Xoc, an Extension-Oriented Compiler for Systems Programming

Russ Cox* Tom Bergan† Austin T. Clements* Frans Kaashoek* Eddie Kohler†
MIT CSAIL* UCLA CS†

Abstract

Today's system programmers go to great lengths to extend the languages in which they program. For instance, system-specific compilers find errors in Linux and other systems, and add support for specialized control flow to Qt and event-based programs. These compilers are difficult to build and cannot always understand each other's language changes. However, they can greatly improve code understandability and correctness, advantages that should be accessible to all programmers.

We describe an *extension-oriented compiler* for C called xoc. An extension-oriented compiler, unlike a conventional extensible compiler, implements new features via *many* small extensions that are loaded together as needed. Xoc gives extension writers full control over program syntax and semantics while hiding many compiler internals. Xoc programmers concisely define powerful compiler extensions that, by construction, can be combined; even some parts of the base compiler, such as GNU C compatibility, are structured as extensions.

Xoc is based on two key interfaces. Syntax patterns allow extension writers to manipulate language fragments using concrete syntax. Lazy computation of attributes allows extension writers to use the results of analyses by other extensions or the core without needing to worry about pass scheduling.

Extensions built using xoc include `xsparse`, a 345-line extension that mimics Sparse, Linux's C front end, and `xlambda`, a 170-line extension that adds function expressions to C. An evaluation of xoc using these and 13 other extensions shows that xoc extensions are typically more concise than equivalent extensions written for conventional extensible compilers and that it is possible to compose extensions.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Design, Experimentation

Keywords extension-oriented compilers

1. Introduction

Programmers have long found it useful to make domain-specific changes to general-purpose programming languages. Recent examples include Sparse (Torvalds and Triplett 2007), Tame (Krohn et al. 2007), and Mace (Killian et al. 2007). The most common implementation is a monolithic preprocessor: a compiler front end that accepts the base language plus domain-specific changes, compiles the changes into base language constructs, writes out the equivalent base program, and invokes the original compiler or interpreter. The most serious problem with this approach is that extensions are heavyweight and isolated. Preprocessors must include entire front ends for the base language itself, and multiple extensions are difficult to use together.

Recent extensible compiler research (e.g., Necula et al. 2002; Nystrom et al. 2003; Grimm 2006; Visser 2004) aims to solve this problem. Starting with an extensible compiler for the base language, a programmer writes only the code necessary to process the domain-specific changes. A typical implementation is a collection of Java classes that provide a front end toolkit for the base language. To extend such compilers, a programmer subclasses some or all of the classes, producing a *new* compiler that accepts the base language with the domain-specific changes. This approach reduces implementation effort but still creates compilers that are not *composable*: changes implemented by two different compilers can't be used in the same program since the compilers can't understand and process one another's input. Recent extensible compilers have some support for extension composition (Van Wyk et al. 2007a; Nystrom et al. 2006), but still require that extension writers explicitly assemble each desired composition.

This paper proposes an *extension-oriented compiler* where new features are implemented by many small extensions that are loaded on a per-source-file basis, much as content-specific plugins are loaded by web browsers and other software. As in current extensible compilers, this approach starts with a compiler for the base language, but extensions do not create whole new compilers. Instead, the base compiler loads the extensions dynamically for each source file.

A key challenge in an extension-oriented compiler is to enforce the boundaries between the different extensions and the base compiler, while still allowing extensions the freedom and power to alter language semantics. We meet this challenge with two key interfaces. First, extension writers use *syntax patterns* (Bravenboer and Visser 2004) to manipulate language fragments and abstract syntax trees (ASTs) using the concrete syntax of the programming language being compiled. This hides the details of both parsing and internal representation from extension modules. Second, extension writers implement most program analyses via *on-demand (lazy) AST attributes*. This hides the details of pass scheduling from extensions; the passes exist only implicitly in the dependencies between the various attributes. These ideas originated with other systems, but combined, and augmented by other design choices—including *extensible functions*, *extensible data structures*, and a *GLR-based parser* with modular grammars—their value is multiplied. The result seems to us qualitatively better for extension programming than systems based on a single interface, such as term rewriting or Java subclassing.

We have designed and implemented *xoc*, an extension-oriented compiler for C, that provides these interfaces. The compiler itself is structured as a core with a set of extensions. The compiler totals 32,062 lines of code and is complete enough that it can process the source files of the Linux kernel.

ASPLOS '08 March 1–5, 2008, Seattle, Washington, USA.

Reprinted from ASPLOS '08, *Proc. 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 1–5, 2008, Seattle, Washington, USA., pp. 244–254.

```
grammar XRotate extends C99
{
    expr: expr "<<<" expr  [Shift]
        | expr ">>>" expr  [Shift];
}

extend attribute
type(term: ptr C.expr): ptr Type
{
    switch(term){
    case ~expr{\a <<< \b} || ~expr{\a >>> \b}:
        if(a.type.isinteger && b.type.isinteger)
            return promoteunary(a.type);
        error(term.line, "non-integer rotate");
        return nil;
    }
    return default(term);
}
```

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

Figure 1. The `xrotate` extension. Note that the `x` and `y` variables introduced in `compiled` don't cause variable capture problems; see section 2.3.

Using xoc we have designed and implemented 15 extensions to date. These range from trivial extensions, such as a bitwise rotate operator, to extensions for which systems programmers have implemented domain-specific front ends, such as Sparse (a source code checker for the Linux kernel). Based on these extensions we conclude that programmer effort for an extension is relatively small and scales with the complexity of the extension. Most extensions are short, and the ones that are larger are large because the extension is more complicated. For example, the rotate extension is 34 lines, while the Sparse extension is 345 lines. We also conclude that the extension writer need not understand much of xoc's internals; xoc's extension interface consists of relatively few grammar rules and attributes, only a few of which need to be modified for a typical extension, and extensions can manipulate xoc's internal program representation naturally using concrete syntax. Finally, we conclude that extensions can compose: an extension writer can supply several extensions to the compiler at the same time and use the combination in a program or in another extension.

This paper identifies extension-oriented compilation as a research problem and makes the following contributions towards solving that problem:

1. The use of syntax patterns to manipulate the abstract syntax tree and attributes, and lazy computation of AST attributes to eliminate explicit scheduling of compilation passes.
2. The design of xoc, a prototype extension-oriented compiler for the C language, which implements syntax patterns and lazy computation of AST attributes, as well as a GLR parser and extensible functions and data structures.
3. A prototype implementation of xoc in zeta, a C-like interpreted procedural language with first-class functions. Zeta makes writing extensions easier than standard C, although extension writers must adjust to a slightly different language. Our implementation runs zeta using a bytecode interpreter and is therefore limited in its performance; we plan to replace the bytecode interpreter with compilation to machine code, which should reduce the compilation time for a large program from tens of seconds to a fraction of a second.
4. An evaluation of xoc and 15 currently implemented extensions. Some of the extensions support features that others have implemented as separate front ends for C, allowing for a comparison of the extension-oriented compiler and front-end approaches. We implemented one extension (bitwise rotate) in multiple extensible compilers, allowing a comparison with our approach.

After presenting the xoc extension interface, we describe its implementation, present several extension case studies, and examine our extensions as a group. We then discuss related work and conclude.

2. The Xoc Extension Interface

The xoc prototype is a source-to-source translator. It reads C programs that might use extensions, analyzes the programs, compiles them into equivalent standard C code, and then invokes gcc to create an object file. If extensions are specified on the command line, xoc dynamically loads those extensions before processing the possibly-extended input files. This section describes how extensions are written by presenting the interface xoc provides to extension writers.

Because the xoc prototype focuses on the front end—back-end extensions are left for future work—its primary data structure is the internal representation of the input program. Xoc's interface for this data structure must support parsing, constructors, accessors, and analysis passes. It is important to make these fundamental compiler manipulations as easy as possible. Xoc achieves this simplification by hiding parsing details behind context-free grammars, hiding program manipulation behind concrete syntax patterns, and hiding analysis scheduling behind lazily-computed attributes.

As a running example throughout this section, we will consider an extension that adds bitwise rotate operators <<< and >>> to the language. The operators behave like the standard << and >> bit shift operators except that bits shifted off the end of the 32- or 64-bit word are shifted back in at the other end. Figure 1 gives the extension's full source code. Even this trivial extension still illustrates the difficult issues inherent to extension-oriented compiler design. For example, the extension's grammar statements add <<< and >>> expressions to the language, but in the context of a source file, other extensions may have added other expression types as well. An expression like "`(a <? (b <<< c)) <<< 4`" (`<?` is gcc's minimum operator) forces the `xrotate` extension to analyze, compute with, and generate code for an operand that uses an extension unknown to `xrotate`'s author; the `xgnu-minmax` extension that implements `<?` must do the same. Our work has concentrated on finding simple interfaces that make extension composition natural, simple, and robust to errors.

2.1 Grammars and parsing

Xoc provides special syntax—a `grammar` statement—for defining and extending the context-free grammar rules used to parse the input. There are three main subproblems: providing a definition of C that is easily extended, keeping different extension grammars separate, and handling ambiguity (inputs that can be parsed multiple ways).

Extending grammars Xoc must provide a base C grammar that is easy for extension writers to understand and to extend. Xoc's base C grammar has 72 symbols, but few are relevant to a typical extension writer (for example, 13 are dedicated to distinguishing between numeric constant formats). Knowing only `expr` (expressions), `stmt` (statements), and a few others relating to C type declarators is surprisingly effective; a working vocabulary of under a dozen symbols suffices for the vast majority of extensions, as discussed in section 4.

Xoc defines a base grammar for standard C:

```
extensible grammar C99 {
    ...
    %left Add Shift
    %precedence Add > Shift
    ...

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

Extensions add new grammar rules with an `extend grammar` statement. For example, the rotate extension defines its new operators using:

```
grammar XRotate extends C99
{
    expr: expr "<<<" expr    [Shift]
        | expr ">>>" expr    [Shift];
}
```

Unlike in parser generators such as yacc, the grammar rules do not specify parsing actions. Xoc's only parsing action is to build a generic parse tree. The interface for manipulating the parse tree is discussed in section 2.2. Not requiring the extension writer to provide explicit actions makes it easier to define new syntax.

This example also illustrates xoc's use of precedence to collapse the C standard's many expression types (`additive-expr`, `multiplicative-expr`, and so on) into just `expr`. The bracketed `[Shift]` tags specify named precedence levels: the new rotate operators have the same precedence as the shift operators.

An extension can also introduce a new precedence level between existing ones; for example, `xalef-iter` introduces Alef's iteration operator `::` with precedence between << and <= (Winterbottom 1995).

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

Keeping grammars separate The `grammar` declarations above define one grammar, `C99`, and two extensions to `C99`: `XRotate` and `XAlefIter`. Since the latter two come from independent extensions, it is important to be clear which grammars are being used in which contexts. After the above definitions, xoc distinguishes between four different possible grammars: `C99`, `C99+XRotate`, `C99+XAlefIter`, and `C99+XRotate+XAlefIter` (`C99+XAlefIter+XRotate` is the same grammar). When parsing the input, xoc uses `C99` plus all loaded extensions, but other parts of the compiler may expect only certain kinds of input syntax. Xoc's type system helps enforce these expectations.

Xoc's type system can express that a particular AST corresponds to a particular non-terminal, as in `C99.expr`. Distinguishing different kinds of ASTs is useful for defining what kind of syntax a function expects. For example, a function that analyzes only standard C99 expressions could take a `C99.expr`, while a function that also allows rotate expressions could take `(C99+XRotate).expr`. These types are defined statically, but subtype instances can be checked dynamically: for example, xoc lets extensions check whether a value that is statically typed as `C99+XRotate` is actually pure `C99`. Finally, the type system also allows a "wildcard" grammar, as in `C99+?`. A wildcard represents an arbitrary set of grammar extensions that is not known statically. This is how xoc shares syntax trees between extensions that have no knowledge of each other's syntax.

Ambiguity Typically, there will be only one possible parse tree for a given input, but the possibility of ambiguity—multiple parse trees—is unavoidable when using a context-free grammar as the input specification. Others have proposed using parsing expression grammars (Ford 2004), which replace the context-free alternation operator with an ordered-choice operator so that ambiguity is impossible. This approach essentially resolves ambiguities quietly; we prefer to treat ambiguity as a signal that the programmer might not be getting the expected result and to respond with a red flag. If there are multiple ways to parse an input string, xoc will discover all possible parses and report an error.

Detecting ambiguity is particularly important when using independently written extensions that might define different meanings for the same syntax. For example, if using rotate and a "be like Java" extension, a programmer might not realize that there are two definitions for >>> (rotate or Java's unsigned right shift). We'd prefer that extensions loaded in different orders produce the same result, so it is important to detect the ambiguity, rather than silently choosing one or the other.

Checking context-free grammars for ambiguity is uncomputable (Lewis and Papadimitriou 1997, pp. 259–261), so xoc settles for detecting the ambiguity when it arises in the input. In the example just given, a conflict will be reported only if the errant developer uses the >>> operator in his program. We have not had many problems with ambiguity so far; future experience may encourage the use of heuristics to detect obvious extension conflicts statically.

2.2 Syntax patterns

Once an input program has been parsed, xoc and its extensions need to traverse the AST, computing its properties and often rewriting it into different forms. Rather than expose the AST using traditional data structures, xoc extensions refer to the AST using concrete syntax (the syntax of the programming language).

For example, the expression `ast ~ expr{\a <<< \b}` evaluates true if `ast` is an AST node generated by the first rule in the `XRotate` grammar above. When this *destructuring* expression evaluates true, it binds the variables `a` and `b` to the AST nodes for the subexpressions. A similar syntax creates new ASTs. If `a` and `b` are already ASTs corresponding to expressions, then the *restructuring* expression `` `expr{\a <<< \b} `` constructs a new AST representing the <<< syntax. Like Lisp, xoc unifies internal and external program representation. Whereas Lisp makes programs look like an internal data structure, xoc makes the internal data structure look like a program.

Syntax patterns can be arbitrarily complex program fragments, not just single rule applications. For example, the following snippet rewrites repeated rotation into an equivalent single rotation:

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

Section 3 discusses how the grammar symbols of `a`, `b`, and `c` are inferred.

The examples so far have assumed that `ast` was declared as `C99+XRotate`, so that the `<<<` operator is available for use in the destructuring and restructuring patterns. When the grammar type includes `+?`, syntax from any extension can appear in slots like `\a` and `\b`, but the concrete syntax used in the pattern is restricted to the named grammars. Consider these three lines:

```
1  C99.expr{\a <<< \b};      // invalid!
2  (C99+XRotate).expr{\a <<< \b};
3  (C99+XRotate+?).expr{\a <<< \b};
```

Line 1 is invalid in xoc (xoc will print an error when loading the extension), because there are no rules involving `<<<` in the `C99` grammar. Lines 2 and 3 are valid but have different meanings. Suppose an extension that introduced a `**` exponentiation operator were also loaded. Line 2 would not allow exponentiation expressions within `\a` and `\b`, but line 3 would.

Xoc provides a few mechanisms to make syntax patterns even more convenient. By convention, xoc defines `C` as `C99` plus all extensions currently in scope plus `?`. Also, xoc uses static type information to infer the grammar and symbol name in syntax patterns. If the static type information is sufficient, explicit names can be omitted. For example, if `ast` is declared as `C.expr`, then one can write `ast ~ {\a <<< \b}` instead of `ast ~ expr{\a <<< \b}`. Also, conversion routines can be registered to convert automatically between non-AST types and AST nodes, allowing the use of zeta values like integers and strings directly in slots.

C's type syntax does not always align with the top-down structure of the underlying types. Even so, xoc arranges for special processing of the syntax patterns before they are used in destructuring and restructuring, so that expressions like `t ~ {\tt*}` and `t ~ {\tt(*)(void*, int)}` work as expected.

2.3 Lazy attributes

In addition to traversing and rewriting the program, compilers analyze the program to determine how to compile it. Many compilers are structured as a series of passes: first variable scoping, then type checking, then constant evaluation, and so on. Extensions may need to use the results of some of these analyses or even introduce their own. Other extensible compilers expose this pass-based structure to extensions, requiring them to declare how they fit into the existing compiler passes. Inspired in part by attribute grammars (Knuth 1968; Paakki 1995; Van Wyk et al. 2007a,b), xoc eliminates passes and pass scheduling by representing analyses as lazily-computed attributes.

Xoc's attributes are *not* attribute grammars: there is no strict enforcement that attribute computations proceed in a particular order, and attribute computations are free to examine the rest of the AST without using the attribute framework.

Attributes are referred to using structure member notation, as in `a.type`, but they are computed on demand at first access and then cached. (Caching the value is safe because ASTs are immutable.)

Each attribute is defined by the ordinary xoc code that computes it. A key feature is that extensions can change the behavior of this code. For example, the `type` attribute, which specifies the type of an expression, is defined in the xoc core as:

```
attribute
type(term: ptr C.expr): ptr Type
{
    switch(term){
    case ~{\a << \b} || ~{\a >> \b}:
        if(a.type.isinteger && b.type.isinteger)
            return promoteunary(a.type);
        error(term.line, "non-integer shift");
        return nil;
    // ... other cases ...
    }
    error(term.line, "cannot type check expression");
    return nil;
}
```

The rotate extension extends the `type` attribute with support for rotations:

```
extend attribute
type(term: ptr C.expr): ptr Type
{
    switch(term){
    case ~{\a <<< \b} || ~{\a >>> \b}:
        if(a.type.isinteger && b.type.isinteger)
            return promoteunary(a.type);
        error(term.line, "non-integer rotate");
        return nil;
    }
    return default(term);
}
```

Inside an `extend attribute` body, the name `default` refers to the definition of the attribute before it was extended. The rotate extension checks for and handles the rotate cases, but leaves all other behavior unchanged by deferring to `default`. If different xoc extensions extend the same attribute, the extensions are chained together via these `default` calls.

The `type` attribute proceeds top-down, but the full power of lazy attributes is realized when attributes are not purely top-down. For example, the set of variables in scope on entry to each AST node depends on nodes to the left in the abstract syntax tree. To help compute this, xoc provides builtin attributes `parent`, `prev`, and `next`, which are a node's parent, left sibling, and right sibling in the AST. These attributes can be used to define any traversal order, which is expanded lazily as it is used. Because of this design, the variable scope at any node can be queried at any time.

Compiler Structure Xoc is implemented using a library of attributes that extensions use and sometimes augment. The attributes described here make up the core of the compiler. The dependencies implicit among xoc's attributes determine the complete compiler structure.

After parsing the initial input to produce an AST `root`, xoc looks at the root's `welltyped` attribute, a boolean specifying whether the entire tree is free of type errors. Extensions that introduce new statements extend `welltyped` to check those statements. For expressions, the `type` attribute (shown above) evaluates to the C type of the expression—`int`, `void*`, and so on—or null if the expression contains type errors. Type checking depends on variable scoping, which is provided by `vars` and `vars_out`, the sets of variables in scope at entry to and exit from an AST node.

If `root.welltyped` is true, xoc uses `root.vars_out` as the list of globals defined in the program. Each global's `compiled` attribute holds the equivalent standard C representation for that global. This representation uses grammar `COutput`, a synonym for `C99+XGnuStmtExpr`. (Allowing the GNU statement expression simplifies the compilation of expressions with side effects.) Xoc passes these `compiled` attributes to the C output code. Finally xoc invokes gcc on the C output.

Variable Capture Any program rewriting system must worry about unintended variable capture in expanded code: if a reference to a variable `x` is copied into a new AST, it should by default continue to refer to the same variable, even if there is a different definition of `x` in scope at the new location. Avoiding unintended variable capture in expansions is called *hygiene* (Kohlbecker et al. 1986). To enforce hygiene, xoc provides a built-in `copiedfrom` attribute that links a copied AST node to the node it was copied from. To compute `sym`, the attribute for the symbol associated with a variable name reference, xoc uses `copiedfrom.sym` when `copiedfrom` is not null, which guarantees that the meaning of names will not change, even when moved between scopes. For example, in the rotate extension (Figure 1), the introduction of the temporaries `x` and `y` does not cause any problems even if the fragments `\a` and `\b` refer to other variables named `x` or `y`. The C output library renames variables appropriately when two distinct variables share a name.

2.4 Other interfaces

Several secondary Xoc interfaces, including extensible data structures, extensible functions, higher-order tree rewriters, and a control flow analysis module, enhance the power of the main extension interfaces.

First, data structures can be marked extensible, meaning that extensions can define additional fields that are looked up in a per-object hash table but still use field access syntax (`a.field`). These extensions are lexically scoped so that different extensions cannot accidentally see each other's fields (Warth et al. 2006). Functions can be marked extensible in the same style as attribute computations; a few xoc functions involve more than just a single AST and must be written this way. For example, `xcanconvert` determines whether it is possible to convert one type implicitly into another. An extension might relax the rules involving character pointers using:

```
extend fn
xcanconvert(from: ptr Type, to: ptr Type): bool
{
    if((from ~ {unsigned char*} && to ~ {char*})
    || (from ~ {char*} && to ~ {unsigned char*}))
        return true;
    return default(from, to);
}
```

Extensible functions are similar to advice in Lisp (Teitelman 1966) and in aspect-oriented programming (Kiczales et al. 1997). The combination of extensible data structures and extensible functions is sufficient to implement attributes, but attributes are so fundamental in xoc that it makes sense to give them their own syntax.

To allow generic traversal of any abstract syntax (usually, to look for a particular type of node), the xoc function `astsplit` returns an array containing an AST's child nodes. To allow generic rewriters, the xoc function `astjoin` performs the inverse operation, building a new AST node given an example and a replacement set of children. Xoc also abstracts this interface into a collection of higher-order tree rewriters, similar to Stratego's strategies (Visser 2004).

Xoc also provides a traditional library for computing control flow graphs.

3. The Xoc Implementation

This section describes our current implementation of xoc, which is the third prototype we've built. This xoc is written in a custom language we call zeta. Zeta is a procedural language with first-class functions. A bytecode interpreter, also called zeta, compiles and runs zeta programs. Zeta is a typical procedural language but augmented with the xoc-specific features discussed in the previous section. Figure 2 gives line counts for zeta and xoc. Of the zeta code making up xoc, half is devoted to type checking C and a quarter to printing C output for gcc.

1,083	C	Utility routines (hash tables, lists, etc.)
1,727	C	GLR parser
1,259	C	DFA-based lexer
14,316	C	Zeta interpreter core
2,955	C	AST support for Zeta
7,707	Zeta	Xoc
3,015	Zeta	Extensions
32,062		*Total*

Figure 2. Line counts for xoc source code.

We wrote the earlier two prototypes directly in extended C, using xoc to compile itself, but bootstrapping xoc in itself made it difficult to experiment with new extensibility features. Using zeta has made it much easier to try (and discard) features. Once xoc is more mature, we are interested in revisiting the question of implementing xoc using itself, but we have found zeta to be a fruitful base for experimenting with extensible compiler structure.

3.1 Grammars and ambiguity

Zeta implements the `grammar` statement using a GLR parser. The C99 grammar is 328 lines and 72 symbols. GLR parsers handle ambiguity—multiple ways to parse a given input—by returning all possible parses, letting the caller choose the correct one. Using a GLR parser allows checking for ambiguity. It also handles C's type-vs.-name ambiguity without tightly coupling the lexer and the type checker, giving considerably more freedom in the design of the compiler. Finally, the GLR parser can accommodate any possible context-free grammar, unlike traditional LR parsers with fixed lookahead (including LALR(1) parsers like yacc). These more restricted language classes are not closed under union, meaning that some extensions would work fine in isolation but cause unnecessary parsing conflicts when used together. In contrast, it is simply not possible to write a context-free grammar that a GLR parser will be unable to handle.

When the parser returns multiple parse trees, it is due either to C's name vs. type ambiguity or to ambiguity introduced by extensions. Xoc invokes a disambiguation function to resolve the name vs. type ambiguities. Any ambiguities that remain are considered errors: xoc prints information about them and exits. Disambiguation is intentionally not extensible; we believe that C's inherent ambiguity is unfortunate enough and do not encourage extension writers to introduce new ambiguous language constructs.

3.2 Abstract syntax

The internal abstract syntax representation is generic: each node is either an `AstRule`, which contains a pointer to the grammar rule and pointers to the appropriate number of children for that rule; an `AstString`, which is a lexical token; or an `AstSlot`, which is a slot like `\a` in a syntax pattern.

AST types (e.g., `(C99+XRotate).expr`) are a property of the entire subtree rooted at that node, and xoc programs need to be able to check whether an AST is an instance of a given grammar type at run-time. This may sound like an expensive run-time check, but since ASTs are immutable, the most specific grammar type can be computed incrementally at AST construction time, making the check inexpensive.

3.3 Syntax patterns

Syntax patterns are implemented by parsing them using the GLR parser, creating a template used for destructuring or restructuring.[1]

Previous systems (e.g., Bachrach and Playford 2001; Weise and Crew 1993) have proposed similar use of syntax patterns, especially for restructuring, but they end up less convenient, because the extension writer has to annotate every slot with its grammar symbol, as in `expr{\a::expr <<< \b::expr}`. These annotations can quickly obscure the convenience of writing in direct syntax.

Since xoc does not require such annotations, the first problem in handling syntax patterns is deciding their meaning. Since input programs may be ambiguous, so can destructuring and restructuring patterns. Worse, since destructuring and restructuring pattern variables can match arbitrary grammar symbols instead of just tokens, even patterns like `expr{-\a}` are ambiguous: perhaps `\a` is meant to stand in for a numeric constant, or perhaps a variable name, or perhaps just an arbitrary expression. The ambiguity of `-\a` could be eliminated if every variable like `\a` were tagged with its grammar symbol as in other systems, but doing so loses much of the brevity and convenience of the destructuring and restructuring patterns. Xoc decides the meaning of syntax patterns in two different ways.

First, AST subtyping like `C.expr` and `C.name` allow xoc to determine the kind of slot by looking at the type of expression being substituted into that slot. This is most often helpful in restructuring.

Second, when the type of the slot is not uniquely specified, xoc can specify as much as it knows (e.g., "this is either a `number` or a `name`"; or "this is some kind of list"; or "this could be anything") and let the GLR parser try all those possibilities and find out which work. If many work, xoc chooses the shallowest parse tree. For example, `expr{\a <<< \b}` treats `\a` and `\b` as `expr`s. Another valid parse would be to treat them as `number`s, converting them into `expr` using the "`expr: number`" grammar rule, but that would result in a deeper parse tree. We have found that the shallowest parse is almost always the one we mean; when it is not, xoc will give a type error and the extension writer can add an explicit annotation like `\a::number`.

4. Case Studies and Evaluation

In order to evaluate xoc, we implemented a variety of extensions, ranging from trivial extensions such as `xrotate` from section 2 to complex extensions such as `xlambda`, which adds first-class closures, and `xsparse`, which implements the same analysis as the Linux kernel checker, Sparse. Figure 3 summarizes the extensions.

The second column in Figure 3 gives each extension's line count. As we hoped, simple extensions require few lines of code, while implementations of more complex extensions seem proportional to their complexity. In addition, we would like to show how xoc's extension interfaces simplify the task of writing extensions compared to the front-end approach and the extensible-compiler approach.

We describe four extensions in detail: `xsparse`, an analysis extension, implemented in xoc to mimic Sparse; `xvault`, another analysis extension, implemented in xoc and compared against a Polyglot version; `xrotate`, a simple rewriting extension, implemented in xoc, CIL, xtc, and Polyglot; and `xlambda`, a more complex rewriting extension, implemented in xoc and xtc. Finally, we present a few statistics and observations about the extensions as a group.

[1] The idea of parsing a code pattern once and then saving the parsed representation for repeated use was first proposed by Hammer (Hammer 1971), who suggested it as an alternative to purely lexical implementations of syntax macros.

4.1 Sparse, the Linux kernel checker

Sparse (Torvalds and Triplett 2007) is a source code checker for the Linux kernel. Sparse checks for violations of Linux coding style and extends the C type system with annotations for two important notions that are applicable beyond the Linux kernel.

First, Sparse adds *address spaces* to C pointer types. Unannotated pointers are considered kernel pointers, which marks them as `address_space(0)` and permits dereferencing. Annotating a pointer with `__user` marks it as `address_space(1)` and `noderef`, which forbids dereferencing. Sparse emits warning for all casts and implicit conversions between address spaces and for all illegal dereferences.

Second, Sparse adds *context modifiers* to C function types. These can be used to statically check basic lock/unlock pairings. The *context* is an integer that follows every possible flow path through each function. By default, the context entering a function must equal the context exiting the function, but a function can be annotated as increasing or decreasing the context by some amount. For example, an `acquire` function would be marked as increasing the context by one. If an unannotated function were to call `acquire` without later releasing the lock, a context mismatch would be flagged.

The `xsparse` extension implements address spaces and context modifiers atop xoc.

Address Space Checking `xsparse`'s implementation of address space checking is straightforward. First, `xsparse` extends the `type` attribute to check `address_space` and `noderef` annotations on cast expressions and pointer dereferences. The extended attribute defers to `default` before doing the necessary checks:

```
extend attribute
type(term: ptr C.expr): ptr Type
{
    t := default(term);
    // Sparse type checks here
    return t;
}
```

The primary advantage of this implementation is its use of `default`, which lets `xsparse` add new behavior without worrying about the details of existing behavior. This makes `xsparse` address space checking naturally composable with other extensions. The `xcanconvert` function is extended in the same manner.

Context Checking For context checking, `xsparse` extends the `welltyped` attribute for functions. This allows it to restrict the definition of "well typed" programs to exclude those with context mismatches. `xsparse` uses a traditional control flow analysis module supplied by xoc to generate and traverse the flow graph of each function being type checked. In order to check new control flow introduced by other possible extensions, `xsparse` passes the function's `compiled` attribute—the standard C representation—to the control flow analysis.

Evaluation To evaluate `xsparse`, we compare with it with Sparse, which is implemented as an entire front end. We checked the Linux kernel with `xsparse`, and `xsparse` produced the same warnings as Sparse.

`xsparse`'s implementation is simpler than Sparse's. Sparse is about 25,000 lines of code, while `xsparse` is 345 lines of code. Of course, this result is not surprising because `xsparse` can leverage xoc's infrastructure for extensibility.

Unlike Sparse, `xsparse` can easily be composed with other extensions. For example, `xsparse` can correctly analyze a program that uses the `xalef-iter` extension's `::` operator for iteration or the `xgnu-conditional` extension's binary `?:` operator.

Name	Lines	Description
`xaif`	50	Make `if` and `while` anaphoric, as in *On Lisp* (Graham 1996).
`xalef-check`	24	Add `check` statement as in Alef (Winterbottom 1995).
`xalef-iter`	196	Add iterator expressions as in Alef.
`xgnu-asm`	47	Parse (but do not analyze) GNU inline assembly.
`xgnu-caserange`	61	Allow ranges in `case` labels (`case 0 ... 9:`).
`xgnu-conditional`	14	GNU binary conditional operator `?:`.
`xgnu-minmax`	24	GNU min and max operators `<?`, `>?`, `<?=`, and `>?=`.
`xgnu-typeof`	33	GNU `typeof` type specifier (`typeof(q) p = q;`).
`xlambda`	170	Heap-allocated lexical closures that are compatible with regular function pointers.
`xloop`	168	Labeled `break` and `continue`, as in Java and Perl.
`xpcre`	452	Perl-like syntax for the PCRE regular expression library, including flow-sensitive checks for out-of-range submatch references.
`xrotate`	34	Rotate operators `<<<` and `>>>`.
`xsparse`	345	Workalike for the Sparse program checker (Torvalds and Triplett 2007). (80 lines for type checking, 245 lines for flow checking.)
`xtame`	516	Tame style event-driven programming (Krohn et al. 2007).
`xvault`	641	An implementation of Vault's flow-sensitive type system (DeLine and Fahndrich 2001) for C.

Figure 3. Extensions written using xoc. The lines column counts non-comment source lines.

`xsparse` is not as fast as Sparse. A typical Linux source file (`do_mounts.c`, 600kB and 15,000 lines after preprocessing) takes fifteen seconds to check with `xsparse` but only a tenth of a second to check with Sparse. We believe most of the slowdown is due to zeta's interpreter and not the extensibility mechanisms themselves. Preliminary tests suggest that replacing the zeta interpreter with an on-the-fly compiler will produce a 20–40x speedup for `xsparse`.

4.2 Vault, a high-level protocol checker

For a more complex case study, we compare `xvault`, an implementation of Vault's type system (DeLine and Fahndrich 2001) in xoc, with Coffer, the implementation of Vault in Polyglot by the authors of Polyglot.

The Vault language uses a flow-sensitive type system to enforce protocols in low-level software. Vault uses linear capabilities, or *keys*, to ensure that all tracked objects are always freed and never used without being allocated. Pre- and post-conditions on functions specify how the held set of keys is affected by a function call. Vault includes a few other features: keys can be tagged with a state name, keys can guard variables (which cannot be used unless their key is held), and variant types can support run-time checks on key state.

`xvault` implements all these features except variant types. Coffer adds Vault-like keys to Java, but does not support key states or key guards. Coffer has to worry about classes and exceptions. We omit those features of Coffer from the comparison. Coffer and `xvault` implement Vault's type system in equivalent ways: they add syntax for the flow-sensitive types, add type-checking rules to understand those types, and use a dataflow analysis to check them.

The `xvault` implementation is smaller than Coffer's mainly due to xoc's handling of syntax trees. Coffer must declare semantic actions, making its grammar extension 199 lines (excluding class and exception parsing rules not needed in C), as opposed to `xvault`'s 20. Further, Coffer must define subclasses to represent the abstract syntax for new method and type declarations. These classes require a few boilerplate methods, such as `visitChildren`, `reconstruct`, and `prettyPrint`, which are not necessary in xoc thanks to syntax patterns and generic traversals. In total, Coffer is 2276 lines of uncommented code (excluding package and import declarations and code to manipulate classes and exceptions) while `xvault` is 641 lines.

	Language	Files	Lines
xoc	Zeta	1	34
CIL	OCaml	9	94
xtc	Java + *Rats!*	7	194 + 35
Polyglot	Java + PPG	13	294 + 28

Figure 4. Files modified and lines of code added to implement rotate in various extensible compilers.

Unlike Coffer, `xvault` is composable with other extensions. For example, because Vault's flow analysis is more powerful than Sparse's, a programmer might choose to load Vault for flow analysis and Sparse for address space checking. We have created a few small test programs to verify that this combination works. A version of Polyglot written in J& (Nystrom et al. 2006) can mix Coffer with other extensions, but this process requires the programmer to compose the grammars and pass schedules manually.

4.3 Bitwise rotate

To further compare xoc to recent extensible compilers, we implemented the bitwise rotate operator in section 2 using CIL, xtc, and Polyglot. Although a trivial extension, this exercise explored the fixed costs of each compiler's extension interface and highlighted key differences between their approaches to extensibility and xoc's. Figure 4 summarizes the effort required to implement each extension.

The core of the rotate extension for CIL is only 48 lines of OCaml (not including comments and whitespace) in a single new file. However, because CIL is targeted only at analysis extensions, adding support for the parsing and abstract syntax of the rotate operator required modifications to seven source files in the CIL core. OCaml's pattern matching facilities and CIL's printf-style restructuring library make the rotate implementation for CIL significantly shorter than the implementations for xtc and Polyglot. However, pattern matching required mentally translating the concrete rotate syntax into its abstract syntax, and CIL's use of printf-style format strings means syntax errors can not be statically detected.

The rotate extension for xtc is 194 lines of Java code, plus 35 lines of *Rats!* parser specification. The bulk of the implementation is concerned with navigating xtc's generic syntax trees and dealing

with their dynamic types. The generic syntax trees obviate the need to know or extend any abstract syntax in the rotate extension; however, without the benefit of destructuring syntax, the rotate extension implementation requires knowledge of precise details of concrete program syntax and of the exact relationships between expression-related non-terminals in the grammar.

The Polyglot-provided extension skeleton alone is 108 lines of Java and PPG (Polyglot Parser Generator) code and 98 lines of shell wrapper, again not including comments or whitespace. Implementing the rotate operator added 214 lines of Java/PPG. Because the design patterns that make Polyglot flexible require many distinct interfaces and classes, even simple extensions consist of many classes and large amounts of boilerplate code. For example, in addition to specifying the translation of rotate syntax, the rotate extension must provide implementations and factories for its abstract syntax, specify where it fits into the pass schedule, and provide a driver for compiling programs written in rotate-extended Java. This comparison (Polyglot code that we wrote) is bolstered by the comparison with Coffer (Polyglot code that Polyglot's authors wrote) above.

4.4 Function expressions

To exercise xoc, we wrote an extension called `xlambda` that adds first-class closures, a feature we have long wished for in C. These closures behave like regular function pointers, without the usual C workaround of an extra `void*` parameter. The new keyword `fn` followed by a function definition creates a heap-allocated closure that can be freed with `free`. For example, the following snippet calls `qsort` with a newly constructed closure:

```
void alphabetize(int ignorecase, char **str, int nstr)
{
    qsort(str, nstr, sizeof(char*),
        fn int cmp(const void *va, const void *vb) {
            const char **a = va, **b = vb;
            if (ignorecase)
                return strcasecmp(*a, *b);
            return strcmp(*a, *b);
        });
}
```

`xlambda` provides safe lexical scoping by capturing by-value copies of all necessary variables from the enclosing environment directly in the closure structure at the time of creation. By-value semantics allow closures to have unlimited extent; unlike stack-based closures (for example, GNU C's nested functions), an `xlambda`-created closure is completely self-contained, and thus remains valid after the function that created it returns.

When xoc compiles the `fn` expression, `xlambda` *lifts* the function, moving its definition to the top level of the program and adding an additional argument that points to an environment structure with fields for all of the variables the closure needs from the enclosing environment (`ignorecase` in the example).

The code `xlambda` will generate begins:

```
int lambda_cmp(struct env_cmp *env, const void *a, ...)
{
    if (env->ignorecase)
    ...
```

In addition, `xlambda` emits code that allocates and initializes a structure containing `env_cmp` and a small, dynamically-generated assembly-language trampoline that adds the extra `env` argument and calls `lambda_cmp`. The closure and the trampoline are allocated in the same block, so that freeing the function pointer frees the closure.

Implementation using xoc `xlambda` uses xoc's grammar support to extend the compiler's grammar with `fn` expressions using the grammar specification:

```
expr: "fn" fndef;
```

Restructuring and generic syntax allow the construction of the code that allocates and initializes the closure object to precisely mimic the literal C code ultimately generated, instead of requiring a translation into abstract syntax. For example, the following snippet generates the environment structure based on the free variables of `func`, the function being lifted.

```
fields: list ptr C.sudecl;
for ((v,_) in func.freevars) {
    if(!v.isglobal)
        fields +=
           list[‘C.sudecl{\(v.type) \(mkid(v.name));}];
}
envtype := ‘C.typespec{struct {\fields}}.type;
```

The fine grained scheduling afforded by attributes allows `xlambda` to construct new syntax fragments at any time. Code generated during compile time does not need to be brought "up to speed" (e.g., if type information is necessary); analysis will be performed when needed, even if xoc has finished the corresponding analysis of the rest of the program.

Evaluation Implementing the `xlambda` extension required 170 lines of Zeta code. Of these 170 lines, 4 declare the grammar extension, 26 perform free variable analysis, 40 perform scoping and type checking, and 98 compile `fn` expressions. Given the complexity of implementing closures, we were happily surprised at the simplicity of `xlambda`'s implementation, and the ease with which closures can be added to the C language.

To further evaluate `xlambda`, we also implemented it using the xtc toolkit. This implementation was 682 lines of Java, plus 38 lines of *Rats!*. Like in the xtc rotate extension, much of the code was concerned with navigating the generic syntax trees in ways that allowed the appropriate parts of the trees to be modified. In addition, 210 lines were dedicated to working around the possibility of variable capture due to xtc's lack of automatic hygiene.

4.5 Composability

We designed xoc's extension interfaces with extension composability in mind. We have no way to make sweeping statements about composability, and it is certainly possible to design extensions that are not usable together. Even so, we wrote a few programs using many extensions as a composability sanity check. Here is a program that combines `xaif`, `xalef-iter`, `xlambda`, and `xpcre`:

```
void
foreach(char **str, int nstr, void(*f)(char*))
{
    f(str[0::nstr]);
}

int
main(int argc, char **argv)
{
    while(getline()) {
        foreach(argv+1, argc-1,
            fn void check(char *pat) {
                if(it =~ pat)
                    printf("%s\n", $0.str);
            });
        free(check);
    }
}
```

This convoluted program matches each line of text returned by `getline` against a set of regular expressions given on the command line. The `::` operator, introduced by `xalef-iter`, executes its containing statement repeatedly, with each value from 0 to `nstr`. The `it` variable, introduced by `xaif`, is equal to the last condition

evaluated by `while`; `it` is copied into the `fn` closure properly. The `xpcre` extension contributes the `=~` match and the `$0.str` syntax. We compiled this program using all possible extension orderings and verified that they all compiled to the same, correct code.

As mentioned earlier, we also checked that programs can use `xsparse` and `xvault` together and that `xsparse` correctly handles programs using `xalef-iter` and `xgnu-conditional`, which introduce new control flow.

Xoc makes it possible to load extensions dynamically and to mix extensions. These features alone are an advance over existing work. Although it is certainly possible to write extensions that conflict when composed—and when it detects conflicts, xoc will report an error—we are encouraged by the fact that the extensions we have written *do* compose. Identifying an exact set of conditions that guarantee the composability of extensions is interesting future work. For now, we have refrained from imposing restrictions until we have a better sense of what kinds of extensions people will write.

4.6 Discussion

Based on our experience implementing the extensions listed in Figure 3, we can make some observations about the ease of writing xoc extensions. More data points are necessary to support hard statements, but xoc extensions seem to require relatively little knowledge of the xoc core. The xoc core has 72 grammar symbols, 56 attributes, and 31 extensible functions. Figure 5 lists the number of times particular symbols, attributes, or functions are extended by the extensions in Figure 3. The extensions make their diverse language changes using relatively few of the grammar symbols, attributes, and functions.

The symbols `abdecor1`, `decor1`, and `typespec` and the functions `xapplydecor`, `xdodecor`, `xdosudecor`, and `xsplittypespec` are all involved in processing C type declarations, by far the ugliest part of C. We have not been able to hide the ugliness completely in xoc. Nevertheless, the figure shows that two core grammar symbols (`expr` and `stmt`), and four core attributes (`compiled`, `type`, `welltyped`, and `forward`) are useful for a wide variety of extensions.

	Symbol		Attribute		Function
6	`expr`	9	`compiled`	2	`xapplydecor`
4	`stmt`	8	`type`	2	`xcompilefnsym`
2	`abdecor1`	8	`welltyped`	1	`typefncall`
2	`decor1`	4	`forward`	1	`xcanconvert`
2	`typespec`	2	`sym`	1	`xdodecor`
1	`attr`	2	`vars_out`	1	`xdosudecor`
1	`fndef`	1	`body`	1	`xsplittypespec`
1	`label`	1	`iscomputation`		
1	`qual`	1	`vars`		
1	`top`				

Figure 5. The number of extensions (from Figure 3) that extend each grammar symbol, attribute, and function.

5. Related Work

The contribution of our work is the interfaces that allow programmers to extend xoc dynamically. These interfaces draw on work done in extensible compilers and other language extensibility mechanisms.

5.1 Extensible compilers

Because of the extension-oriented focus, extensions must be easy to write and use; otherwise the base effort required to create a new extension threatens to dwarf the incremental effort required to define the extension-specific details. The following collection of features are essential to xoc:

- dynamic loading of extensions
- changing syntax via extensible grammars
- syntax patterns for manipulating the internal representation
- a typed syntax tree
- lazy attributes for computing and saving analyses
- a general purpose programming language for writing extensions

This section discusses how these features evolved in xoc by examining a few extensible compilers which had a direct influence: CIL, a compiler supporting analysis extensions (Necula et al. 2002); Polyglot, an extensible Java compiler framework (Nystrom et al. 2003); xtc, an extensible C and Java compiler framework (Grimm 2006; Hirzel and Grimm 2007); Stratego, a specification language for program transformations (Visser 2004); and Silver, an extensible attribute grammar system (Van Wyk et al. 2007a,b).

Dynamic loading CIL, Polyglot, xtc, Stratego, and Silver are "compiler kits" that must be rebuilt for each different extension or set of extensions, each time producing a new standalone compiler. Mixing multiple extensions requires composing them manually. A version of Polyglot ported to the J& language (Nystrom et al. 2006) addresses extension composability but still requires constructing a new compiler for each set of extensions.

In contrast, an extension-oriented compiler like xoc accepts plugins during compilation. It is not necessary to rebuild xoc each time the user wants to try a different extension.

Extensible grammars Those extensible compilers that provide support for changing the input grammar (CIL does not) allow extension writers to specify changes by writing additional grammar rules. Polyglot and Silver accept context-free grammar rules but use an LALR parser, making it possible for extensions to add valid grammar rules that are nonetheless rejected by the parser. Stratego solves this problem by using a GLR parser (Tomita 1987; van den Brand et al. 2002), which allows it to handle any context-free grammar (and thus any arbitrary grammar additions). Xtc solves this problem by switching formalisms: it uses parsing expression grammars (PEGs) and a packrat parser (Ford 2004), which also handle arbitrary additions.

Like Stratego, xoc uses a GLR parser to allow arbitrary grammar additions. As discussed in section 3.1, GLR parsing has the added benefit over packrat parsing that it allows the detection of ambiguities introduced by combinations of grammar extensions. Unlike Polyglot and xtc, xoc's grammar modifications are limited to rule addition; more flexible features such as rule removal are at odds with automatic composability. Polyglot for J&, for example, allows rule removal but requires combined grammars to be composed by hand.

Syntax patterns CIL provides simple string-based primitives for restructuring and destructuring concrete syntax, like C's `printf` and `scanf`. Polyglot provides a similar `printf`-style restructuring syntax. Xtc provides a more general mechanism in which patterns are stored in a separate file. A program called the `FactoryFactory` compiles them to Java methods that extensions can call. The CIL and Polyglot approach keeps the patterns near their use but cannot check that they are well-formed when compiling extensions. The xtc approach can check patterns but requires that they be defined in a separate file, apart from their use.

Stratego provides the best of both approaches using concrete syntax patterns (Bravenboer and Visser 2004). Stratego's syntax patterns are easy to use and are syntax-checked at compile time. Like Stratego, xoc provides domain-specific support for concrete syntax patterns in it's implementation language. Unlike Stratego, xoc's syntax patterns are typed, as discussed in the next section.

Typed syntax Parse trees have implicit types: for example, a parse tree representing a statement cannot be used where a parse tree representing a variable name is expected. Compilers differ on whether they expose these types in the implementation language (`StmtNode`, `NameNode`, etc.) or just use a single abstract syntax type (`Node`).

CIL, Polyglot, and Silver use explicitly typed syntax trees. This makes it possible for the implementation language's compiler to check that syntax trees are well-formed (where a statement node is expected, only a statement node can be used). This also gives the compiler writer more control over the internal syntax tree representation. For example, Polyglot uses Java interfaces to classify abstractly related syntax types, such as the `Binary` interface, which is the super type of binary expressions, or `Term`, which is the supertype of AST nodes that can contribute to control flow. Such interfaces allow the implementation language to check that the required functionality of new node types has been implemented.

On the other hand, using a single abstract syntax type makes traversals of foreign syntax particularly easy, since the `Node` object typically defines a `children` array with pointers to the child syntax nodes. Stratego and xtc both take this approach. In contrast, generic traversal using typed abstract syntax requires something more heavyweight, like the visitor pattern (Gamma et al. 1994).

Xoc provides a typed syntax tree with subclassing support tailored to the syntax tree (`C.expr` and so on). Typed syntax is particularly important for syntax patterns: because xtc and Stratego use untyped syntax trees, they cannot diagnose errors in which the wrong type of node is passed to a restructuring pattern (for example, using syntax for a statement where a variable name is expected).

Internally, xoc's syntax trees have a uniform representation exposed by the `astsplit` and `astjoin` primitives. This makes xtc- and Stratego-like generic traversals possible. However, it trades flexibility for loss of control; it is not possible to customize the internal representation in xoc as it is in Polyglot.

Lazy attributes Compilers are traditionally organized as a sequence of passes. Adding extensions in such a model requires defining how the extension's computation fits into the pass structure. Polyglot extensions explicitly declare their scheduling requirements to the Polyglot core, which deduces a schedule; CIL, xtc, and Stratego require scheduling passes by hand. Both approaches complicate extension design, since extensions must be aware of when various analyses take place.

Xoc's solution to scheduling draws on Silver, where all computations are expressed using an attribute grammars and can thus draw on extensive work on attribute grammar scheduling (Knuth 1968). Like Silver's attribute grammars, xoc's lazy attributes let the programmer define tree traversal implicitly; the required order naturally arises from the sequence of references to other attributes. However, xoc's added flexibility comes at the loss of formalism: xoc's attribute functions cannot be automatically analyzed like conventional attribute grammars can.

General-purpose programming language Most compilers are written using general-purpose programming languages like C, Java, or ML, but some projects use custom domain-specific programming models. Domain-specific models can simplify common idioms or constructs and provide stronger static guarantees, but, if not coupled with a general-purpose language, they can make some tasks considerably more difficult.

Stratego is based entirely on term rewriting; Silver is based entirely on attribute grammars. Both models are very different from the general-purpose programming languages whose programmers are xoc's target audience. Learning a different model of computation is a significant barrier to entry for these systems.

Xoc starts with a general-purpose programming language and adds domain-specific constructs—syntax patterns, a grammar type system, and lazy attributes—to make writing extensions more convenient. Unlike in Stratego and Silver, the domain-specific constructs are integrated with a general-purpose language. Syntax patterns and lazy attributes can be bypassed in favor of lower-level constructs like generic traversal and extensible functions when necessary. We have also experimented with domain-specific constructs for implementing type checkers (Bergan 2007), but their use does not preclude the use of the general-purpose language. On the other hand, providing general-purpose constructs precludes static detection of errors like attribute circularity or coverage of new syntax.

5.2 Other approaches to language extension

Macros Macros are the most popular method for extending a language, mainly due to the power demonstrated by Lisp's macros (Hart 1973; Steele and Gabriel 1993; Graham 1996). Lisp was also the first to introduce restructuring expressions—the Lisp term is *quasiquotation*—to make code generation easier to write and understand. Programmers have ported Lisp's macros into other languages (e.g., Bachrach and Playford 2001; Baker and Hsieh 2002; Weise and Crew 1993). Macros are purely syntactic transformations, making them excellent for adding new syntax but not useful for semantic changes to existing syntax. Since macros cannot access compiler internals, they typically do not even type check their arguments, producing cryptic errors when invoked incorrectly.

Xoc's extension interface is less concise than macros, but in return for the loss of brevity, extension writers get automatic syntax checking, the ability to reuse compiler analyses, and the ability to introduce semantic changes or restrictions.

Provable extensions Some recent work has focused on being able to prove correctness properties for specific classes of language extensions. Semantic type qualifiers (Chin et al. 2005) allow users to define typing rules for extended sets of qualifiers; the rules are automatically validated against desired runtime invariants. Other work has made progress in proving the correctness of dataflow analyses, compiler transformations, and optimizations (Lerner et al. 2003, 2005). Xoc's current design prefers flexibility of extension to provable correctness. A system that enforced safety invariants and proved correctness properties for some extensions while allowing others to escape the resulting limitations would give useful flexibility to extension authors.

6. Conclusion

Xoc is an extension-oriented compiler, which allows an extension writer to make a small change to the base language and combine this extension with others, perhaps written by others, much like how web browsers and other software load content-specific plugins.

A challenge in the design of xoc is to give extension writers the power to modify the grammar and manipulate and analyze the AST without forcing the extension writer to understand the representations within the compiler. Xoc resolves this tension by using syntax patterns, written in the syntax of the base language, to manipulate language fragments and ASTs, and using AST attributes, computed on demand, to hide the scheduling of compiler passes.

Experience with using xoc to write 15 extensions that extend the compiler in various different ways confirms that these interfaces work well. None of the 15 extensions needed to bypass the interfaces, and in fact xoc provides no mechanism for doing so.

Furthermore, the line counts of these extensions indicate that the extensions are easier to develop than corresponding ones written using a domain-specific front end or extensible compilers.

Acknowledgments

Todd Millstein provided much useful advice on the design of xoc. Robert Grimm provided advice on using xtc. The anonymous reviewers gave valuable feedback on an earlier draft. During this work, Russ Cox was supported in part by a fellowship from the Fannie and John Hertz Foundation, and Eddie Kohler by Sloan Research and Microsoft Research New Faculty Fellowships. This project was partially supported by the National Science Foundation under Grant Nos. 0430425 and 0427202, and by Nokia Research Center Cambridge.

References

Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of the 16th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

Jason Baker and Wilson C. Hsieh. Maya: Multiple dispatch syntax extension in Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

Tom Bergan. *Typmix*: a framework for implementing modular, extensible type systems. Master's thesis, University of California Los Angeles, 2007.

Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2004.

Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, 1994.

Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice-Hall, 1996.

Robert Grimm. Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

Michael Hammer. An alternative approach to macro processing. In *Proceedings of the International Symposium on Extensible Languages*, Grenoble, France, 1971.

Timothy P. Hart. MACRO definitions for LISP. AI Memo 57, MIT AI Project—RLE and MIT Computation Center, 1973. (reproduced in Steele and Gabriel 1993).

Martin Hirzel and Robert Grimm. Jeannie: Granting Java native interface developers their wishes. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2007.

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986.

Max Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, 2002.

Nathanial Nystrom, Michael Clarkson, and Andrew Myers. Polyglot: an extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

Nathanial Nystrom, Xin Qi, and Andrew Myers. J&: nested intersection for scalable software composiiton. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

Jukka Paakki. Attribute grammar paradigm: a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of Lisp. In *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages*, 1993.

W. Teitelman. Pilot: A step towards man-computer symbiosis. Technical Report AITR-221, Massachusetts Institute of Technology, 1966.

Masaru Tomita. An efficient augmented context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46, January–June 1987.

Linus Torvalds and Josh Triplett. Sparse – a semantic parser for C. `http://www.kernel.org/pub/software/devel/sparse/` (retrieved December 2007), 2007.

Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 143–158, 2002.

E. Van Wyk, D. Bodin, L. Krishnan, and J. Gao. Silver: an extensible attribute grammar system. In *Proceedings of the 7th Workshop on Language Descriptions, Tools, and Analysis*, 2007a.

Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for Java. In *Proceedings of the European Conference on Object-Oriented Programming*, 2007b.

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

Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

Phil Winterbottom. Alef reference manual. In *Plan 9 Programmers Manual, Volume Two*. Harcourt Brace Jovanovich, 1995.