would match `‘C99.expr{\a+\b+\c}`. Similarly, the pattern used for matching `‘C99.expr{(\a+\b)+\c}` would be identical to the pattern used for matching `‘C99.expr{\a+\b+\c}`, so both patterns would match the same expressions.

Eliminating parentheses from the input is perhaps an extravagance—most expression patterns are not complicated enough to benefit from it—but canonicalization is helpful in the treatment of C types, making it possible for the (canonicalized) XST for the type `int (*)(void)` (a pointer to a function returning an int) to match the (canonicalized) pattern `‘C99.type{\t*}` (pointer to `t`). Chapter 4 explains this use for canonicalizers in detail.

3.5. Generic traversal

In addition to syntax patterns, zeta provides a generic mechanism for traversing and creating XSTs, in the form of two functions `xstsplit` and `xstjoin`. `Xstsplit(x)` returns an array containing the XST nodes that are children of `x`, and `xstjoin(x, arr)` creates a new XST using the same rule that `x` does, but using the nodes listed in `arr` instead of `x`'s own children. `Xstsplit` and `xstjoin` are useful mainly in the core of an extension-oriented compiler, for example in the base case implementation for attributes. While writing extensions, we use syntax patterns almost exclusively.

3.6. Attributes

Grammars and syntax patterns provide the ability to create, traverse, and transform XSTs, but to be useful, extensions need to be able to hook into the compilation process. They need a way to reuse and extend individual analyses (for example, type-checking of expressions) and compiler passes. Exposing the compiler internals to extensions is a key part of keeping extensions simple—no duplication of effort—but it must be done in a way that allows and encourages multiple, independently-written extensions to cooperate.

Zeta addresses these needs with attributes. An attribute is a function taking a single XST node as its argument, referenced using dot notation. The first time an attribute is referenced, zeta calls the function to compute the attribute value, which it then caches for future use. For example, xoc defines an attribute `type` on `C.expr` XST nodes. If `e` is a `C.expr`, then the first reference to `e.type` invokes the `type` attribute's function on `e` and caches the result. Future references to `e.type` return the cached result.

Extensions can write zeta functions to change the behavior of existing attributes. For example, an extension adding a bitwise left rotate operator can type-check the new expressions using:

```
extend attribute
type(e: C.expr): C.type
{
    if(e ~ ‘{\a <<< \b}){
        if(a.type.isinteger && b.type.isinteger)
            return a.type;
        error(e, "non-integer rotate");
        return nil;
    }
    return default(e);
}
```

The `extend attribute` statement replaces the function invoked to compute the given attribute with a new implementation. Inside that new function, the special name `default` refers to the old implementation. Here, the new implementation checks that the arguments to rotate are both integers; the type of the expression is the type of the operand being rotated. If passed an expression that is not a rotate, the new implementation invokes the old implementation by calling `default`.

Functions computing attributes can refer to attributes on other XST nodes or to other attributes on the same XST node. The fact that attributes are computed on demand eliminates the need to break an extension-oriented compiler into explicit passes. For example, the implementation of `type` given above refers to the `type` attribute on `e`'s children `a` and `b` during the computation of `e.type`. A traditional compiler would need to arrange explicitly for the child `type`s to have been computed before computing the `type` of `e`. As another example, evaluating `e.type` on an expression that is a variable name requires looking up that name in the set of active variables at that point in the program. That set is also provided by an attribute, called `namescope`, so the computation of `e.type` refers to (and likely invokes the computation for) `e.namescope`. A traditional compiler would need to arrange explicitly for `namescope`s to have been already computed when computing `type`s, perhaps by putting the computation of `namescope`s in an earlier pass than the computation of `type`s. The lazy evaluation of zeta's attributes eliminates the need to declare intra-attribute or inter-attribute dependencies explicitly. This helps keep extensions simple.

The price of not explicitly declaring attribute dependencies is that it is possible to create dependency cycles, in which one attribute depends, perhaps indirectly, on itself. This was a common problem for us as we developed the xoc core, but we have never run into this while writing extensions. Zeta diagnoses cycles at run-time and provides a stack trace showing the code involved in the cycle.

3.7. Extensible functions

Attributes are functions of a single variable—the XST node—and while many useful computations and analyses can be phrased as attributes, some cannot. For example, a C compiler must be able to decide, given two types, whether it is permissible to implicitly convert a value of the first type into the second. Implementing this functionality using attributes would be awkward, requiring a curried function as the attribute value. To accommodate cases like this one, zeta provides extensible functions, which are like ordinary zeta functions but they can be extended using almost the same syntax as attributes.

Xoc implements the type conversion question as an extensible function named `canconvert`. For example, `canconvert(‘C.type{int}, ‘C.type{long})` is true; `canconvert(‘C.type{int*}, ‘C.type{long*})` is not. In ANSI C, a pointer to a signed char cannot be implicitly converted into a pointer to an unsigned char. Some compilers are less strict, allowing this without warning. An extension can implement this behavior by extending `canconvert`:

```
extend fn
canconvert(old: C.type, new: C.type): bool
{
     if(old ~ ‘{signed char*} && new ~ ‘{unsigned char*}
     || old ~ ‘{unsigned char*} && new ~ ‘{signed char*})
          return true;

     return default(term);
}
```

Extensible functions are essentially a generalization of attributes, except that results are not cached. The caching behavior is the main reason to prefer attributes over extensible functions, to eliminate repeated computation. (In the worst case, removing caching from attributes might produce an exponential increase in the amount of computation done.)

Chaining attribute and extensible function implementations explicitly via `default` means that it is possible to construct a pair of extensions that behave differently when loaded in different orders: if a second extension is loaded after a first, the second might undo some of the effects of the first. Such order dependence runs counter to the overall goal of composability, but sometimes it is useful for one extension to refine the changes made by another. Chapter 5’s Sparse extension takes advantage of this capability.

3.8. Summary

Extension-oriented syntax trees are the heart of an extension-oriented compiler. It is important that the XST interfaces are powerful enough to write extensions but at the same time limited enough that independently-written extensions can work together. Zeta’s interfaces achieve this balance by exposing functionality but at the same time hiding low-level details; hiding the low-level details makes it easier to write extensions and, more importantly, makes it possible for multiple extensions to work together.

Grammars expose a method for defining the input program syntax but hide the construction of XSTs and the details of parsing. This enables composition of extensions: to compose extensions, simply take the union of all the rules defined by the base compiler and extensions.

Syntax patterns expose a method for manipulating XSTs but hide their low-level representation. This enables extensions to manipulate and pass along (but not explicitly recognize) syntax created by unknown extensions. Canonicalizers provide a mechanism for rewriting input programs to make syntax patterns more useful; Chapter 4 will use them to make syntax patterns semantically meaningful for C types.

Attributes expose and allow the modification of core compiler analyses and provide a framework for interacting with other extensions without exposing details like pass scheduling and which other extensions are loaded. This enables extensions to work together to process input programs that mix features introduced by multiple extensions.

These interfaces promote composability of extensions, but they do not guarantee it. Extensions can introduce conflicting grammar rules or conflicting attribute redefinitions or even introduce cyclic attribute dependencies. The following chapters describe xoc and extensions to it. In the context of that experience, Chapter 7 returns to the question of extension composability.

Compiling C with XSTs

Using zeta and its extension-oriented syntax trees (described in Chapter 3), we have designed and implemented a prototype extension-oriented C compiler called xoc. Xoc accepts a command-line argument directing it to load extension modules, like

```
xoc -x alef -x sparse file.c
```

It loads the named extensions (`alef` and `sparse` in this example) and then reads the input `file.c`, a program written in C plus any changes introduced by the extension modules. Xoc type checks the input program and then compiles it from the potentially-extended C to a lower-level standard (unextended) C representation that can be handled by a traditional C back end. In addition to implementing xoc itself, we have implemented a variety of extension modules. This chapter describes the implementation of the xoc core and the interface it presents to extensions; the next chapter describes three significant extensions.

4.1. Overview

Xoc starts by parsing its command line and loading any specified extension modules. Then it must read the input source. The C language is complicated by a text-based preprocessing phase, often implemented as an external program (the so-called C preprocessor). Xoc invokes a standard C preprocessor on the input files named as the command line and uses that preprocessor's output as its input. Thus, the preprocessing phase is not extensible.

Next, xoc parses the input using its C99 grammar to produce an XST. Zeta's explicit accommodation of ambiguity allows xoc to delay resolving the C type versus name ambiguities. After parsing, xoc invokes a separate disambiguation phase to resolve the C type versus name ambiguity and produce an unambiguous tree. At this point, xoc canonicalizes the XST and begins the real work of compilation.

Xoc type checks its input and translates any features introduced by extensions into standard C. These two phases are both implemented as attributes: the attribute machinery schedules the computation and provides a framework that allows the xoc core and any loaded extensions to cooperate to type check and compile the program.

Our work focuses on the extensibility of compiler front ends, so once the input program has been type checked and translated into standard C, xoc invokes a standard C compiler (specifically, the GNU C compiler gcc) on this standard C representation to produce an object file.

4.2. Resolving the C99 type versus name ambiguity

The main challenge in parsing C is handling the type versus name ambiguity introduced by C's `typedef` construct. For example, the statement `x * y;` can be parsed two different ways: as a multiplication of two variables `x` and `y`; or as a declaration of a new variable `y` with type `x*`. Which choice is correct depends on whether `x` names a type or a variable. Typically, C compilers avoid this ambiguity during parsing by performing some semantic analysis during the parse. When the lexer reads a name like `x`, it uses the computed semantic information to decide whether `x` is a name or a type and return the appropriate token to the parser. This approach tightly couples the lexer and the semantic analysis of the program, making it difficult for the compilation to proceed as independent phases.

Because zeta's parser can handle ambiguity during parsing, xoc can avoid the tight coupling of traditional C compilers. Xoc uses an intentionally ambiguous grammar, allowing any name to be used as either a variable name or a type name. When parsing a statement like `x * y;`, zeta returns an XST that encodes both possible parses. Xoc then invokes a disambiguator to choose the correct one. Even if neither is correct, detecting the error in a semantic phase instead of during parsing allows xoc to give a more specific error message.

The disambiguator traverses the XST in a single top-down pass, using `xstsplit` and `xstjoin` to traverse non-standard syntax introduced by extensions. When it sees a `C.decl`, it records that all the `C.decornames` introduced by that declaration are now names of variables (or names of types, if the declaration includes the word `typedef`). To handle a node with multiple parse choices, the disambiguator explores all possibilities, keeping the choice with the fewest number of errors (variable names used as type names or vice versa). If both choices have no errors, the ambiguity was not created by the C type versus name problem, so it must have been introduced by an extension. In this case, xoc reports an error and exits. Assuming no extension-introduced ambiguities, the disambiguator returns an XST with no parse ambiguities.

Disambiguating the XST as a phase separate from parsing makes it possible to treat a mistake like using a type name where a variable name was expected as a semantic error, not a syntactic one. For example, the C program

```
typedef int A;
A a;
return A+1;
```

is not ambiguous: the `return` statement can only be parsed if `A` is taken to be variable name, although `A` is a type name. A traditional C compiler would report a syntax error here. Xoc can give the more helpful error that `A` is being used as a variable name but is in fact a type name.

Making the disambiguator extensible would only be useful when introducing new ambiguities. For example, if the disambiguator were extensible, an extension could introduce nicknames as another kind of identifier and then could extend the disambiguator to resolve the resulting type versus name versus nickname ambiguities. Creating additional ambiguities strikes us as a bad idea, so we have not made the disambiguator extensible.

4.3. Canonicalization of types

After disambiguating the input, xoc invokes any canonicalizers to transform the tree into canonical form. The xoc core defines two canonicalizers, one for expressions and one for types; extensions can introduce their own canonicalizers for new syntax they introduce. Xoc's expression canonicalizer is simple: it removes parentheses as suggested in Chapter 3. The type canonicalizer is more involved and more important. In C, the syntax of types does not reflect their semantic structure. For example, `int*` denotes a pointer to an integer, and in general one might expect a pattern match like `‘C.type{\t*}` to match a pointer to any type `t`. Without type canonicalizers, that pattern would not match all pointers. For example, a pointer to a function returning `int` is written `int(*)(void)`, which does not have the same syntactic structure as the pattern `‘C.type{\t*}`. The type canonicalizer transforms types into a form that allows pattern matching to follow type semantics, not type syntax.

Understanding the type canonicalizer requires a careful understanding of C declarations. In C, a declaration is split in two pieces, a type specifier and a type declarator. In a declaration like `int *p`, the type specifier is `int` and the declarator is `*p`. The idea behind declarators is that declaration reflects use. Declaring `int *p` creates a new variable `p` such that `*p` is an `int`; therefore `p` must be a pointer to an `int`. Declaring `int (*f)(void)` creates a new variable `f` such that `(*f)()` is an `int`; therefore `f` must be a pointer to a function returning an `int`. C constructs that take types as arguments, like casts and one form of the `sizeof` operator, use as their syntax a declaration in which the name is dropped from the declarator to produce a so-called abstract declarator, such as the `int*` and `int(*)(void)` used above. C's type syntax is simultaneously elegant and terribly confusing, especially for new programmers. The C Infrequently Asked Questions list, which parodies answers to common questions about C, explains that "in C, declaration reflects use, but it's one of those weird distorted mirrors" [54].

Xoc's type canonicalizer rewrites the syntax of types so that it follows the logical semantics of the type. It adopts the `typeof` type specifier from GNU C, which allows using `typeof(`*type*`)` as a type specifier. Using the `typeof` syntax, the canonicalizer can rewrite any type so that only one abstract declarator can be applied at a time; it also ensures that abstract declarators are always applied to `typeof` types. For example, the canonicalizer transforms `int*` into `typeof(int)*` and `int(*)(void)` into `typeof(typeof(int)(void))*`. The canonicalized syntax makes it clear that both are pointers: both now match the syntax pattern `C.type{\t*}`. Figure 1 illustrates the transformation. Introducing `typeof` creates a kind of parentheses for types, so the canonicalizer must also take care to remove unnecessary parentheses. For example, it simplifies `typeof(typeof(int)*)` and `typeof(typeof(int))*` to `typeof(int)*`.

Xoc's type canonicalizer also handles storage specifiers, type qualifiers, and GNU C attributes. A storage specifier is a word in a declaration that is not part of the type but rather specifies where to store a given declaration. For example, `extern int *x` declares that `x` is an external variable allocated elsewhere. `Typedef` is also a storage specifier: it declares that `x` has no storage at all but instead is a type name. Logically, storage specifiers are separate from types, but syntactically, they are intermixed. For

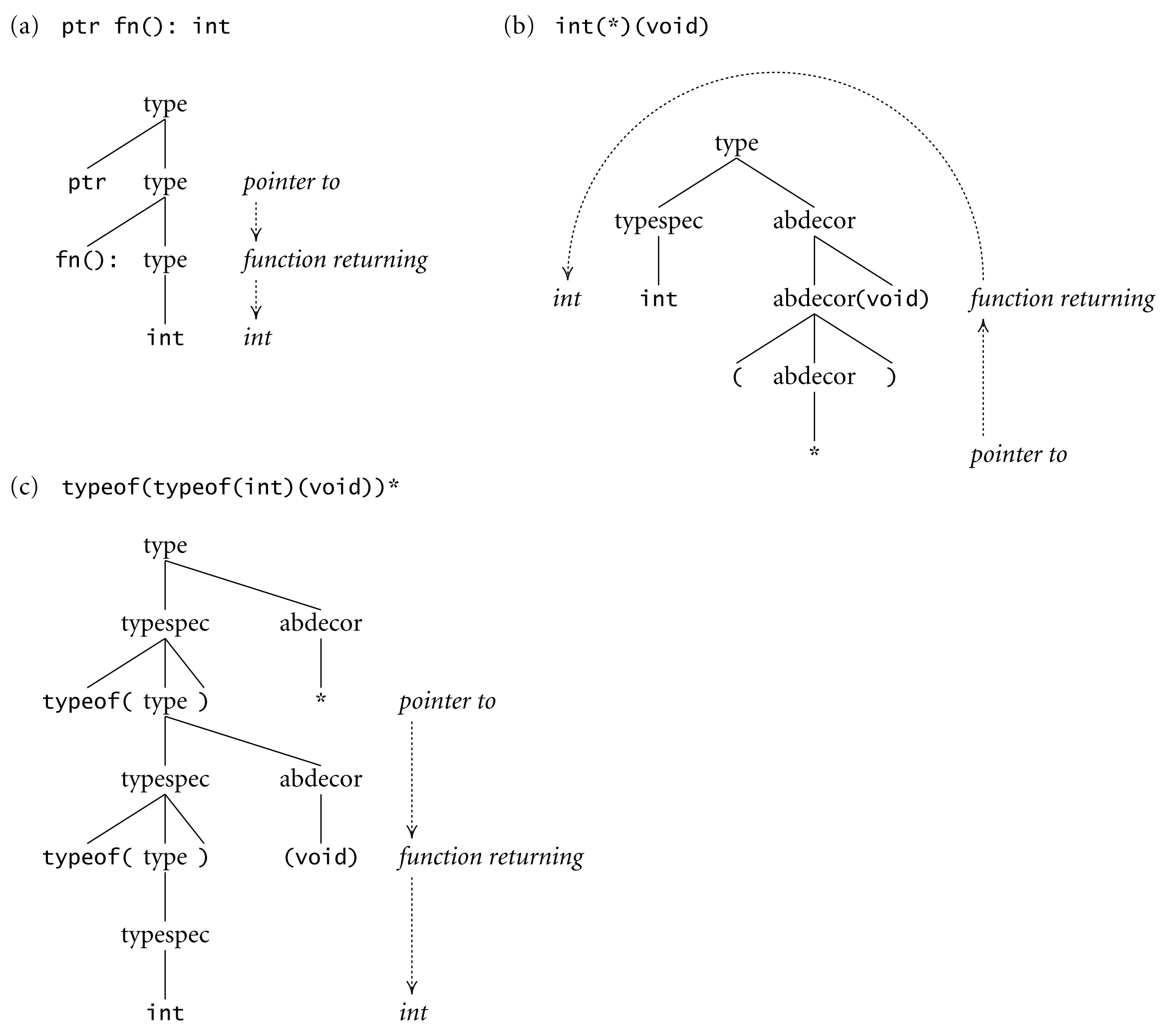

Figure 1. The C type problem and its solution. The natural top-down semantic structure of a type like "pointer to function returning int" would put the "pointer" node at the top of the tree, as in the zeta-like type syntax of (a). C's type declarator syntax create a bottom-up structure as in (b). The problem is that for a syntax pattern like `‘C.type{\t*}` to match any pointer type, the type syntax needs to have structure like (a), not like (b). Xoc's type canonicalizer rewrites types with nested abstract declarators like (b) into types with only single abstract declarators, adopting the GNU C `typeof` syntax as a kind of parentheses for types. Xoc's translation of (b) is shown in (c). The italic text connected by dashed arrows shows how to read each type.

example, most C programmers would write `extern int *x`, but it is equally valid, if uncommon, to write `int extern *x`. A type qualifier is a word like `const` or `volatile` in a type like `const char*` (pointer to const char) or `char* volatile` (volatile pointer to char).

GNU C attributes, which have nothing to do with zeta's concept of attribute, are a generalization of storage specifiers and type qualifiers. Instead of defining new reserved words like `extern` and `const`, GNU C defines a special syntax `__attribute__((`*list*`))` that can be used anywhere a type qualifier can. The advantages of this bizarre syntax are that new attributes can be created without reserving new words and that preprocessor macros can be used to make the __attribute__ clauses disappear from the view of non-GNU C compilers. For example, in GNU C, annotating a function with __attribute__((noreturn)) tells the compiler that the function does not return. This use does not preclude the use of noreturn in other contexts as a variable name.

Xoc's type canonicalizer removes storage specifiers, type qualifiers, and GNU C attributes from the explicit XST syntax and instead attaches them to the tree as pre-initialized attributes. For example, the expression `C.type{const int __attribute__((red))} canonicalizes to a `C.type{int} with its attrs attribute set to list [`C.attr{const}, `C.attr{red}]. (Any ordinary type qualifier is also a GNU C attribute; for simplicity, xoc chooses the GNU C attribute form as the canonical representation.) Similarly, a declaration like `C.decl{extern int x;} canonicalizes to a `C.decl{int x;} with its storage attribute set to `C.storage{extern}.

Zeta's semantics are that attributes are not automatically copied when nodes are, because an attribute value may depend on the node's placement in the parse tree; to make copies inherit the attrs and storage, the default definitions consult copiedfrom:

```
attribute attrs(t: C.type): list C.attr
{
	if(t.copiedfrom)
		return t.copiedfrom.attrs;
	return nil;
}
```

(The attribute will be non-nil only if it has been pre-initialized, as the canonicalizer does.)

Canonicalizers perform syntactic transformations. The type canonicalizer is elaborate but purely syntactic: it does not require examining the context in which a type appears, which would fall under type checking's jurisdiction. For example, type canonicalization does not expand typedefs. Similarly, expression canonicalization cannot perform constant evaluation, because it cannot interpolate named constants from enum declarations.

4.4. Type checking

Once the input XST has been disambiguated and canonicalized, xoc can type check it. Xoc implements type checking using three attributes:

```
attribute typecheck(C);
attribute type(C.expr): C.type;
attribute type(C.type): C.type;
```

The typecheck attribute applies to multiple kinds of XST nodes, like top-level declarations, function definitions, statements, and statement blocks. If x is an XST node, the first reference to x.typecheck type checks x. The attribute has no value; it is evaluated only for the side effect of printing type errors.

The type attribute on C.expr XSTs evaluates to the type of the expression, like `C.type{int} for `C.expr{1+2} or for `C.expr{x} when x is an integer variable or an integer constant. Type-checking a name like x requires name resolution, discussed

```
int x;
int y = x;
```

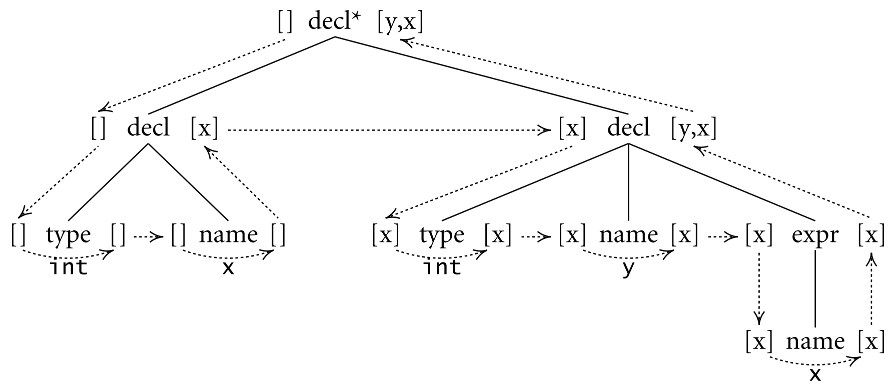

Figure 2. An illustration of the computation of the `entrynamescope` and `namescope` attributes on the simplified parse tree for a pair of declarations. The bracketed list on the left of each node is the scope on entry to that node; the list on the right is the scope on exit. Dashed arrows denote data flow between attributes: an arrow from one list to a second means that the first list is used to compute the second. The attribute grammar literature refers to such attributes as "threaded attributes," because of the way they the data flow is threaded through the entire tree [65]. Declarations augment the default threading behavior by adding new names to their exit scopes.

in the next section.

The `type` attribute on `C.type` XSTs evaluates to the type denoted by that XST. The need for this attribute surprised us, since compiler texts rarely talk explicitly about type-checking types. The attribute is necessary to handle expansion of type names introduced by `typedef`, so that, for example, `‘C.type{x}` can be replaced by `‘C.type{int}`. The type attribute also resolves a type like `‘C.type{struct foo}` to its current definition. (Like type names, struct tags can refer to different structures at different points in the program.) Both of these expansions also require name resolution.

4.5. Scopes for name resolution

When xoc encounters a name like `x` during type checking, xoc must resolve the name, to find the declaration that `x` refers to. Traditional imperative compilers implement name resolution using a mutable symbol table in a compiler-directed recursive walk of the input tree. During the walks, the compiler adds and removes symbols from the table. The set of names available at a given point in the program is the set of names in the table when the traversal reaches that point. That set is not saved, though: once the traversal has moved on, the set of possible names is no longer available [1][24].

Xoc needs a solution that fits into the attribute-directed input processing, which may look up names in any order. The set of all symbols that can be referred to at a particular XST node is called the *scope* at that node. Xoc implements two attributes `entrynamescope` and `namescope` that evaluate to the scope on entry to and exit from a particular XST node. In the absence of declarations, the scope on entry to an XST node `x` is either the scope on exit from `x`'s left sibling or, if `x` has no left sibling, the scope on entry to `x`'s parent. Similarly, the scope on exit from an XST node `x` is either the scope on exit from `x`'s rightmost child or, if `x` has no children, simply `x`'s `en-`

trynamescope. These rules correspond to a pre+post-order traversal of the input tree that visits each node twice, once before traversing the children and once after traversing the children. The first visit sets the entrynamescope, and the second sets the exit namescope. Declarations add to the exit namescope before passing it along to the next visit in the traversal. Figure 2 illustrates the name scopes for a pair of declarations.

Xoc would like to ensure that, by default, reusing an expression in a new XST does not change its meaning. Specifically, names in the copy should refer to the same symbols as the names in the original. To implement this, the namesym attribute, which looks up the symbol corresponding to a name, looks for the name in the original scope when invoked on a copied node:

```
attribute namesym(n: C.expr): Sym
{
    if(n.copiedfrom)
        return n.copiedfrom.namesym;
    ...
}
```

(The type Sym is the xoc data structure representing symbols.)

This discussion of scopes has assumed that, like in C, names only refer to declarations that occur earlier in the program. C++, Java, and other languages allow references (in some cases) to names that occur later in the program. Bergan's thesis [7] demonstrates that this kind of attribute framework is general enough to accommodate such languages.

4.6. Compilation

Once the input is type checked, xoc can compile it. Xoc only has to translate the input program (written in C plus any loaded extensions) to standard C, and then it invokes a standard C compiler to translate the program to machine code. Xoc refers to the translation from C plus extensions to standard C as compilation.

The bulk of the compilation work is implemented as a collection of compiled attributes defined on statements, expressions, declarations, types, and most other C grammar symbols. The attributes are defined on XSTs of type C (aka C99+?), but they return XSTs of type COutput, defined as C99+StmtExpr (for convenience when compiling expressions, the output is allowed to contain the GNU C "statement expression" construct):

```
attribute compiled(C.stmt): COutput.stmt;
attribute compiled(C.expr): COutput.expr;
attribute compiled(C.decl): list COutput.stmt;
attribute compiled(C.type): COutput.type;
attribute compiled(C.block): COutput.block;
...
```

The compiled form of a statement is a statement, and the compiled form of an expression is an expression. The compiled form of a declaration is a sequence of initialization statements. Xoc's C printer prints declarations for variables as needed, which makes it easy to use new temporary variables in the compiled form. Thus the only explicit semantics needed from a declaration is the potential initialization of the declared variables. Translating declarations into statement sequences also allows new kinds of initializers to be introduced and translated.

The fact that types need to be compiled was a surprise, but in retrospect it makes sense: if an extension introduces a new type, that type has to be translated into a standard C type when compiling to object code.

The `compiled` attribute for blocks and other symbol types is not particularly interesting: the default implementation just compiles the children and then constructs a new XST node with those children.

4.7. Data flow analysis

By default, xoc does not perform any sophisticated analyses on input programs, but extensions may need to. To accommodate them, xoc implements a simple data flow library that operates on `COutput` (non-extensible) XSTs. Extensions that wish to invoke data flow checks do so by extending the implementation of the `typecheck` attribute on function definitions. They compile the body of the function and then apply the analysis to the standard C form. Running the data flow analysis on the compiled form rather than the original form makes it possible to analyze code that uses extensions.

Data flow library clients start by creating a control flow graph from the compiled form; they can pass in a filter function to remove uninteresting nodes from the graph. Once the control flow graph has been constructed, clients can invoke a full data flow analysis using their own transfer and join functions [27]. The node and edge values used in these functions are of an arbitrary, client-specified type. The library lets the clients focus on the details of a specific analysis, not the general data flow machinery.

4.8. Extension interface

Xoc's extension interface is the C99 grammar and collection of attributes that it defines. This chapter has already discussed the three most important attributes: `typecheck`, `type`, and `compiled`. An xoc extension is a zeta module that is loaded by the xoc core in response to the command-line option `-x`. Extensions typically define a couple of new grammar rules and then appropriate type-checking and compilation to handle those rules. For example, Figure 3 shows a small extension that introduces a new bitwise rotate operator *a* <<< *b* similar to the C shift operators. The extension adds a single new grammar rule for `expr`, extends the `type` attribute on `C.exprs` to handle the new expression, and then extends the `compiled` attribute to compile the new expression down to standard C. The next chapter examines more substantial extensions.

4.9. Summary

The C programming language presents a number of thorny challenges to the extension-oriented compiler designer. The type versus name ambiguity threatens to couple parsing tightly to semantic analysis. Type declarations threaten to require an alternate type representation. This chapter has described how XSTs gracefully handle both of these problems.

The demand-driven execution of XST attributes does not fit well with the usual

```
from xoc import *;

extend grammar C99
{
    expr: expr "<<<" expr [Shift]
}

extend attribute
type(term: C.expr): C.type
{
    switch(term){
    case ‘{\a <<< \b}:
        if(a.type.isinteger
        && b.type.isinteger)
            return a.type;
        errorast(term,
            "non-integer rotate");
        return ‘C.type{int};
    }
    return default(term);
}
```

```
extend attribute
compiled(term: C.expr): COutput.expr
{
    if(term ~ ‘{\a <<< \b}){
        n := a.type.sizeof * 8;
        term = ‘C.expr{
            ({
                \(a.type.unsigned) x = \a;
                \(b.type) y = \b;
                (x << y) | (x >> (\n-y));
            })
        };
        return term.compiled;
    }
    return default(term);
}
```

Figure 3. A simple xoc extension implementing a bitwise rotate operator *a* <<< *b*. The `extend grammar` clause introduces the new grammar rule for the operator. The `extend attribute type` definition implements the type-checking rule for the expression, passing all other expressions to the original attribute definition via the call `default(term)`. The `extend attribute compiled` definition implements the expression's behavior by rewriting it using only standard C and a GNU statement expression (the `({ ... })` construct) and then invoking the `compiled` attribute on the rewritten version. (Like the extension of `attribute type`, it passes all other expressions to the original attribute definition via the call `default(term)`.)

imperative name resolution pass of traditional compilers. This chapter has also described how to implement name resolution cleanly using attributes. In the case of name resolution, the attribute-based solution is not obviously cleaner than the traditional solution, but the important part is that it can be accommodated, making the attribute framework the only interface extensions need to understand. Put another way, the xoc core's implementation of name resolution may be slightly more complicated using XST attributes than it would be otherwise; this is an acceptable tradeoff if, in return, extensions will be simpler to write and more composable. The next chapter examines the extensions written using xoc.

Extensions

Chapter 4 described the xoc compiler core. We envision two main ways that extensions might choose to extend xoc. First, an extension might add custom support for a particular runtime library. For example, an extension could create a yacc-like interface [31] to a parsing library with a plain C function call interface. Second, an extension might introduce additional checks to an error-prone interface. For example, the C library `qsort` function takes two arguments: the data to be sorted and a comparison function that accepts two `void*` arguments to compare two elements. It is important that the data being sorted matches the type expected by the comparison function, but the implicit conversion to `void*` keeps the C compiler from checking this condition. A `qsort`-specific extension could check the conversions to make sure that the comparison function matches the data being sorted. There are also extensions that xoc cannot support. Since xoc is only a front end, extensions that require changes to code generation—for example, segmented stacks, garbage collection, exceptions, or use of special hardware instructions—cannot be implemented.

This chapter describes two large collections of extensions built atop xoc. The first collection falls into the first category above. It implements the types and syntax supported by the Alef programming language by translating them into calls to the C library that replaced Alef. As mentioned in Chapter 1, Alef was a promising new language that was abandoned due to the cost of maintaining an entire compiler. The xoc implementation of Alef features does not need to implement an entire compiler—it reuses the xoc core's handling of standard C—making it much shorter than the original Alef implementation.

The second collection falls into the second category above. It implements the checking functionality of the Sparse program checker used by Linux kernel developers. Those checks involve new pointer attributes, function attributes, and data flow analyses.

Both of these examples are extension collections, not single extensions. The ease with which xoc combines multiple extensions encourages modularity, making it natural to split a large language change into a set of cooperating, small extensions. Figure 1 shows the line counts of the extensions we have written for xoc. In addition to the Alef and Sparse suites, we have also implemented two features introduced by GNU C.

This chapter presents only the essential code from each extension along with line counts. Appendix B gives the full source code for each extension.

5.1. Alef

The first extension collection recreates the types and syntax of Alef as an interface to libthread, the C library that replaced it. The main functionality provided by the library is the channel types, send and receive expressions, the alt statement, and thread creation. As an additional illustration, this section also considers one of Alef's lesser-

Name	Lines	Description
alef_alloc	39	Alef channel allocation
alef_alt	106	Alef alt statement
alef_chan	20	Alef channel type
alef_iter	103	Alef iterator expressions
alef_sendrecv	56	Alef channel send, receive operators
alef_task	98	Alef task (thread creation) statement
	422	total Alef
sparse_addrspace	68	Sparse address space attribute
sparse_context	245	Sparse flow-sensitive context checking
sparse_force	29	Sparse force attribute
sparse_grammar	16	Sparse attribute syntax
sparse_noderef	33	Sparse noderef attribute
	391	total Sparse
gnu_cond	26	GNU a ?: b binary conditional expression
gnu_typeof	44	GNU typeof(expr) type specifier
	883	total, all extensions

Figure 1. Line counts for xoc extensions. Most extensions are under a hundred lines of code. The largest extension, the context checking part of Sparse, is only 245 lines.

used features, the iterator expression.

5.1.1. Channel types

In Alef, threads communicate over typed, unidirectional communication ports called channels, following Pike's Newsqueak [47], which in turn was inspired by Hoare's CSP [29]. Messages on channels are typed, first-class objects: a `chan(int)` carries only `int`s, while a `chan(chan(int))` carries only channels that carry `int`s. In libthread, channels are intended to be used for only a single message type—in particular, the size of the message is fixed at channel creation—but the type system does not enforce this: all channels have type `Channel*`. The first Alef extension introduces typed channels, which compile into libthread's untyped channels. The extension is twelve lines, four to introduce the new type, and seven to compile that type down to the libthread `Channel*`:

```
 1    extend grammar C99
 2    {
 3        typespec: "chan" "(" type ")";
 4    }
 5
 6    extend attribute
 7    compiled(t: C.type): COutput.type
 8    {
 9        if(t ~ ‘{chan(\_)})
10            return ‘COutput.type{Channel*};
11        return default(t);
12    }
```

The `if` statement (line 9) translates a channel type into the C type `Channel*` as the compiled form.

```
int
add(chan(int) c1, chan(int) c2)
{
	return <-c1 + <-c2;
}
```

```
int
add(Channel *c1, Channel *c2)
{
	int tmp1, tmp2;

	return (chanrecv(c, &tmp1), tmp1) +
		(chanrecv(c, &tmp2), tmp2);
}
```

Figure 2. A short Alef program (left) and its translation into standard C using the libthread library (right). The alef_channel and alef_sendrecv extensions implement this translation.

5.1.2. Channel operations

Alef provides special syntax for send and receive operations on channels. The operator `<-` receives a message from a channel, as in `x = <-c`, and the operator `<-=` sends a message to a channel, as in `c <-= x`. These operators are typed, so that, for example, it is a compile-time error to send a pointer on a channel expecting an integer.

The code to introduce the syntax is the following five lines:

```
extend grammar C99
{
	expr: "<-" expr
	    | expr "<-=" expr;
}
```

The code to type-check the operators is another twenty-seven. The fragment handling the receive operator is:

```
1		if(term ~ `{<-\e}){
2			if(e.type != nil){
3				if(e.type ~ `{chan(\t)})
4					return t;
5				errorast(term, "receive from non-channel");
6			}
7			return nil;
8		}
```

It recognizes the receive operator (line 1), checks that the expression being received from is itself a channel (line 3), and returns the channel's message type as the type of the expression (line 4). The check of `e.type` against `nil` (line 2) is not strictly necessary, but avoids an extra error message if `e` fails to type check.

The fragment handling the send operator is similar, but must check that the message being sent is an acceptable type:

```
1		if(term ~ `{\e <-= \v}){
2			if(e.type && v.type){
3				if(e.type ~ `{chan(\t)}){
4					if(canconvert(v.type, t))
5						return t;
6					errorast(term, "cannot send "+v.type.fmt+
7						" on chan of "+t.fmt);
```

```
 8             }
 9             errorast(term, "send on non-channel");
10         }
11         return nil;
12     }
```

The code recognizes the send operator (line 1), checks that `e` and `v` have both type checked (line 2), and then checks that `e` is a channel (line 3) and that `v` can be converted to `e`'s message type (line 4).

The code to compile a receive operator is as follows:

```
1    if(term ~ ‘{<-\e} && e.type ~ ‘{chan(\t)}){
2        tmp := mktmp(t);
3        return ‘C.expr{(chanrecv(\e, &\tmp), \tmp)}.compiled;
4    }
```

The code looks for a receive expression and determines the channel message type (line 1) and allocates a temporary variable to hold the message (line 2). Then it creates a C code fragment that first calls libthread's `chanrecv` function to receive into the temporary variable and then evaluates to the value of the temporary variable (line 3). Finally, it returns the compiled form of that C expression. If the extension writer forgot the `.compiled` on line 3, the extension would not type check (as a valid zeta program): it would be returning a `C.expr` from a function expected to return a `C99.expr`. The type system thus ensures that the eventual compiled form uses no extensions. The compilation of the send operator is similar; it is given in Appendix B.

5.1.3. Alt statements

The final channel operation is the `alt` statement, which blocks waiting for any of its channel operations to become ready and then executes one. Libthread provides an `alt` function implementing the core operation; the extension only implements the syntax.

First, the extension must introduce the syntax:

```
extend grammar C99
{
    stmt: /alt/a "{" altclause* "}";
    altclause: "case" expr ":" stmt*;
}
```

Writing `/alt/a` instead of `"alt"` uses a regular expression pattern instead of a literal string. The `/a` suffix specifies that the pattern can be ambiguous, meaning if another regular expression also matches, the lexer should return both. This allows the use of `alt` to introduce the alt statement without precluding the use of `alt` as an ordinary identifier elsewhere in the program. In contrast, writing `alt` would make `alt` a reserved word like `while` or `switch`.

The grammar rules given above are somewhat lax: in fact, the case expressions for alt statements must be one of the three forms <-*c*, *e* = <-*c*, or *c* <-= *e*. Using just `expr` postpones the syntax check to type checking, which is more work but can give more precise error messages. Another benefit of using `expr` is that the expressions can be type-checked (though not compiled) as ordinary channel expressions.

```
alt {
case <-c1:
     print("receive c1\n");
case c2 <-= e:
     print("send e on c2\n");
}
```

```
Alt alts[3];

alts[0].op = CHANRCV;
alts[0].c = c1;
alts[0].v = NULL;

alts[1].op = CHANSND;
alts[1].c = c2;
alts[1].v = &e;

alts[2].op = CHANEND;

switch(alt(alts)){
case 0:
     print("receive c1\n");
     break;
case 1:
     print("send e on c2\n");
     break;
}
```

Figure 3. A short Alef alt statement (left) and its translation into standard C using the libthread library (right). The alef_alt extension implements this translation.

The alt extension does more code generation than the previous extensions. Figure 3 shows a simple alt statement and its translation into C. To generate the code for the alt statement, the alt extension iterates over the clauses, creating two lists of statements, one containing initializers and one containing switch body statements. Appendix B gives the full source code; here is the snippet that creates the new initializer and body for a send:

```
case ‘{\chan <-= \val}:
     assert chan.type ~ ‘{chan(\t)};
     tmp := mktmp(t);
     inits = ‘C.stmt{{
          \tmp = \val;
          \alts[\i].op = CHANSND;
          \alts[\i].v = \tmp;
          \alts[\i].c = \chan;
     }} :: inits;
     body = ‘C.stmt{{
          case \i:
               \b
               break;
     }} :: body;
```

The `inits` list is the initializers; the body list is the switch body. Once the clauses have been processed, constructing the final C code is simple:

```
int print(char *msg);

task print("hello\n");
```

```
int print(char *msg);

struct fnargs {
     int (*f)(char*);
     char *a0;
};

void
tramp1(void *v)
{
     struct fnargs cl;

     cl = *(struct fnargs*)v;
     free(v);
     cl.f(cl.a0);
}

struct fnargs *cl;
cl = malloc(sizeof *cl);
cl->f = print;
cl->a0 = "hello\n";
taskcreate(tramp1, cl);
```

Figure 4. An Alef task (thread creation) statement and its translation into C using the libthread library (right). The alef_task extension implements this translation.

```
        return ‘C.stmt{{
            \inits
            switch(alt(\alts))
                \body
        }}.compiled;
```

5.1.4. Thread creation

After channel syntax, the most missed feature of Alef was its convenient syntax for creating new threads. The statement

```
task f(x, y, z);
```

created a new Alef task (thread) executing the function call `f(x, y, z)`. Libthread, like most C thread libraries, instead provides the clunkier

```
void taskcreate(void (*f)(void), void *a)
```

which creates a new task running `f(a)`. Starting a task with a function that requires more than a single void pointer argument requires extra effort. The extension automates that effort, creating a new argument structure type to hold the values `f`, `x`, `y`, and `z` as well as a new trampoline function that, passed a structure containing those values, executes `f(x, y, z)`. Then it expands the `task` statement above into code that allocates and initializes a new argument structure and passes it and the trampoline function to `taskcreate`.

This extension is interesting because it creates a new C structure type to hold the function pointer and its arguments. The number of elements in the structure and their types depend on the exact expression used in the `task` statement. To create the new type, the extension loops over the arguments making a list of structure member declarations:

```
decls := list [ ‘C.sudecl{\(f.type)* f;} ];
for(a in args){
     name := "a"+string(i++);
     decls = ‘C.sudecl{\(a.type) \name;} :: decls;
}
```

In the structure, the function pointer is named `f` and the arguments are named `a0`, `a1`, and so on. Once the member declaration list is constructed, creating the type is easy:

```
t := ‘C.type{ struct { \decls } };
```

The creation of the trampoline function is similar, building up a list of statements and using them to create the function definition. Appendix B gives the full code.

5.1.5. Iterator expressions

The previous sections recreated the most popular features of Alef. An extension-oriented compiler should make writing extensions easy enough that even unpopular features can be implemented, as long as they are important to at least one programmer. This section describes one such feature, Alef's iterator expressions.

In Alef, an iterator expression is an expression *lo*`::`*hi*. It evaluates to every value from *lo* up to *hi*, causing the statement in which it appears to be executed multiple times. For example, the single statement

```
b->print("%.2x", digest[0::16]);
```

prints a 16-byte array by printing each byte as a two-digit hexadecimal number. The iterator expression makes that single statement equivalent to the loop:

```
for(i=0; i<16; i++)
     b->print("%.2x", digest[i]);
```

It is not clear, in retrospect, what the compelling use cases for iterator expressions were. The Alef tutorial gives example code multiplying two 4x4 matrices, calling it “elegantly obscure” [18]:

```
typedef float Matrix[4][4];
void mul(Matrix r, Matrix a, Matrix b){
     int i, j, k;

     r[0::4][0::4] = 0;
     r[i=0::4][j=0::4] += a[i][k=0::4]*b[k][j];
}
```

Whatever the intended use, iterator expressions did not catch on in publicly released Alef code. In the 30,000 lines of Alef source code in the second edition release of Plan 9, there is exactly one instance of an iterator expression, the `b->print` line given above.

The fact that iterators did not catch on makes them a good example of the advantages of an extension-oriented compiler. Clearly someone thought iterators were

```
#define kernel __attribute__((address_space(0))
#define user   __attribute__((address_space(1), noderef))

void copy_from_user(void kernel*, void user*);

char user *u;
char kernel *k;

copy_from_user(k, u);
copy_from_user(k, k);  // type error!
copy_from_user(u, k);  // type error!
copy_from_user(u, u);  // type error!

x = *k;
x = *u;    // dereference of no-deref!
x = u[1];  // dereference of no-deref!
```

Figure 5. A program demonstrating the additional type checking introduced by Sparse. The listing defines `kernel` as a synonym for address space 0 and `user` as a synonym for address space 1 with no dereferencing allowed. Sparse checks that these tags are enforced, so that all but the first `copy_from_user` call would be reported as incorrect, as would the attempts to dereference the user pointer `u`.

---

worth the trouble to write a hundred or so lines of code to implement them in the Alef compiler and about two pages of documentation in the Alef manuals. Yet today they are just a forgotten, unused feature that is easier to leave alone than to remove.

The Alef iterator extension is 103 lines of code, about the same length as the iterator support in the Alef compiler. Instead of burdening a language core with iterator support, an extension-oriented compiler encourages the implementation of iterators as an extension, allowing users who need them to have them, but without burdening other users with needing to learn about them.

5.2. Sparse

The Alef extensions add new syntax to C. Now our focus shifts to adding new analyses and type restrictions to the compiler. The Sparse program checker, described in Chapter 1, provides two new kinds of analyses. First, it adds new pointer qualifiers and checks, to ensure that kernel pointers and user pointers are not accidentally misused inside an operating system kernel. Second, it adds a path-sensitive context-tracking mechanism, to catch instances where, for example, one path through a function acquires a lock but forgets to release it.

5.2.1. Pointer qualifiers

Sparse adds two new pointer qualifiers, phrased as GNU C attributes. The attribute `address_space(`*n*`)` marks a pointer as pointing at address space *n*, where *n* is an integer constant. If a pointer has no address space tag, it is assumed to be in address space 0. The Linux kernel uses three address spaces: 0 for kernel memory, 1 for user memory, and 2 for memory-mapped I/O. The second attribute, `noderef`, marks a pointer as one that cannot be dereferenced. If `p` is a `noderef` pointer, expressions like p[i] and *p are invalid, though address and size calculations like &p[i] and sizeof(*p) are still valid.

The first step for the extension is to add the attributes to the grammar. The base xoc grammar already accepts unadorned names (like noderef) attributes, but address_space takes an argument and thus needs to be added separately:

```
extend grammar C99 {
    attr: /address_space/a "(" expr ")";
}
```

The address space extension extends both canconvert and cancast to enforce the rule that it is not allowed to change the address space of a pointer, implicitly or explicitly. The extension-defined attribute address_space returns the address space number for a given type. With that attribute, it is easy to implement the rule:

```
1     extend fn
2     cancast(old: C.type, new: C.type): C.type
3     {
4         if(old ~ ‘{\told*} && new ~ ‘{\tnew*}){
5             if(told.address_space != tnew.address_space){
6                 sys.werrstr("address space mismatch");
7                 return nil;
8             }
9         }
10        return default(old, new);
11    }
```

The implementation looks for the case of casting one pointer to another (line 4) and checks that the two address spaces match (line 5). If they don't, it returns nil to signal an error, but first writes the system error string, which xoc will use in its error message to provide more detail (line 6). The implementation of canconvert is similar.

Sparse does provide a way around the checks: the force attribute to a cast will allow changing address spaces. In the xoc implementation, force is implemented in a separate extension. It implements a helper attribute without_force that removes the force attribute from a type. With that helper, it is easy to extend cancast once again, this time to allow casts between pointers when the cast specifies force:

```
1     extend fn
2     cancast(old: C.type, new: C.type): C.type
3     {
4         if(old ~ ‘{\told*}
5         && new ~ ‘{\tnew*}
6         && hasattr(tnew, ‘C.attr{force}))
7             return ‘C.type{\(tnew.without_force)*};
8         return default(old, new);
9     }
```

The extension looks for the case of casting from one pointer to another (lines 4-5). If the cast specifies the force attribute (line 6), then the cast succeeds, but the result type does not keep the force attribute (line 7). When force is present, this extension to cancast does not call the previous one (default). This overrides any restrictive checks that already existed, like the address space check above.

The combination of the address space and force extension illustrates that some-

```
void lock(void)    __attribute__((context(1)));
void unlock(void)  __attribute__((context(-1)));

void
mistake(int n)
{
     lock();
     if(n == 0)
          goto out;
     while(n-- > 0)
          work(n);
     unlock();
out:
     done();
}
```

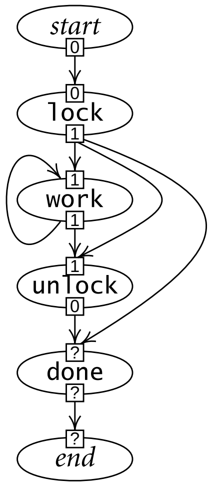

Figure 6. A program demonstrating the use of Sparse's `context` function attribute to detect locking inconsistencies. The `context` attribute declares the effect of the functions `lock` and `unlock` on the "context," an imaginary flow-sensitive variable that propagates through the function's control flow graph. The listing declares `lock` to add 1 to the context, while `unlock` subtracts 1. The graph shows the function `mistake`'s flow graph. The context propagates along control flow edges from one node's exit to another node's entry. The boxes on each node show the context at entry to and exit from that node. Most nodes do not change their context; the calls to `lock` and `unlock` do change the context, as directed by the annotations. The node calling `done` can be reached along two different paths with different contexts: arriving immediately from `lock` (when the `if` is true) brings a context of 1, while arriving from `unlock` brings a context of 0. Sparse diagnoses this context mismatch as a locking inconsistency: is the lock held or not at the call to `done`?

times, the order in which extensions are loaded matters. The two extensions of `cancast` above both handle some pointer conversions without calling `default` to consult extensions loaded earlier. In particular, a `force` cast that changes address spaces will match the `if` conditions in both. If the address space extension is loaded last, its implementation of `cancast` will be the first one called, and it will reject the cast. If the force extension is loaded last, it will get called first, and it will accept the cast. Thus, to mimic Sparse's semantics, it is important that the force extension be loaded after the address space extension. In our implementation, a wrapper extension called `sparse` loads the individual pieces in the correct order. Its full implementation is:

```
import sparse_grammar;
import sparse_addrspace;
import sparse_noderef;
import sparse_force;    // must load after addrspace and noderef
import sparse_context;
```

5.2.2. Context analysis

In addition to adding new syntax and creating new pointer types, xoc can be used to implement new analyses. Sparse implements a simple flow-sensitive type system that can be used to catch instances where functions are not paired properly. Sparse's type checker creates an imaginary variable called the "context" that propagates through the function. At entry to the function, it has value zero. Calls to functions marked with a `context(`*delta*`)` attribute add the integer constant *delta* to the context. For example, the Linux kernel uses contexts by declaring that acquiring a lock has context 1 and releasing a lock has context −1. If any point in a function can be reached along two different paths with different contexts, Sparse prints a warning, because it is undefined whether the lock is held or not at that point in the program. Figure 6 gives an example.

At 245 lines, the extension implementing the Sparse context checking functionality for xoc is the longest extension we have written, but even 245 lines is much shorter than a new compiler (like Sparse). The extension reuses the flow analysis provided by xoc. Specifically, the extension implements simple transfer and join functions (about ten lines each) that plug into the generic xoc core flow analysis library, runs the data flow analysis, and interprets the results. Appendix B lists the full extension source code.

5.3. Summary

The extensions described in this chapter extend xoc in many different directions: they add new data types, new syntax, new control flow, and new analyses. The variety of possible extensions demonstrate that xoc's interfaces can accommodate many kinds of language changes. The Alef and Sparse extension collections recreate existing tools, demonstrating that xoc can be used in real-world settings. Finally, the extensions are short—they range in size from tens to hundreds of lines—demonstrating that writing xoc extensions is significantly less work than writing a new compiler or program analysis tool from scratch. Further, xoc makes it convenient to build complex new behaviors, like Alef and Sparse, as a collection of cooperating, smaller extensions.

CHAPTER 6

Performance

The descriptions of zeta and xoc in Chapters 3 and 4 focused on functionality; this chapter focuses on implementing that functionality efficiently.

As a simple benchmark, consider running the Sparse extension to type check the Linux source file `init/do_mounts.c`, the first substantial file compiled during the Linux kernel build. Before preprocessing, the file is 449 lines (9,661 bytes); after, it is 15,520 lines (605,390 bytes), many of which contain long macro expansions. The GNU C compiler compiles this file in 0.4 seconds, even with expensive optimizations (`-O3`) enabled. The (not xoc-based) Sparse checks it in 0.1 seconds. Xoc with the Sparse extension described in Chapter 5 checks it in 4.7 seconds. By this rough comparison, then, xoc is about fifty times slower than a traditional compiler. Even so, making xoc and the Sparse extension run in as little as 4.7 seconds required effort. A few seconds to check a single file may not sound like much, but the Linux kernel tree has over 4,000 source files: Sparse takes minutes to check the entire tree, while xoc with the Sparse extension takes hours. This chapter describes the optimizations we have made to zeta so far.

6.1. Compiling zeta

The first implementation of xoc and zeta was an interpreter; it took most of a second just to start and load xoc, and then 16 seconds to check `do_mounts.c` after careful tuning. Profiling showed that at least 30% of the its time was spent in the byte code instruction decoding and dispatch loop.

To reduce the byte code interpreter overhead, we reimplemented zeta as a compiled language. The zeta described in this dissertation is based on a compiler that translates zeta source modules to C code. The compiler invokes a standard C compiler to generate a dynamic library that can be loaded into the zeta runtime. The compiled dynamic libraries are cached, with metadata about the compilation, alongside the original zeta files. If the zeta sources have not changed, subsequent uses will load the cached dynamic libraries without recompiling. Changing to a compiler reduced the time to start xoc from 0.6 seconds to 0.1 seconds and reduce the time to check `do_mounts.c` from 16 seconds to 4.7 seconds.

Figure 1 shows the amount of time spent in each phase for the two implementations. The comparison is not exact: the zeta language has evolved since we wrote the compiler, and the interpreter is not up to date. For example, we added XST canonicalization after creating the compiler. Thus, the interpreted version has no canonicalization pass; it runs an earlier version of xoc that used other mechanisms.

	interpreted zeta	compiled zeta	sparse	gcc	gcc -O3
start	0.6	0.1			
parse	2.3	0.8			
disambiguate	2.8	0.4			
canonicalize	-	0.3			
typecheck	10.3	3.1			
total	16.0	4.7	0.1	0.3	0.4

Figure 1. The performance of zeta, xoc, and the Sparse extension checking Linux 2.6.20's `init/do_mounts.c`, compared with the actual Sparse implementation, the GNU C compiler with no optimizations, and the GNU C compiler with optimizations enabled. Despite tuning, xoc is still fifty times slower than Sparse and twelve times slower than an optimizing compiler. The two zeta columns compare an early interpreted implementation of zeta against the current compiled implementation, showing the amount of time spent in each phase. All times are seconds on a 3 GHz Intel Pentium 4.

6.2. Parsing

After zeta loads xoc, xoc's first step is to parse the input. Zeta must provide a parser that can efficiently handle changes to the grammar: the exact set of grammar rules depends on which extensions are loaded. Zeta's parser must also handle ambiguity—multiple parses for a given input—by recording all possible parses and making them available. To provide this functionality efficiently, zeta uses a generalized LR (GLR) parser. Specifically, it uses a right-nulled GLR (RNGLR) parser [52]. Zeta originally used a more traditional GLR parser, following a technical report by McPeak [41], but the GLR parser McPeak describes does not handle empty productions correctly in all cases; the advantage of RNGLR is that it always handles empty productions correctly. Aside from the handling of empty productions, the differences between the two are slight. In the rest of this section, we use the term GLR to mean either GLR or RNGLR.

Because zeta's grammars may change when new extensions are loaded, zeta does not precompile the LR(1) parsing tables like tools like yacc [31]. Instead, it computes the tables and the underlying finite automaton on the fly, expanding the transition graph as needed.

We experimented with using LR(0) tables instead of LR(1), reasoning that the construction was simpler and the GLR parser would be able to handle the resulting ambiguities. That approach worked correctly and may have saved a small amount of time during table construction, but it took 100 times as long to parse the input due to the additional ambiguity. GLR parsers run fastest when operating in unambiguous sections of input: LR(0) tables have many more shift/reduce conflicts than LR(1) tables, causing temporarily ambiguous parses that the GLR parser executes more slowly. It is best when ambiguity is the exception rather than the rule.

Another important case involving ambiguity arose in the implementation of zeta. Zeta provides the shorthand `expr[,]+` as a grammar symbol for a comma-separated list of `expr` symbols. This symbol destructures into a zeta list containing `C.expr` XST nodes. Internally, zeta must generate grammar rules to implement `expr[,]+`. It can choose the left-recursive version:

```
expr[,]+: expr[,]+ "," expr
        | expr;
```

or the right-recursive version:

```
expr[,]+: expr "," expr[,]+
        | expr;
```

The left-recursive version is the one suggested when using parser generators, because it operates in a fixed amount of parser stack; the right-recursive version requires parse stack proportional in size to the number of elements in the sequence, but it fits more naturally with zeta's right-recursive representation of linked lists, as a head element followed by a tail list. Since our GLR implementation has no fixed stack size limit, we initially chose the right-recursive rules. Unfortunately, the right-recursive rules introduce significant ambiguity in some cases. For example, C allows a trailing comma after a list of expressions in an initializer. The following grammar rules capture the essence of the problem:

```
init: "{" expr[,]+ comma? "}"
comma: ","
```

These rules create an ambiguity: when the parser sees an `expr` followed by a comma, it doesn't know whether to expect the comma to be followed by another `expr` or by the closing brace. The right-recursive rules push the entire `expr` sequence onto the parse stack before applying any grammar rules; entertaining the possibility of collapsing the stack to a single `expr[,]+` at each comma incurs overhead linear in the number of expressions on the stack, making the overall parse time at least quadratic. For a typical Linux kernel source file, there is no runtime difference between the left-recursive and right-recursive rules, but the Linux kernel also contains machine-generated C files containing byte arrays representing binary data such as firmware images. The file that exposed the difference contains a 4,696-element array initializer. Parsing that file, including 230 kilobytes of included header text, takes under half a second using the left-recursive rules. Using the right-recursive rules, it would take over an hour.

6.3. XST copying

In order to ensure that every XST node has a well-defined parent, the description in Chapter 3 specifies that zeta must make copies of subtrees when reusing them to create new XSTs. Such copying would be quite expensive and might not be necessary: if the tree is going to be copied again but not traversed, the copy is unnecessary. To reduce the number of times XST nodes are copied, zeta implements copying using a variant of copy-on-reference. Specifically, each XST node contains a pointer back to its parent. Two different XST nodes might point at the same child, but the child node contains a `parent` field recording which is the actual parent. The other node needs to make a copy of the child the first time it traverses the child pointer. Zeta initializes the `parent` field in new XST nodes with a special value `NoParentYet`. When traversing a child pointer, if the child's `parent` is `NoParentYet`, then the current parent can adopt the child, rather than copying it, by changing the `parent` field.

6.4. Name lookup

Name lookup is a common operation in a compiler, making the representation of name scopes important. Because an explicit scope is computed for every node in an XST, it is important that scopes share storage. Considering only storage, implementing scopes as a linked lists is ideal: an XST with n nodes and d declarations would require only $O(d)$ storage, not $O(n)$ or $O(dn)$. Unfortunately, name lookup in a scope implemented as a linked list could require $O(dk)$ time for k lookups. Following a suggestion by Appel [1] and an implementation by Okasaki [46], zeta provides a functional (also called persistent or immutable) map type based on red-black trees, which xoc uses to implement scopes. These require $O(d \log d)$ space but only $O(k \log d)$ time for k lookups.

In the current zeta implementation, using functional maps instead of linked lists more than halves the overall run time of checking `do_mounts.c`.

6.5. Summary

Zeta and xoc need to work correctly, but they also need to work quickly. We have found some ways to make them run faster than they did originally, but they still run about fifty times slower than the actual Sparse tool and about twelve times slower than an optimizing compiler. This chapter detailed the opportunities we have found for speeding up zeta and xoc. More work will be required in the future to match the speed of traditional compilers. CPU time profiling shows no obvious "hot spots" in the current implementation. We believe improvements are likely to be algorithmic, reducing either the number of nodes in the XST or the number of times they are copied.

CHAPTER 7

Discussion and Future Work

The extensions presented in Chapter 5 and the performance presented in Chapter 6 are both evaluations of the extension-oriented syntax tree interface described in Chapter 3. This chapter continues the evaluation qualitatively. It considers each of the goals set out for extension-oriented compilers and how well the work meets them. Then it considers each of the XST interfaces and how well each worked. Finally, it addresses some common questions about the work. These discussions also provide opportunities to discuss future work.

7.1. Extension-oriented compiler goals

Chapter 1 laid out the goals for an extension-oriented compiler, that extensions be powerful enough to write real-world extensions; that they be simple to write, not requiring detailed knowledge of the compiler; and that they be composable, so that independently-written extensions can be used together. This section examines how well xoc and its extensions meet each of these goals.

7.1.1. Power

The first goal for an extension-oriented compiler is that the extensions be powerful. An extension-oriented compiler is useless if it does not support the languages people use and the kinds of extensions people want to write. Chapter 4 demonstrated that XSTs can be used to create a C compiler. Chapter 5 showed how to recreate two existing extensions to C, extensions that were important enough to expert programmers that they wrote tens of thousands of lines of code to implement them. Those extensions are the Alef programming language and the Sparse program checker. They add new syntax, new data types, new type restrictions, and new analyses to C. We believe these categories span most of the kinds of extensions that people would want to add to C.

There is, however, one important limitation to what xoc can implement. Because xoc is only a compiler front end and invokes a standard C compiler as its back end, we cannot use xoc to implement extensions that change the behavior of the back end, like most implementations of exceptions or garbage collection. We believe an extension-oriented back end is an interesting topic for future research, as discussed below.

7.1.2. Simplicity

The second goal for an extension-oriented compiler is that the extensions be simple to write, meaning both that they are short and that they don't require detailed knowledge of the compiler. Qualitatively, grammars make it easy to add new syntax, syntax patterns and canonicalizers allow manipulation of input programs in terms of input syntax, and attributes provide a simple interface to compiler functionality. All of these reduce the amount that extension writers need to know about the compiler internals. Quantitatively, the thirteen extensions listed in Chapter 5, Figure 1 range from 16 to 245 lines, totaling 883 lines. They use only five distinct grammar symbols (`attr`, `expr`, `stmt`, `type`, and `typespec`). They use or extend only seven distinct attributes and seven distinct extensible functions. These statistics demonstrate that writing extensions requires knowing only a small amount of detail about the underlying compiler.

7.1.3. Composability

The third goal for an extension-oriented compiler is that the extensions be composable, meaning that extensions should be usable together unless they make conflicting changes to the language. This goal is the hardest of the three to quantify or even to define precisely. In the absence of a formal definition of "conflicting changes," let us consider a few examples.

First, two extensions have a syntactic conflict if they both introduce the same new syntax. For example, if one extension introduces >>> as a bitwise rotate operator while a second extension introduces the same syntax as an unsigned right shift (like in Java), the two extensions cannot be used together: an input file using the >>> operator will have two different parses, so zeta's parser will report an ambiguity. Extensions that add different new operators or that add different kinds of syntax (for example, one that adds a new kind of statement and another that adds a new kind of expression) are expected to work together: most likely, one would override the other.

Extensions that make orthogonal semantic changes should also work together. For example, the Alef channel extension introduces a new data type, while the Alef iterator extension introduces new control flow. These should definitely be composable (and they are). Extensions that make related semantic changes should also work together, unless the changes are at odds with each other. For example, the Alef channel extension and the Sparse pointer qualifier extension both make changes to the type system. The former introduces a new data type, and the latter introduces a new pointer qualifier. These should be (and are) composable, so that a program can use `chan(void user*)` for a channel carrying user pointers. On the other hand, an (imagined) extension allowing implicit conversions between pointers and integers weakens the type system. Such an extension would be at odds with an extension like Sparse, which strengthens the type system. We do not expect that these two extensions to work well together.

Most of xoc's composability of extensions arises naturally from its use of attributes. For example, two extensions that introduce different, new kinds of expressions cooperate to type-check an expression using both new syntaxes because both extensions extend the `type` attribute and only handle the cases that they recognize.

Some of the composability required extra planning. For example, an extension that introduces new control flow, like the Alef iterator, should ideally compose with an extension that analyzes control flow, like the Sparse context checker. These two extensions do compose, but making that happen required a small amount of planning. As described in section 4.7, the control flow graph module in the xoc core applies only to the `COutput` syntax type. Extensions that use it for flow analysis must pass in the `compiled` form of the tree, in which all extensions have been translated to standard C. Thus the Sparse context checker does handle Alef iterator expressions, because it sees the equivalent C code instead.

Xoc does not guarantee composability, nor does it detect semantic conflicts between extensions. For example, if one extension restricts the type system, nothing stops a second extension from overriding the restriction. Sometimes, this is even desired: the sparse_addrspace extension rejects all pointer casts that change the address space, and the sparse_force extension adds a new cast keyword `force` that overrides the address space restriction. Being able to modify the effects of other extensions might sometimes be a mistake, but as this example shows, it can also be useful.

We believe that a type system or some other precise characterization of extensions that could *guarantee* composability (or perhaps at least non-interference) is interesting future work.

7.2. Extension-oriented syntax tree interfaces

Chapter 2 described four key interfaces to extension-oriented syntax trees: extensible grammars, syntax patterns, canonicalization, and attributes. This section examines how well each of these worked in building xoc and its extensions.

7.2.1. Extensible grammars

Extensible grammars allow the use of any context-free grammar, in contrast to tools, such as yacc, that limit themselves to LR(1) or LALR(1) grammars. Being able to handle any context-free grammar is important for composability, because the class of general context free grammars is closed under union, but subclasses like LR(1) and LALR(1) are not. Restricting the entire grammar to be in one of these subclasses could lead to unfortunate composability problems, in which two modules worked fine in isolation but could not be used together due to creating a non-LR(1) grammar.

Because they handle ambiguity cleanly, extensible grammars also make it possible to solve old problems in new, cleaner ways. The best example of this is xoc's handling of the C type versus name ambiguity, described in Chapter 4. Most compilers take it as a given that they will need to tightly couple the lexer and the semantic analysis in order to parse C; using a GLR parser makes it possible to cut the two apart.

Extensible grammars, because they are extensible, make it natural to split a language into modular pieces. For example, even in the xoc core, the GNU C built-ins are isolated in a separate file and implemented as an (always-loaded) extension.

7.2.2. Syntax patterns

After years of working with yacc and other parser generators, the combination of syntax patterns and extensible grammars has been one of the nicest parts about using zeta and xoc. Not needing to write explicit parsing actions and not needing to define a separate abstract syntax tree data structure makes it easy to experiment with and prototype new syntax.

In addition to being easier to use than a traditional data structure interface, syntax patterns hide the XST effectively. As an example of effective XST hiding, Chapter 6 described changing the implementation of `expr[,]+` from being right-recursive to left-recursive, for performance reasons. We made this change after implementing all of xoc, but changing that detail required no changes to xoc. The syntax pattern interface to XSTs completely hid that detail, creating flexibility in the choice of implementation.

7.2.3. Canonicalization

Canonicalization is an important addition to syntax patterns. Without canonicalization, there is no good way to manipulate C types using XSTs. An earlier implementation of xoc written before canonicalization had a separate grammar describing type semantics and required explicit conversions from the C form to the alternate form. In that system, introducing a new type, like Alef's channels, required making changes to two grammars and extending the functions that converted between the two. Automatic canonicalization of the C form makes it possible to manipulate C types as C types, eliminating the extra grammar and conversions. Eliminating these reduces the amount extension writers need to learn.

7.2.4. Attributes

Attributes are the glue that holds the extension-oriented compiler together and also binds the extensions to the compiler core. As a connective mechanism, they work well: writing extensions by extending the `typecheck`, `type`, and `compiled` attributes has seemed straightforward and natural. As an interface to the compiler core, they also work well: for example, using the `type` attribute to access the type checking of subexpressions also seems natural. More important than seeming easy to use, the `extend attribute` mechanism successfully provides a way for extensions to cooperate to make compiler changes. For example, when using the Alef and Sparse extensions together, a type like `chan(void user*)` has a well-defined meaning only because attributes make it possible for the two extensions to cooperate to define the semantics of the type. Thus, as an extension mechanism and extension interface, attributes are a success.

Implementing the xoc core using attributes required fitting computations that are traditionally imperative into the attribute-driven evaluation model. The best example of this is the computation of name scopes, described in Chapter 4. The final result is elegant, but the path there was not obvious. Programmers familiar with a functional language (especially a lazy functional language like Haskell) are likely to find attributes more comfortable than programmers used to imperative languages.

7.3. Common questions

When discussing extension-oriented compilers with other researchers, some questions arise repeatedly. This section examines each of these common questions.

7.3.1. Why compile C?

One common question is why we chose C as the input language. The simple answer is that we wanted to build a system that we would use, and we tend to write programs in C and C++. C certainly has some rough edges (see Chapter 4), but it is far simpler than C++. Also, those rough edges turned out to be good demonstrations for the power of XSTs.

7.3.2. Why use zeta? Why not write xoc using itself?

Another common question is: now that you can extend C, why not write xoc in extended C rather than zeta? This would be an interesting exercise, but our focus has been on the extension mechanisms themselves (XSTs in particular), not the language they are embedded in. We believe that XSTs could be added to any existing compiler language, though they might fit better into some than others. For example, the XSTs work nicely with zeta's built-in data types, specifically lists and functional maps, and syntax patterns fit well into zeta's pattern matching syntax. These suggest that, for example, the XST interfaces might fit better in ML [42] than in C.

Xoc started out being written in itself, before XSTs as they are today, before extensible grammars, before syntax patterns, and before attributes. We successfully added extensible grammars, syntax pattern constructors, and extensible functions to that version of xoc, but since it didn't have syntax pattern matches, it required explicit parsing actions to be specified alongside the grammar rules. When we realized we could replace the explicit data structure representation with syntax pattern matches, it was difficult to plan a bootstrapping path. In the end, we abandoned the C implementation and added the XST support to a simple interpreted language we had lying around to produce zeta. Zeta turned out to be a fruitful platform for experimenting with new ideas. The details of syntax patterns changed a couple times, and we tried multiple iterations on the general idea of attributes before we settled on the version used in this dissertation. Zeta also served as a pliable base for Tom Bergan to explore a custom language for describing type checking rules [7].

In short, we believe XSTs could be added to any language with some effort, but zeta made it easy for us to experiment to decide exactly how XSTs should work.

7.3.3. Why context-free grammars? Why not PEGs and a packrat parser?

As mentioned above, the most important reason to use a GLR parser instead of an LALR(1) parser as in yacc is that the class of context free grammars (valid GLR inputs) is closed under union, while the class of LALR(1) grammars is not. Packrat parsers [19] accept parsing expression grammars (PEGs), which are also closed under union, so a natural question is: why not use PEGs instead of GLR?

PEGs replace the unordered choice (alternation) operator of context free grammars with an ordered choice operator. Doing so eliminates any possibility of ambiguity in the grammar: if an input can be parsed using two different rules, the order can be consulted to determine which rule is the correct one.

In our work, we have found that being explicit about ambiguity is useful. The fact that the GLR parser returns all possible parses makes it possible to solve the type-versus-name ambiguity while keeping the parser and semantic analysis separate. The fact that the GLR parser returns all possible parses also makes it possible to infer the symbols for most slots in syntax pattern matches. Finally, the fact that the GLR parser returns all possible parses makes it possible to detect ambiguity created when multiple extensions create multiple ways to parse the same input syntax; xoc can then report the situation as an extension conflict.

7.3.4. What about an extension-oriented compiler back end?

Our work has focused exclusively on the compiler front end. One reason is that many more programmers are interested in creating new language features (i.e., extending the front end) than optimizing the compiler output (i.e., extending the back end). Some front end extensions could benefit from tighter coupling with the back end. For example, the fact that xoc must translate extensions to standard C precludes (or at least makes difficult) the implementation of extensions that depend on changes to code generation, like exceptions and garbage collection.

We believe that a fully extension-oriented compiler would be an interesting avenue for future research. In particular, being able to extend the back end could make it possible for users to take advantage of new hardware instructions.

7.3.5. What guarantees can be made about composability of extensions?

Composability of extensions is the main feature that sets extension-oriented compilers like xoc apart from extensible compiler toolkits like Polyglot and xtc. Thus a natural question is what guarantees can be made about composability. The system described in this thesis makes no formal guarantees. We have focused on creating a system in which composability is possible and common, but have no methods for guaranteeing it or analyzing two extensions to see whether they compose.

It would be interesting to study whether an extension-oriented compiler could restrict extensions in a way that would not affect actual uses but could guarantee composability. It would also be interesting to study more expressive type systems for extensions that might be able to check that a particular extensions composes. However, until we have more experience with the kinds of extensions programmers want to write, it seems premature to restrict them.

7.3.6. Will anyone but the xoc developers be able to write extensions?

We don't know. Like many research projects, we have no users outside our own developers. We believe that writing an xoc extension is simpler than writing a Polyglot or xtc extension, though, and both of those systems have external users.

7.3.7. Won't xoc make programs harder to debug?

There are two issues: debugging extensions themselves and then debugging programs that use extensions.

If an extension is incorrect, that manifests to an extension user as a compiler bug: the source code using the extension is correct, but the generated executable binary is not. These are some of the most frustrating bugs to track down, so it is important that extensions are correct. Since only the xoc developers have written extensions, we have no evidence about how error-prone writing an extension would be for an extension writer just learning the system. However, our experience has been that while we made and debugged many hard-to-find mistakes in the xoc core, all the mistakes we made writing extensions presented themselves immediately. The mistakes we made in the xoc core were design mistakes, not implementation errors. In particular, it took multiple tries to design a namescope attribute that was correct and did not cause attribute cycles. Extension writers are not typically exposed to these issues, because they are extending one or two well-known attributes, not designing the attributes.

Once the extensions are correct, debugging programs that use extensions requires making the debuggers aware of the language changes. For extensions that are mainly syntactic sugar, xoc already does the work of generating line number annotations in the C translation that point back to the original source file. Thus, source listings and line-by-line stepping in the debugger already work. Extensions that make semantic changes to the language would benefit from being able to extend the debugger. For example, an extension that adds new types might extend the debugger to display values of those types in a convenient format. Extensible debuggers already exist (for example, acid [68] and deet [25]) and would complement an extension-oriented compiler nicely.

7.3.8. Won't xoc create a "Tower of Babel" problem?

The most common question others have asked is this: if extension-oriented compilers allow programmers to create their own language adaptations, won't it lead to a world in which each programmer writes in his own language and no one can read each other's programs?

The first answer to this question is glib: if that situation came to pass, what success it would imply for our work! What influence it would imply for our ideas! We can only hope to be so lucky.

A more considered answer is that programmers today already work in different dialects: each code library introduces new functions, assigning special meanings to new words. And yet programmers understand each other's code, because programmers working together take the time to establish a shared vocabulary and understanding. In fact, an extension-oriented compiler might make it easier to learn a new library, by providing a simpler or more type-safe interface, like the Alef extension in Chapter 5 does for libthread. Even Bjarne Stroustroup, the creator of C++, has argued for the ability to make library-specific language adaptations [58]. A large project might require some restraint to keep all programmers using only a fixed set of extensions, but this is no different than keeping programmers using only a fixed set of libraries. In both cases the solution is social, not technological.

7.3.9. What is the expected influence of this work?

As mentioned in the introduction, most of today's software can be extended via plug-in modules. Mainstream compilers have not yet followed suit, but we believe that, in time, they will. When they do, we hope they will adopt the approach described in this dissertation. However, this approach is such a large departure from the typical structure of a mainstream compiler that it is unlikely that such a compiler would adopt it all at once. Instead, we hope that mainstream compiler authors can learn from our work and perhaps find ways to take apply pieces of our approach to their own compilers. Most importantly, we hope that when mainstream compilers do adopt plug-in extensibility, they strive as we have to make the extensions as easy as possible to write and to compose.

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.

This dissertation introduced extension-oriented syntax trees (XSTs) as a mechanism for building extension-oriented compilers. The key interfaces to XSTs are extensible grammars, which allow piecewise definition of grammar rules and define the parser and the form of XSTs; syntax patterns, which allow the manipulation of XSTs using concrete input syntax; canonicalizers, which allow transformation of the XST into a convenient, standard form; and attributes, which provide both a mechanism for exposing core functionality and a framework in which multiple extensions work together to implement new functionality.

This dissertation also described xoc, an extension-oriented compiler for C built using XSTs. Then it described two suites of extensions built using xoc; the extension suites recreated functionality from Alef, a C-like language developed at Bell Labs in the 1990s, and Sparse, a program checker used by Linux kernel developers. These demonstrate that xoc extensions can meet our goals: they are simultaneously powerful, simple, and composable.

We hope that the work described in this dissertation inspires future exploration of extension-oriented compilation by others, and we look forward to the day when these ideas will be incorporated into mainstream compilers.

Zeta language definition

Zeta's most important feature is its support for extension-oriented syntax trees, as described in Chapter 3. This appendix defines the syntax and semantics of the rest of the language.

A.1. Names and scopes

In a zeta program, a name like `x` might refer to a module, a variable, a function, or a type depending on context. Names can only refer to declarations that occur earlier in the source file in the current scope or in an enclosing one. Scopes can be managed explicitly using the `{ }` statement block construct. The `{` starts a new scope, and the `}` ends the scope. Any names introduced between the braces are not available after the braces close. Because names can be declared by both statements and expressions, statements and expressions that involve control flow typically introduce their own scopes. These are noted below.

As an example, the small zeta program:

```
{
    typedef x: int;
    a: x;
    b := a + 1;
}
```

defines three names: `x` is an alias for the type `int`, `a` is a variable of type `int`, and `b` is also a variable of type `int`. All three names go out of scope at the closing brace.

A.2. Modules

Zeta programs are made out of units called modules. A module is defined by a single zeta source file; its name is the base name of the file: `typecheck.zeta` defines a module named `typecheck`.

The names that are in scope at the end of a module's source file can be referred to by a second module after it imports the first. For example, if a module `m` defines a function `f`, a second module can invoke that function as `m.f` or `f` depending on which variant of the `import` statement is used to import `m`.

The first time module `m` is imported, zeta loads the module, running all the top-level statements in the module's zeta source. Subsequent import statements do not reexecute the statements.

A.3. Data types

Zeta provides basic data types, a small collection of container types, data structures, functions, and extension-oriented syntax trees. Zeta's functions can be polymorphic in the style of ML. To keep the implementation of polymorphism simple, all zeta data types are represented by word-sized (32-bit) values. Types larger than 32 bits—strings, tuples, lists, arrays, and so on—are represented by pointers to heap-allocated storage.

Zeta compares two values for equality by comparing their 32-bit values. For booleans and numbers, this produces the usual result. Zeta takes special care with strings to ensure that strings with identical contents point at the same instance, so that the pointer comparison reflects whether the actual string contents are identical [23]. Larger data types such as tuples, lists, arrays, and structs, only compare equal if they are the same instance. For example, after

```
a := (1, 2);
b := a;
c := (1, 2);
```

the three variables `a`, `b`, and `c` all point at tuples containing the integers `1` and `2`; `a` and `b` will compare equal to each other, but `c` will not compare equal to either.

Zeta automatically initializes variables and data structures by zeroing them, making "used and not set" bugs impossible. Thus, an uninitialized `bool` is always false, an uninitialized `int` or `float` is always zero, and an uninitialized value of one of the more complex types is always `nil`.

A.3.1. Basic types

Zeta's basic data types are `bool` (boolean), `int` (32-bit integer), `float` (32-bit floating-point), and `string` (32-bit pointer to character string). As mentioned above, zeta's string type arranges that pointers to the same string contents use the same pointer, so that comparing string pointers for equality is equivalent to comparing their contents. The empty string is represented by the nil pointer. Thus uninitialized strings default to the empty string.

A.3.2. Tuples

A tuple type is defined by a list of contained types, like `(int, string)`. A tuple of one type is the same as the type itself: `(int)` and `int` are the same type.

The type `(int, string)` is a pointer to a record containing an `int` and a `string`. Unlike tuples in many languages, it is possible for a tuple-typed variable to be `nil`.

A.3.3. Lists

A list type is defined by its contained type, like `list int` or `list list string`. or `list (int, string)`. Lists are implemented as singly-linked lists and are immutable.

A.3.4. Arrays

An array is defined by its contained type, like `array int` or `array (int, string)`. Arrays have fixed size and allocate their objects in a contiguous memory region for constant-time indexing. Arrays are mutable.

A.3.5. Maps

A map is defined by its key and value types, like `map(int, string)`. Maps are mutable data structures implemented using red-black trees. The key type is compared using ==, which may not be the desired comparison for composite types like lists or tuples. See the discussion of equality below. The map type in zeta holds a pointer to the actual map, which must be explicitly allocated. For example,

```
m: map(int, string);      m is a nil pointer
m = map {1 => 2};         m now points at a new map with one entry
print m[1];               prints 2
p := m;                   p points at the same map as m
p[1] = 2;
print p[1];               prints 2
print m[1];               prints 2
```

Map lookups for non-existent keys return the default zero value for the map's value type.

A.3.6. Functional maps

Functional maps are like maps but are immutable. The only way to add to a functional map is to create a new functional map and reassign the variable holding the map:

```
m: fmap(int, string);     m is an empty map
m = fmap {1=>2, 3=>4};    m now points at a new map with two entries
print m[1];               prints 2
p := m;                   p points at the same map as m
p = fmap {p, 3=>5};       p now maps 3 to 5
print p[1];               prints 2
print p[3];               prints 5
print m[3];               prints 4
```

If `m` is a functional map, it is illegal to assign to the expression `m[1]`.

A.3.7. Structs

A struct is a heap-allocated record containing a series of fields. Struct types are given names as part of their definitions. For example,

```
struct Point {
    x: int;
    y: int;
};
```

defines a struct type `Point` whose members contain two integers `x` and `y`.

Structs can contain tagged unions, which introduce subtypes. For example:

```
struct Point {
     x: int;
     y: int;
     tag Two {
     }
     tag Three {
       z: int;
     }
}
```

defines three types `Point`, `Point.Two`, and `Point.Three`. A value of type `Point.Two` or `Point.Three` can be used where a value of type `Point` is expected (they are both subtypes of `Point`). If `p` is a value of static type `Point`, then only its `x` and `y` fields can be accessed. A run-time subtyping check can establish whether `p` actually points at a `Point.Three`, making the `z` field accessible. Subtyping is discussed in detail below.

A.3.8. Extension-oriented syntax trees

Extension-oriented syntax trees are discussed in detail in Chapter 3. In addition to the type names introduced by grammars (for example, `C99.expr`), the type `Xst` denotes a value of any XST type.

A.3.9. Functions

Function types are defined by a list of argument types and a single return type (returning multiple values can be accomplished using a tuple type). For example,

```
f: fn(int, string): bool;
```

defines a variable `f` that is a (pointer to a) function taking `int` and `string` arguments and returning a `bool`. Zeta makes no distinction between function variables and traditional function definitions. To zeta, a function definition is simply a declaration of a function variable along with an initial value.

A.3.10. Modules

After importing a module `m`, the name `m` has “module” type. While there can be names of module type, there are no values of module type, so `m` cannot be copied, nor assigned to another variable, nor passed to a function.

A.3.11. Nil

The expression `nil` has nil type and can be used as any type except `bool`, `float`, and `int`.

A.3.12. Subtyping

The presence of tagged structs, XSTs, and polymorphism introduces the concept of subtyping, not only in those types but in any container types. Define that `t` is a subtype of `u` (written `t <: u`) if every value of type `t` is also a value of type `u`. Put another way, knowing that a value has type `t` is more specific than knowing it has type `u`.

It should be clear from the definition that any subtype relation must be reflexive and transitive. Zeta's subtype relation is the reflexive transitive closure of the following base rules. In what follows, names in italics stand for any type. Starred names stand for any list of types.

First, `nil` can be used as any type represented by a pointer:

`nil <: string`
`nil <:` *T* — for any tuple type *T*
`nil <: list` *a*
`nil <: array` *a*
`nil <: map (`*a*`,` *b*`)`
`nil <: fmap (`*a*`,` *b*`)`
`nil <:` *S* — for any struct type *S*
`nil <:` *X* — for any XST type *X*

Second, *T* `<:` *S* if *T* denotes a tagged struct type contained in the struct *S*. (In the example above, `Point.Two <: Point`).

Third, *X* `<:` *Y* if *X* and *Y* are both XST types and *X* is a subtype of *Y* as described in Chapter 3. Also, *X* `<: Xst` for any XST type *X*.

Finally, the rules for composite types:

`(`*t0*, *t1*, ..., *tn*`) <: (`*u0*, *u1*, ..., *un*`)`,
  if *t0* `<:` *u0*, *t1* `<:` *u1*, ..., and *tn* `<:` *un*.

`fn(`*t0*, *t1*, ..., *tn*`):` *tr* `<: fn(`*u0*, *u1*, ..., *un*`):` *ur*,
  if `(`*u0*, *u1*, ..., *un*`) <: (`*t0*, *t1*, ..., *tn*`)` and *tr* `<:` *ur*.

`list` *a* `<: list` *b*, if *a* `<:` *b*.

`fmap(`*a*, *b*`) <: fmap(`*c*, *d*`)`, if *c* `<:` *a* and *b* `<:` *d*.

`array` *a* `<: array` *b*, if *a* = *b*.

`map(`*a*, *b*`) <: map(`*c*, *d*`)`, if *a* = *c* and *b* = *d*.

The rules for `array` and `map` are stricter than those for `list` and `fmap` because the former are mutable data structures.

A.3.13. Parametric polymorphism

Zeta provides polymorphic functions in the style of ML, allowing function arguments and return types to use type variables like `'a` to denote a variable of unknown type. Values of the same variable type can be tested for equality using the == operator.

A.4. Statements

Most statements in zeta can appear either inside function bodies or simply as top-level statements in a module. The exceptions are include statements and grammar statements, both of which can only be used at top level. A module's top-level statements are executed when the module is loaded, as explained above.

A.4.1. Expression statement

stmt: *expr* `;`

An expression statement evaluates the expression it contains.

A.4.2. Print statement

stmt: `print` *expr* `;`

A print statement prints the expression, followed by a newline character.

A.4.3. Typedef statement

stmt: `typedef` *name* `:` *type* `;`

A typedef statement introduces *name* into the current scope as an alias for *type*.

A.4.4. Statement block

stmt: `{` *stmt** `}`

A statement block executes its statements in sequence. Statement blocks introduce a new scope; names declared inside a block are no longer usable once the block ends, even though their values may be.

A.4.5. Conditional statements

stmt: `if (` *expr* `)` *stmt*
stmt: `if (` *expr* `)` *stmt* `else` *stmt*

An `if` statement is made up of an expression called a condition and one or two sub-statements called branches. The `if` statement first evaluates the condition. If the condition is true (that is, not false, zero, or nil), the first branch is executed; otherwise the second branch is executed, if there is one. An `if` statement executes in its own scope: if the *expr* declares a name, that name is available both branches but not to statements following the `if`. Similarly, the two branches execute in their own scopes: names introduced in the first branch cannot be referred to in the second branch.

Expressions can declare different names when they evaluate true than when they evaluate false. For example, the pattern match `a ~ list [b, c]` only introduces the names `b` and `c` when it evaluates true. The condition's true declarations are available only in the true branch; its false declarations are available only in the false branch.

A.4.6. Assertion statements

stmt: `assert` *expr* `;`

An assertion statement evaluates its condition *expr* and halts program execution if expr is false (that is, false, zero, or nil). The condition's true declarations are made available in the scope in which the `assert` appears. For example, after this asserted pattern match:

```
assert a ~ list [b, c];
```

the names `b` and `c` will be in scope. Thus, an assertion statement's effect on its scope differs from the otherwise equivalent conditional:

```
if(!expr) halt();
```

A.4.7. Loop statements

stmt: `while (` *expr* `)` *stmt*

The `while` statement evaluates *expr*. If *expr* is true, it executes *stmt* and then repeats. The *stmt* executes in its own scope. Names introduced by a true evaluation of *expr* are available to *stmt*; names introduced by a false evaluation are made available in the scope in which the `while` statement appears. Thus, the `while` statement can mimic even the scope effects of an assertion:

```
while(!expr) halt();
```

stmt: `do` *stmt* `while (` *expr* `);`

The `do-while` statement's execution is like `while`'s, except that it runs *stmt* once before checking *expr*. Thus, names introduced by *expr* (true or false) are not available outside of *expr*, while names declared by *stmt* are made available in the scope of the `do-while`. (Of course, if *stmt* is itself in braces (i.e., a statement block), it declares no names.)

stmt: `for (` *expr*? `;` *expr*? `;` *expr*? `)` *stmt*

The three-expression `for` loop executes identically to that of C. The loop executes in its own scope, as does *stmt*. Names declared by a true evaluation of the condition *expr* are made available in *stmt*. No names declared during the loop are made available in the scope in which the `for` statement appears.

stmt: `for (` *expr* `in` *expr* `)` *stmt*

The `for-in` statement iterates over a collection—an array, list, map, or functional map. Like the three-expression `for` loop, it executes in its own scope: no names are declared in the scope in which the loop appears. The `for-in` loop's iterator variable—the first *expr*—is treated as a declaration and must be appropriate to the collection being iterated. For arrays and lists, the iterator variable must be a simple name like `x`. For maps, the iterator variable can be either a simple name or a pair `(k, v)` that will be initialized with successive keys and values. If a simple name is given, it is initialized with a tuple containing the key and value.

stmt: `break;`

stmt: `continue;`

The `break` and `continue` statements behave as in C: `break` stops the innermost loop, while `continue` skips over the rest of the innermost loop's *stmt* body, starting the next iteration.

A.4.8. Switch statements

stmt: `switch ( `*expr*` ) { `*case**` }`
case: `case `*expr*` : `*stmt**

A switch statement evaluates its expression once and then matches it against each of the case expressions in turn. When a match is found, it executes the associated statement list; further matches are not considered. The case expressions can be arbitrary patterns as described in the pattern matching section below. The switch statement is identical to an `if else if` ... sequence with pattern matches as conditions, except that side effects in the switch *expr* are executed exactly once.

A.4.9. Functions

stmt: `fn `*name*` ( `*fnarg*[,]*` ) : `*type*` { `*stmt**` }`
fnarg: *name*` : `*type*

A function statement defines a function body associated with *name*. If *name* has not already been declared, the function statement also declares *name* as a variable of function type. The return type annotation `:` *type* is optional. If omitted, the function is assumed to have the empty tuple as its return type.

A.4.10. Import statements

stmt: `import `*name*` ;`
stmt: `from `*name*` import `*name*[,]+` ;`
stmt: `from `*name*` import * ;`

The import statements load the named module and make its names available. The first form (`import` *name*) declares *name* in the current scope. The module's own names can be accessed via dot expressions, as in `m.var`. The second form does not declare the module name in the current scope; instead it makes the listed module names available directly. The third form is like the second but makes all names declared in the module available directly.

A.4.11. Include statement

stmt: `include `*file*` ;`

The `include` statement, which can only appear at top level, replaces itself in the input with the contents of *file*.

A.4.12. Grammar statement

The `grammar` statement, which can only appear at top level, defines a new grammar. Grammar statements are discussed in detail in Chapter 3.

A.5. Expressions

A.5.1. Constant expressions

expr: `true`
expr: `false`
expr: *integer*
expr: *floating*
expr: `"` *string* `"`
expr: `nil`

These expressions evaluate to the obvious constants.

A.5.2. Constructors

expr: `list [` *expr*[,]* `]`
expr: *expr* `::` *expr*

These expression create new list values. The first form creates a new list with the given expressions as members; they must all be subtypes of some common type. If no expressions are given, the first form evaluates to `nil`. The second form creates a new list from a head and a tail.

expr: `(` *expr*[,]* `)`

A parenthesized expression list with zero or two or more expressions creates a new tuple value. A parenthesized single expression evaluates to just that expression.

expr: `array [` *expr*[,]* `]`
expr: `array (` *expr* `)`

These expressions create new array values. The first form creates a new array with the given expressions as members; they must all be subtypes of some common type. The second form creates a new array whose length is given by *expr*; the members are zeroed. The second form has type "array of any," and can be used as any array type. This is type-safe because it is not possible to create explicit variables of type "array of any," so there is no way to use the second form as two different types simultaneously.

expr: `map {` *keyvalue*[,]* `}`
keyvalue: *expr* `=>` *expr*

This expression creates a new map, optionally populating it with the given key/value pairs. The key expressions must all be values of a common type, and the value expression must all be values of a common possibly-different type.

expr: `fmap { }`
expr: `fmap {` *expr* `,` *keyvalue*[,]* `}`

These expression create new functional maps. The first form creates an empty map and evaluates to `nil`. The second form starts with an existing map and applies the given key/value pairs to create a new map. Functional maps are immutable: the existing map is not changed.

A.5.3. Declaration expressions

expr: *name* `:` *type*

This expression declares a new variable with the given name and type. The variable is initialized to the zero value for that type. Thus, the expression `(a: int) + (b: int)` evaluates to zero but declares two new integer variables in the process.

expr: *expr* := *expr*

This expression declares and initializes a new variable or variables. The type of the new variable or variables is inferred from the type of the initial value. The expression on the left can be either a simple name, as in `x := 1`, which declares and initializes a new integer variable, or can be a possibly-nested tuple of names, as in `(a, b) := (1, 2)` or `(a, (b, c)) := (1, (2, 3))`.

A.5.4. Assignment expressions

expr: *expr* = *expr*

The assignment expression changes the value of the left expression to be that of the right expression. The left expression must be assignable, meaning it is a variable name, an array element, or a map element.

expr: *expr* += *expr*
expr: *expr* -= *expr*
expr: *expr* *= *expr*
expr: *expr* /= *expr*
expr: *expr* %= *expr*
expr: *expr* <<= *expr*
expr: *expr* >>= *expr*
expr: *expr* &= *expr*
expr: *expr* ^= *expr*
expr: *expr* |= *expr*

Like in C, these expressions are shorthands for *expr* = *expr op expr*, where the left expression's side effects are evaluated only once.

A.5.5. Variable references

expr: *name*

If the current scope defines *name* as a variable, the expression evaluates to *name*'s current value. Variable references can be assigned to.

A.5.6. Casts

There are no automatic type conversions in zeta expressions, unlike in C. All conversion between types must be done explicitly using casts.

expr: `bool (` *expr* `)`

This expression converts an expression of any value into a boolean: false, 0, and nil convert to false and any other values convert to true.

expr: `int (` *expr* `)`
expr: `float (` *expr* `)`

This expression converts a boolean, float, or string to an integer or float. Strings are converted by interpreting them as base 10.

expr: `string ( `*expr*` )`

This expression converts a boolean, integer, float, or XST to a string. XSTs are converted by concatenating all the string literals in the rules in the tree; the result is intended for debugging and may or may not be syntactically valid.

expr: `list ( `*expr*` )`
expr: `array ( `*expr*` )`

These expressions convert arrays to lists and vice versa.

A.5.7. Equality expressions

expr: *expr* == *expr*
expr: *expr* != *expr*

These expressions check for equality (or inequality) of their operands, which must both be subtypes of some common type. As explained above, equality on non-basic types is only pointer comparison, not a deep equality check.

A.5.8. Comparison expressions

expr: *expr* < *expr*
expr: *expr* <= *expr*
expr: *expr* > *expr*
expr: *expr* >= *expr*

These expressions compare their operands, as in C. Both operands must be the same type; only `int`, `float`, and `string` can be compared.

A.5.9. Arithmetic expressions

expr: *expr* | *expr*
expr: *expr* ^ *expr*
expr: *expr* & *expr*
expr: *expr* << *expr*
expr: *expr* >> *expr*
expr: *expr* + *expr*
expr: *expr* - *expr*
expr: *expr* * *expr*
expr: *expr* / *expr*
expr: *expr* % *expr*
expr: ! *expr*
expr: ~ *expr*
expr: + *expr*
expr: - *expr*

On integer and floating-point values, these expressions have the same meaning as in C. There are no implicit conversions between the arithmetic types: binary operators must have operands of equal type.

If both *expr*s are lists, then *expr* + *expr* evaluates to the concatenation of the two lists. It shares storage with the second operand, but not the first. If both *expr*s are strings, then *expr* + *expr* evaluates to the concatenation of the two strings.

A.5.10. Lookup expressions

expr: *expr* `.` *name*

The dot lookup syntax provides access to a name that exists within *expr*, which may be a module name, a struct type name, a grammar type name, or a struct value.

A.5.11. Indexing and slices

expr: *expr* `[` *expr* `]`

An indexing expression extracts a member from a collection. It extracts an element from an array, a member from a tuple, a character from a string, a keyed value from a map, an element from a list, and a keyed value from an association list. Since zeta has no character type, an extracted string character is represented by the corresponding one-letter string. Indexing into a list with an integer index requires time proportional to the index size: `l[0]` is equivalent to `l.hd`, and `l[3]` is equivalent to `l.tl.tl.tl.hd`. If `l` is a list of (key, value) pairs, then indexing into `l` with an index of the key type will search the list for the first pair with the given key and return the corresponding value. It is usually a better idea to use a functional map. Keyed lookups that fail return the default zero for the value type.

expr: *expr* `[` *expr* `,` *expr*? `]`

The slice expression extracts a range of members from an array or string. If the second expression is omitted, the slice extends to the end of the array or string Negative indices will have the length of the array or string added to them before use, so that `x[-5,]` returns the last five elements in an array.

A.5.12. Subtype queries

expr: *expr* `<:` *type*

Because of the subtype rules of tagged structures and also XST types, a variable declared as type *T* in a zeta program may, at run time, have a more specific type *U*. For example, a variable might be statically identified as a `Point` but at run time might sometimes be a `Point.Two`. Subtyping queries allow programs to dynamically query the type of a particular variable. If `v` is a value, the expression `v <:` *U* evaluates true if `v`'s actual type is a subtype of (or equal to) *U*. A common use of this in xoc is to check whether an XST of static type `C` is actually a more specific kind, like `C.expr`. If the expression being checked is a simple name, then the true scope of the subtype query re-binds that name to the more specific type:

```
x: C;
if(x <: C.expr){
     // here, x has static type C.expr
}
// now x has type C again
```

It is only valid to use dynamically typed values, like tagged structs or XSTs, in subtyping queries: subtype queries cannot be applied to strings, lists, or tuples, for example.

expr: *expr* `!<:` *type*

The `!<:` expression evaluates to the negation of the `<:` operator.

expr: `tag` `(` *expr* `)`

The `tag` built-in, which only applies to values with dynamic types, returns a string representation of the most specific type of its argument. For example, if `p` is statically typed as a `Point` but actually a `Point.Two` during execution, `tag(p)` would return the string `Point.Two`.

A.5.13. Function expressions

expr: `fn (` *fnarg*[,]+ `) :` *type* `{` *stmt** `}`

A function expression is an anonymous function body. Except for the lack of a name, it is handled identically to the function statement. In fact, the function statement simply declares a name and attaches the equivalent function expression as the body.

A.5.14. Pattern matches

expr: *expr* ~ *pattern*

A pattern match is a boolean expression checking whether *expr* matches the *pattern*. A successful pattern match can bind new variable names in the current name scope, so pattern matches are most often used as conditions. Patterns are described in detail below.

A.5.15. Conditional expressions

expr: *expr* && *expr*

expr: *expr* `||` *expr*

Like in C, && and `||` are short-circuit boolean operators: the expression on the right is not evaluated if it is not needed to determine the overall result. Unlike in C, conditional expressions have an effect on the name scope in which they appear. When an && expression evaluates true, both branches are guaranteed to have succeeded, so names generated by either successful match are added to the && expression's success scope. When it evaluates false, only names common to both expression's failures are added to the overall expression's failure scope. The rules for `||` are reversed: when it evaluates false, names generated by either failure are added to the `||` expression's failure scope. When it evaluates true, only names common to both successes are added to the success scope.

expr: `!` *expr*

The `!` operator negates the boolean value of *expr*, also exchanging its success and failure scopes.

A.6. Patterns

Pattern match syntax mimics ordinary expression syntax. There is no ambiguity because patterns appear only after the ~ operator in a `case` label, where expressions cannot. As a simple example, `x ~ list [ 1, y]` is true if `x` is a two-element integer list with first element `1`. In its success name scope, `y` is bound to the second value in the list.

A.6.1. Constants

pattern: *constant*

Constants like `3.14` and `true` are also patterns: they match that value.

A.6.2. Names

pattern: *name*

A name matches any value. After a successful match, that name is bound to the value it matched.

A.6.3. Conditional matching

pattern: *pattern* && *pattern*
pattern: *pattern* `||` *pattern*

The && and `||` operators have the same meaning in patterns as they do in ordinary expressions: an && pattern only matches if both clauses match, while an `||` pattern matches if at least one of the clauses matches.

A.6.4. Constructors

pattern: *structname* `(` *pattern*[,]* `)`
pattern: `array [` *pattern*[,]* `]`
pattern: `list [` *pattern*[,]* `]`
pattern: *pattern* `::` *pattern*
pattern: `(` *pattern*[,]* `)`
pattern: `...`

These constructors all match the same form they would construct as ordinary expressions. The *structname* denotes a structure name, perhaps with a leading module name followed by a dot. The wildcard `...` matches the end of an array, tuple, list, or struct.

A.6.5. Declarations

pattern: *name* `:=` *pattern*

The declaration pattern matches a value if *pattern* matches the value. It also binds *name* to that value.

A.7. Type expressions

Type expressions are different from types. The types described above are the actual types in the zeta type system. This section describes the syntax for naming types. The main difference is that type expressions can use `typedef`'ed names, while the type system does not.

A.7.1. Basic types

type: `int` | `float` | `string` | `bool`

These are the basic types.

A.7.2. Functions

type: `fn ( `*type*[,]* `) : type`

This expression denotes a function type with the named argument types and return type. Like in function statements, the return type is optional; if omitted it is taken to be the empty tuple.

A.7.3. Tuples

type: `( `*type*[,]* `)`

This expression denotes a tuple type.

A.7.4. Container types

type: `list` *type*
type: `array` *type*
type: `map (` *type* `,` *type* `)`
type: `fmap (` *type* `,` *type* `)`

These expressions denote the list, array, map, and fmap types.

A.7.5. Polymorphic types

type: `'` *name*

A polymorphic variable type is written as an apostrophe `'` followed by a name. Like in ML, polymorphic types are implicitly quantified at the top level, so `list 'a` denotes a homogeneous list.

A.7.6. Type names

type: *name*

A type name is a name given to a type by an earlier `typedef` statement. The name can also be prefixed by a module name and a dot, in order to name a type declared in an imported module.

Xoc extension listings

Xoc's extensions are short, but not short enough to list in full during Chapter 5. This appendix gives the full source listings for all the extensions discussed in that chapter.

B.1. Alef channel types

```
// Alef channel type
//
// chan(t) is a type for any type t.
// All chan(t) are represented by Channel*
// in the underlying C code.

from xoc import *;

extend grammar C99
{
    typespec: "chan" "(" type ")";
}

extend attribute
compiled(t: C.type): COutput.type
{
    if(t ~ ‘{chan(\_)})
        return ‘COutput.type{Channel*};
    return default(t);
}
```

B.2. Alef channel allocation

```
// Alef channel allocation statement

from xoc import *;
import alef_chan;
import util;

extend grammar C99
{
     stmt: "alloc" expr[,]+ ";";
}

extend attribute
typecheck(term: C)
{
     exprs: list C.expr;
     if(term ~ ‘C.stmt{alloc \(exprs);}){
          for(e in exprs)
               if(e.type && e.type !~ ‘{chan(\_)})
                    errorast(term, "alloc of non-channel");
          return;
     }
     default(term);
}

extend attribute
compiled(term: C.stmt): COutput.stmt
{
     exprs: list C.expr;
     if(term ~ ‘C.stmt{alloc \(exprs);}){
          bl: list C.blockitem;
          for(e in exprs){
               assert e.type ~ ‘{chan(\t)};
               bl = ‘C.blockitem{\e = chancreate(\(t.sizeof), 0);} :: bl;
          }
          bl = util.reverse(bl);
          return ‘C.stmt{ { \bl } }.compiled;
     }
     return default(term);
}
```

B.3. Alef channel send and receive operators

```
// Alef channel send and receive operations

from xoc import *;
import alef_chan;

extend grammar C99
{
     expr: "<-" expr
         | expr "<-=" expr;
}

extend attribute
type(term: C.expr): C.type
{
     if(term ~ ‘{<-\e}){
          if(e.type){
               if(e.type ~ ‘{chan(\t)})
                    return t;
               errorast(term, "recv from non-channel");
          }
          return nil;
     }
     if(term ~ ‘{\e <-= \v}){
          if(e.type && v.type){
               if(e.type ~ ‘{chan(\t)}){
                    if(canconvert(t, v.type))
                         return t;
                    errorast(term, "cannot send "+v.type.fmt+
                         " on chan of "+t.fmt);
               }else
                    errorast(term, "send on non-channel");
          }
          return nil;
     }
     return default(term);
}
```

```
extend attribute
compiled(term: C.expr): COutput.expr
{
    if(term ~ `{<-\e}){
        assert e.type ~ `{chan(\t)};
        tmp := mktmp(t);
        return `C.expr{
            (chanrecv(\e, &\tmp), \tmp)
        }.compiled;
    }
    if(term ~ `{\e <-= \v}){
        assert e.type ~ `{chan(\t)};
        tmp := mktmp(t);
        return `C.expr{
            (\tmp = \v, chansend(\e, &\tmp), \tmp)
        }.compiled;
    }
    return default(term);
}
```

B.4. Alef alt statement

```
// Alef alt statement

from xoc import *;
import alef_chan;
import alef_sendrecv;
import util;

extend grammar C99
{
    stmt: /alt/a "{" altclause* "}";
    altclause: "case" expr ":" stmt*;
}

extend attribute
typecheck(term: C)
{
    if(term ~ `C.stmt{alt { \clauses }}){
        for(c in clauses){
            assert c ~ `{case \e: \body};
            if(e.type == nil)
                continue;
            if(e !~ `{<-\_} && e !~ `{\_ = <-\_} && e !~ `{\_ <-= \_})
                errorast(e, "malformed alt case");
        }
        return;
    }
    default(term);
}
```

```
extend attribute
compiled(term: C.stmt): COutput.stmt
{
     if(term ~ ‘{alt { \clauses }}){
          n := length(clauses);
          alts := mktmp(‘C.type{Alt[\(n+1)]}); // +1 for CHANNOP
          inits: list C.stmt;
          body: list C.stmt;
          i := 0;
          for(c in clauses){
               assert c ~ ‘{case \e: \b};
               switch(e){
               case ‘{<-\chan}:
                    assert chan.type ~ ‘{chan(\t)};
                    inits = ‘C.stmt{{
                         \alts[\i].op = CHANRCV;
                         \alts[\i].v = nil;
                         \alts[\i].c = \chan;
                    }} :: inits;
                    body = ‘C.stmt{
                         {
                         case \i:
                              \b
                              break;
                         }
                    } :: body;

               case ‘{\val = <-\chan}:
                    assert chan.type ~ ‘{chan(\t)};
                    tmp := mktmp(t);
                    inits = ‘C.stmt{{
                         \alts[\i].op = CHANRCV;
                         \alts[\i].v = \tmp;
                         \alts[\i].c = \chan;
                    }} :: inits;
                    body = ‘C.stmt{
                         {
                         case \i:
                              \val = \tmp;
                              \b
                              break;
                         }
                    } :: body;
```

```
            case `{\chan <-= \val}:
                assert chan.type ~ `{chan(\t)};
                tmp := mktmp(t);
                inits = `C.stmt{{
                    \tmp = \val;
                    \alts[\i].op = CHANSND;
                    \alts[\i].v = \tmp;
                    \alts[\i].c = \chan;
                }} :: inits;
                body = `C.stmt{
                    {
                    case \i:
                        \b
                        break;
                    }
                } :: body;

            case _:
                util.panic("unexpected alt case");
            }
            i++;
        }
        inits = `C.stmt{ \alts[\i].op = CHANEND; } :: inits;
        return `C.stmt{
            {
                \inits
                switch(alt(\alts))
                    \body
            }
        }.compiled;
    }

    return default(term);
}
```

B.5. Alef task (thread creation) statement

```
// Alef task syntax

from xoc import *;
import util;

extend grammar C99
{
     stmt: "task" expr ";";
};

extend attribute
typecheck(term: C)
{
     if(term ~ ‘C.stmt{task \e;}){
          if(e.type == nil)
               return;
          if(e !~ ‘{\_(\_)}){
               errorast(e, "task statement requires"+
                    " function call expression");
               return;
          }
          return;
     }
     default(term);
}

ntramp := 0;
fn
trampname(): string
{
     return "__tramp"+string(ntramp++);
}

extend attribute
compiled(term: C.stmt): COutput.stmt
{
     args: list C.expr;
     if(term ~ ‘C.stmt{task \f(\(args));}){
          // Build up:
          //   * list of struct declarators
          //   * list of arguments referring to closure named "cl"
          // them from a closure instance named "cl".
          i := 0;
          decls := list [ ‘C.sudecl{\(f.type)* f;} ];
          clargs: list C.expr;
          for(a in args){
               n := "a"+string(i++);
               decls = ‘C.sudecl{\(a.type) \n;} :: decls;
               clargs = ‘C.expr{cl.\n} :: clargs;
          }
          decls = util.reverse(decls);
          clargs = util.reverse(clargs);

          // Create new struct type.
          t := ‘C.type{ struct { \decls } };

          // Create new trampoline function.
          tname := trampname();
          tramp := ‘C.fndef{
               void \tname(void *v) {
                    \t cl;
                    cl = *(\t*)v;
                    (free(v));
                    cl.f(\clargs);
               }
          };
          newfn(tramp);

          // Create variable to hold allocated closure
          // in expansion of task statement.
          // Note that this cl is not the same as the trampoline
          // function’s cl local variable.
          cl := Sym.Var("cl", term, ‘C.type{\t*}, ...);
          cl.islocal = true;

          // Create list of initializers filling in cl.
          // Cannot construct inits in the loop above,
          // because it needs cl, and cl needs t,
          // and the loop above creates the member list for t.
          inits: list C.stmt;
          for(a in args){
               n := "a"+string(i++);
               inits = ‘C.stmt{\cl->\n = \a;} :: inits;
          }
          inits = util.reverse(inits);

          // Expand task statement into closure alloc,
          // closure initialization, and call to taskcreate library fn.
          return ‘C.stmt{
               {
                    \cl = malloc(sizeof *cl);
                    \inits
                    taskcreate(\tname, \cl, 1<<20);
               }
          }.compiled;
     }
     return default(term);
}
```

B.6. Alef iterator expressions

```
// Alef-style iterator.
//
//   a[i=0::10] = 0;
//   a[0::10] = 0;
//
// both turn into
//
//   for(i=0; i<10; i++)
//        a[i] = 0;

from xoc import *;

extend grammar C99
{
     %right AlefIter;
     %priority Shift > AlefIter > Relational;

     expr: expr "::" expr [AlefIter];
}

struct Iterator
{
     var: C.expr;
     lo: C.expr;
     hi: C.expr;
};

extend attribute
type(term: C.expr): Type
{
     switch(term){
     case ‘{\a=\b ::\c}:
          if(a.type.isinteger
          && b.type.isinteger
          && c.type.isinteger)
               return a.type;
          errorast(term, "non-integer iterator");
     case ‘{\b ::\c}:
          //print b;
          if(b.type.isinteger
          && c.type.isinteger)
               return ‘Type{int};
          errorast(term, "non-integer iterator");
     }
     return default(term);
}

attribute
iterator(term: C.expr): Iterator
{
     switch(term){
```

```
	case `{\a=\b ::\c}:
		return Iterator(a, b, c);
	case `{\b ::\c}:
		sym := Sym.Var("i", term, `Type{int}, ...);
		sym.islocal = true;
		return Iterator(`C.expr{\sym}, b, c);
	}
	return nil;
}

attribute
iterators(term: C): list Iterator
{
	k := astsplit(term);
	out: list Iterator;
	for(i:=length(k)-1; i>=0; i--)
		out = k[i].iterators + out;
	if(term <: C.expr && term.iterator)
		out = term.iterator :: out;
	return out;
}

attribute
without_iterators(term: C): C
{
	if(term.iterators == nil)
		return term;
	if(term <: C.expr && term.iterator)
		return term.iterator.var;
	k := astsplit(term);
	for(i:=0; i<length(k); i++)
		k[i] = k[i].without_iterators;
	term = astjoin(term, k);
	return term;
}

extend attribute
compiled(term: C.stmt): COutput.stmt
{
	if(term.iterators == nil)
		return default(term);

	body1 := term.without_iterators;
	assert body1 <: C.stmt;
	body := body1;
	for(i in term.iterators)
		body = `C.stmt{
			for(\(i.var) = \(i.lo); \(i.var) < \(i.hi); \(i.var)++)
				\body
		};
	return body.compiled;
}
```

B.7. Sparse attribute syntax

```
from xoc import *;

extend grammar C99 {
     attr: /address_space/a "(" expr ")";
}

extend attribute
canonical(attr: C.attr): C.attr
{
     switch(attr){
     case ‘‘{address_space(0)}:
          return ‘‘C.attr{__no_attribute__};
     }
     return default(attr);
}
```

B.8. Sparse address space attributes

```
import sparse_grammar;
from xoc import *;
import sys;
import util;

attribute
address_space(t: C.type): int
{
     for(a in t.attrs){
          if(a ~ ‘C.attr{address_space(\e)}){
               if(e.constvalue ~ Constant.Integer(i))
                    return i;
               errorast(t, "address space is not constant integer");
               return 0;
          }
     }
     return 0;
}

attribute
has_address_space(t: C.type): bool
{
     for(a in t.attrs)
          if(a ~ ‘C.attr{address_space(\_)})
               return true;
     return false;
}
```

```
fn
no_address_space(a: C.attr): bool
{
     return a !~ ‘{address_space(\_)};
}

attribute
without_address_space(t: C.type): C.type
{
     tt := astcopy(t);
     tt.attrs = util.filter(no_address_space, t.attrs);
     return tt;
}

extend fn
canconvert(old: C.type, new: C.type): bool
{
     if(old ~ ‘{\told*} && new ~ ‘{\tnew*}){
          if(told.address_space != tnew.address_space){
               sys.werrstr("address space mismatch");
               return false;
          }
          if(told.has_address_space)
               return default(‘C.type{\(told.without_address_space)*}, new);
     }
     return default(old, new);
}

extend fn
cancast(old: C.type, new: C.type): C.type
{
     if(old ~ ‘{\told*} && new ~ ‘{\tnew*}){
          if(told.address_space != tnew.address_space){
               sys.werrstr("address space mismatch");
               return nil;
          }
     }
     return default(old, new);
}
```

B.9. Sparse noderef attribute

```
// Sparse no-deref attribute.

from xoc import *;

// No need to extend grammar: single words
// like noderef are already parsed as valid attributes.

fn
noderef(term: C.expr, e: C.expr)
{
     if(e.iscomputation
     && !term.isaddressof
     && e.type ~ ‘{\tt*}
     && hasattr(tt, ‘C.attr{noderef}))
          warnast(term, "deref of no-deref expr");
}

extend attribute
type(term: C.expr): C.type
{
     if((t := default(term)) == nil)
          return nil;

     if(term ~ ‘{*\e} || term ~ ‘{\e->\_})
          noderef(term, e);
     else if(term ~ ‘{\a[\b]}){
          noderef(term, a);
          noderef(term, b);
     }

     return t;
}
```

B.10. Sparse force casts

```
// Sparse force attribute in casts

from xoc import *;
import util;

fn
noforceattr(a: C.attr): bool
{
     return a !~ ‘{force};
}

attribute
without_force(t: C.type): C.type
{
     tt := astcopy(t);
     tt.attrs = util.filter(noforceattr, t.attrs);
     return tt;
}

extend fn
cancast(old: C.type, new: C.type): C.type
{
     if(old ~ ‘{\told*}
     && new ~ ‘{\tnew*}
     && hasattr(tnew, ‘C.attr{force}))
          return ‘C.type{\(tnew.without_force)*};
     return default(old, new);
}
```

B.11. Sparse context checking

```
// Sparse context attributes and __context__ statement.

from xoc import *;
import flow;

extend grammar C99 {
	attr: /context/a "(" expr "," expr "," expr ")";
	stmt: "__context__" "(" expr "," expr ")" ";";
}

// Type check __context__ statement.
// Actually ignore it.  Will check argument during compiled.
extend attribute
typecheck(term: C)
{
	if(term ~ `C.stmt{__context__(\_, \delta);})
		return;
	default(term);
}

// Compiled form of __context__ statement is an empty statement
// with a pre-initialized ctxdelta attribute.

attribute
ctxdelta(term: C): int
{
	if(term.copiedfrom)
		return term.copiedfrom.ctxdelta;
	return 0;
}

extend attribute
compiled(term: C.stmt): COutput.stmt
{
	if(term ~ `C.stmt{__context__(\_, \delta);}){
		out := `COutput.stmt{;};
		out.line = term.line;
		if(delta.constvalue ~ Constant.Integer(d))
			out.ctxdelta = d;
		else
			errorast(term, "context delta not integer constant");
		return out;
	}
	return default(term);
}
```

```
////////
// Data flow analysis of contexts.

// Context data structure, attached to each node in control flow graph.
struct Ctx
{
     inx: int;
     outx: int;
     have: bool;
};

attribute
ctx(term: C): Ctx
{
     return Ctx(...);
}

// Filter function - keep only interesting nodes,
// ones that have something to do with context.
fn
ctxfilter(term: C): bool
{
     return term.ctxdelta || term ~ ‘COutput.expr{\f(\_)};
}

// For debugging, return the string that should be
// used as the label on the edge a -> b.
fn
edgelabel(a: C, b: C): string
{
     s := string(a.ctx.outx);
     if(a.ctx.outx != b.ctx.inx)
          s = "!" + s;
     return s;
}

extend struct Sym
{
     havedelta: bool;
     delta: int;
}
```

```
fn
typectxdelta(t: C.type): (bool, int)
{
     if(t == nil)
          return (false, 0);
     if(t ~ ‘{\tt*})
          t = tt;
     for(a in t.attrs)
          if(a ~ ‘{context(\_, \down, \up)}){
               switch((down.constvalue, up.constvalue)){
               case (Integer(d), Integer(u)):
                    return (true, u - d);
               }
               errorast(t, "context delta not integer constant");
               return (true, 0);
          }
     return (false, 0);
}

fn
symctxdelta(sym: Sym.Var.Fn, isuse: bool): (bool, int)
{
     if(sym == nil)
          return (false, 0);

     if(!sym.havedelta){
          (have, delta) := typectxdelta(sym.type);
          if(have){
               sym.havedelta = true;
               sym.delta = delta;
          }
          else if(isuse || sym.storage !~ ‘{static}){
               sym.havedelta = true;
               sym.delta = 0;
          }
     }
     return (sym.havedelta, sym.delta);
}

fn
exprctxdelta(e: C.expr): (bool, int)
{
     if(e ~ ‘{\_::name}){
          sym := e.namesym;
          if(sym <: Sym.Var.Fn)
               return symctxdelta(sym, true);
     }
     return typectxdelta(e.type);
}
```

```
fn
dataflow(fun: Sym.Var.Fn, cfg: flow.Cfg): bool
{
     errors := 0;

     // Called on cfg.start: set initial context.
     fn init(a: C)
     {
          a.ctx.have = true;
          a.ctx.inx = 0;
     }

     // Called each time a node is visited:
     // compute the out context from the in context.
     // It is guaranteed that either init(a) or join(_, a)
     // has been called in the past.
     fn transfer(a: C)
     {
          // If this is a function call, get the context from the fn.
          delta: int;
          if(a ~ ‘C.expr{\f(\_)})
               (_, delta) = exprctxdelta(f);
          else
               delta = a.ctxdelta;
          a.ctx.outx = a.ctx.inx + delta;
     }

     // Called for each out arrow from a after visiting a:
     // check that if b has already been visited, the ctx
     // we had before matches the ctx coming from a.
     // Returns whether to visit b.
     fn join(a: C, b: C): bool
     {
          // If this is our first join of b, just record a’s out context.
          if(!b.ctx.have){
               b.ctx.have = true;
               b.ctx.inx = a.ctx.outx;
               return true;
          }

          // If we’ve been here before, the context from before
          // has to match the context we’re seeing now.
          // Only warn about the first error: when it rains, it pours.
          if(a.ctx.outx != b.ctx.inx && !errors++)
               warnast(b, fun.name+": context mismatch "+
                    string(a.ctx.outx)+" vs "+string(b.ctx.inx));
          return false;
     }

     d := flow.Dataflow(init, transfer, join);
     flow.dataflow(cfg, d);

     return !errors;
}
```

```
// Hook analysis into type-checking.
// After type-checking a function, run the sparse analysis
// on the compiled (standard C) form of the function body.
extend attribute
typecheck(term: C)
{
     default(term);
     if(term <: C.fndef){
          assert term ~ ‘{\_ \_ \body};
          cfg := flow.buildcfg(body.compiled, ctxfilter);

          // Set up for data flow analysis by initializing the
          // expected end-of-function context.  One wrinkle
          // is that sparse does not require annotations on
          // static functions that get inlined: the inliner runs before
          // sparse’s analysis, so those functions don’t actually
          // exist as far as sparse is concerned.  Since we don’t
          // have an inliner, we instead infer the appropriate
          // context annotation for static functions that have not
          // yet been used.
          fsym := term.function;
          (have, delta) := symctxdelta(fsym, false);
          cfg.end.ctx.have = have;
          cfg.end.ctx.inx = delta;

          // Run the data flow analysis.  If it encounters any
          // errors, it will return false and we can bail out.
          if(!dataflow(fsym, cfg)){
               if(false)
                    flow.showcfg("/tmp/a.sparse", cfg, edgelabel);
               return;
          }

          // If we didn’t set an annotation earlier, we need to
          // reconcile the fsym.delta and the delta we inferred
          // for cfg.end.
          if(!have && cfg.end.ctx.have){
               if(!fsym.havedelta){
                    fsym.havedelta = true;
                    fsym.delta = cfg.end.ctx.inx;
               }else{
                    if(fsym.delta != cfg.end.ctx.inx){
                         warnast(term, fsym.name+": context mismatch at end of function: "+
                              string(fsym.delta)+" vs "+string(cfg.end.ctx.inx));
                    }
               }
          }
     }
}
```

B.12. GNU typeof(expr) type specifier

```
from xoc import *;

extend grammar C99
{
     typeof: "typeof" "(" expr ")";
}

extend attribute
type(term: C.type): C.type
{
     switch(term){
     case ‘{typeof(\e::expr)}:
          if(e.type == nil)
               errorast(term, "no type here! "+string(e));
          return e.type;
     }
     return default(term);
}

extend attribute
compiled(term: C.type): COutput.type
{
     switch(term){
     case ‘{typeof(\e::expr)}:
          if(term.copiedfrom)
               return term.copiedfrom.compiled;
          if(e.type)
               return e.type.compiled;
          else{
               errorast(term, "no type! " + string(e));
               return ‘COutput.type{int};
          }
     }
     return default(term);
}

extend attribute
iscomputation(term: C.expr): bool
{
     if(term.parent ~ ‘C.typeof{typeof(\_::expr)})
          return false;
     return default(term);
}
```

B.13. GNU a ?: binary conditional expression

```
from xoc import *;

extend grammar C99
{
     expr: expr "?" ":" expr [Conditional];
};

extend attribute
type(term: C.expr): C.type
{
     if(term ~ ‘{\a ?: \b})
          return ‘C.expr{\a ? \a : \b}.type;
     return default(term);
}

extend attribute
compiled(term: C.expr): COutput.expr
{
     if(term ~ ‘{\a ?: \b}){
          tmp := mktmp(a.type);
          return ‘C.expr{
               (\tmp = \a) ? \tmp : \b
          }.compiled;
     }
     return default(term);
}
```

References

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

[2] Apple Computer, Inc.. *Dylan Reference Manual.* 1995.

[3] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language.* Prentice Hall, 2005.

[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, 2001.

[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, 2002.

[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, 1999.

[7] Thomas Anthony Bergan. Typmix: A Framework For Implementing Modular, Extensible Type Systems. Master's thesis, University of California Los Angeles, 2007.

[8] Computer History Museum. Fellow Awards | 1997 Recipient Dennis Ritchie. Available online at `http://www.computerhistory.org/fellowawards/index.php?id=71` as of August 2008.

[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, Orlando, Florida, United States, 1991.

[10] Coverity. Linuxbugs. `http://linuxbugs.coverity.com` Offline as of 2006. 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, 1986.

[12] Glen Ditchfield. An Overview of Cforall. Available online at `http://plg.uwaterloo.ca/~cforall/` as of August 2008.

[13] Eelco Dolstra and Eelco Visser. Building Interpreters with Rewriting Strategies. *Proceedings of the 2002 Workshop on Language Descriptions, Tools, and Applications*, Grenoble, France, 2002.

[14] Gabriel Dos Reis and Bjarne Stroustroup. Specifying C++ concepts. *Proceedings of the 2006 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, 1992.

[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, Canada, 2001.

[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*, 1995.

[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, January 2004.

[20] James Gosling. Ace: a syntax-driven C preprocessor, 1989. Available online at `http://swtch.com/gosling89ace.pdf` as of August 2008.

[21] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice-Hall, 1993.

[22] Michael Hammer. An Alternative Approach to Macro Processing. *Proceedings of the International Symposium on Extensible Languages*, pages 58-64, Grenoble, France, 1971.

[23] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, 1996.

[24] David R. Hanson and Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[25] David R. Hanson and Jeffrey L. Korn. A Simple and Extensible Graphical Debugger. *Proceedings of the 1997 USENIX Technical Conference*, 1997.

[26] Timothy P. Hart. MACRO Definitions for LISP. MIT AI Project—RLE and MIT Computation Center AI Memo 57. Cambridge, MA, October 1973. Reproduced in [56].

[27] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North-Holland/Elsevier, 1977.

[28] Martin Hirzel and Robert Grimm. Jeannie: Granting Java native interface developers their wishes. *Proceedings of the 2007 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 19-38, Montreal, Canada, October 2007.

[29] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM* 21(8), pages 666-677, August 1978.

[30] Gerard J. Holzmann. Static source code checking for user-defined properties. *Proceedings of the 6th World Conference on Integrated Design & Process Technology*, Pasadena, California, United States, June 2002.

[31] Stephen C. Johnson. YACC: Yet Another Compiler Compiler. *Unix Programmer's Manual, Volume 2*, Murray Hill, NJ, 1979.

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, 2nd Edition*. Prentice Hall, 1988.

[33] Donald E. Knuth. Semantics of context free languages. *Mathematical Systems Theory* 2(2), pages 127-146, June 1968.

[34] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*, pages 151-161, Cambridge, Massachusetts, United States, 1986.

[35] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems* 18(3), pages 263-297, August 2000.

[36] Eddie Kohler, Benjie Chen, M. Frans Kaashoek, Robert Morris, and Massimiliano Poletto. Programming Language Techniques for Modular Router Configurations. MIT Laboratory for Computer Science Technical Report MIT-LCS-TR-812. Cambridge, MA, August 2000.

[37] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events Can Make Sense. *Proceedings of the 2007 USENIX Technical Conference*, Santa Clara, CA, 2007.

[38] Donis Marshall. *Programming Microsoft Visual C# 2008: The Language*. Microsoft Press, 2008.

[39] David Mazières. A Toolkit for User-Level File Systems. *Proceedings of the 2001 USENIX Technical Conference*, pages 261-274, Boston, MA, June 2001.

[40] M. Douglas McIlroy. Macro Instruction Extensions of Compiler Languages. *Communications of the ACM* 3(4), pages 214-220, April 1960.

[41] Scott McPeak. Elkhound: A Fast, Practical GLR Parser Generator. University of California Berkeley Technical Report UCB/CSD-2-1214. Berkeley, CA, December 2002.

[42] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[43] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *Proceedings of the 11th International Conference on Compiler Construction*, 2002.

[44] Nathaniel Nystrom, Michael Clarkson, and Andrew Myers. Polyglot: an Extensible Compiler Framework for Java. *Proceedings of the 12th International Conference on Compiler Construction*, April 2003.

[45] Nathaniel Nystrom, Xin Qi, and Andrew Myers. J&: Software Composition with Nested Intersection. *Proceedings of the 2006 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21-36, 2006.

[46] Chris Okasaki. Red-Black Trees in a Functional Setting. *Journal of Functional Programming* 9(4), pages 471-477, July 1999.

[47] Rob Pike. The Implementation of Newsqueak. *Software—Practice and Experience* 20(7), pages 649-659, July 1990.

[48] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems* 8(3), pages 221-254, 1995.

[49] *Plan 9 from Bell Labs, Second Edition*. Harcourt Brace & Company, 1995. The source code in the introduction is from the file `/sys/src/cmd/acme/scrl.l`.

[50] Plan 9 from Bell Labs, Fourth Edition. Available online at `http://plan9.bell-labs.com/plan9/` as of August 2008. The source code in the introduction is from the file `/sys/src/cmd/acme/scrl.c`.

[51] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A Nanopass Infrastructure for Compiler Education. *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, Snowbird, Utah, United States, September 2004.

[52] Elizabeth Scott and Adrian Johnstone. Right-Nulled GLR Parsers. *ACM Transactions on Programming Languages and Systems* 28(4), July 2006.

[53] Doc Searls. Linus and the Lunatics, Part I. *Linux Journal*, 2003.

[54] Peter Seebach. Infrequently Asked Questions in comp.lang.c.

[55] Sparse – a Semantic Parser for C. `http://kernel.org/pub/software/devel/sparse/`

[56] Guy L. Steele, Jr. and Richard P. Gabriel. The Evolution of Lisp. *ACM SIGPLAN Notices* 28(3), pages 231-270, 1993.

[57] Bjarne Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1997.

[58] Bjarne Stroustrup. A Rationale for Semantically Enhanced Library Languages. *Proceedings of Library-Centric Software Design (LCSD'05)*, 2005 (co-located with OOPSLA 2005).

[59] A. S. Tanenbaum. A Tutorial on ALGOL 68. *Computing Surveys*, pages 155-190, June 1976.

[60] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. *Proceedings of Compiler Construction 2001*, pages 365-370.

[61] Eric Van Wyk, Derek Bodin, Lijesh Krishnan, and J. Gao. Silver: an extensible attribute grammar system. *Proceedings of the 2007 Workshop on Language Descriptions, Tools, and Applications*, Braga, Portugal, 2007.

[62] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for Java. *Proceedings of 21st European Conference on Object-Oriented Programming*, Berlin, Germany, August 2007.

[63] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, United States, September 1998.

[64] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. Institute of Information and Computing Sciences, Utrecht University Technical Report UU-CS-2004-011. 2004.

[65] Scott A. Vorthman. Coordinated Incremental Attribute Evaluation on a DR-Threaded Tree. *Attribute Grammars and Their Applications*, 1990.

[66] Daniel Weise and Roger Crew. Programmable Syntax Macros. *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pages 156-165, 1993.

[67] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices* 29(12), pages 31-37, 1994.

[68] Phil Winterbottom. Acid: a debugger built from a language. *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 211-222, 1994.

[69] Phil Winterbottom. Alef Language Reference Manual. *Plan 9 Programmer's Manual: Volume Two*, 1995.