Reliable and Automatic Composition of Language Extensions to C

The ableC Extensible Language Framework

TED KAMINSKI, University of Minnesota, USA
LUCAS KRAMER, University of Minnesota, USA
TRAVIS CARLSON, University of Minnesota, USA
ERIC VAN WYK, University of Minnesota, USA

This paper describes an extensible language framework, ableC, that allows programmers to import new, domain-specific, independently-developed language features into their programming language, in this case C. Most importantly, this framework ensures that the language extensions will automatically compose to form a working translator that does not terminate abnormally. This is possible due to two modular analyses that extension developers can apply to their language extension to check its composability. Specifically, these ensure that the composed concrete syntax specification is non-ambiguous and the composed attribute grammar specifying the semantics is well-defined. This assurance and the expressiveness of the supported extensions is a distinguishing characteristic of the approach.

The paper describes a number of techniques for specifying a host language, in this case C at the C11 standard, to make it more amenable to language extension. These include techniques that make additional extensions pass these modular analyses, refactorings of the host language to support a wider range of extensions, and the addition of semantic extension points to support, for example, operator overloading and non-local code transformations.

CCS Concepts: • **Software and its engineering → Extensible languages**; **Translator writing systems and compiler generators**;

Additional Key Words and Phrases: extensible compiler frameworks, domain specific languages, language composition, attribute grammars, context-aware scanning

ACM Reference Format:
Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 98 (October 2017), 29 pages. https://doi.org/10.1145/3138224

1 INTRODUCTION AND MOTIVATION

A fundamental issue in software development is the reuse of independently-developed software components that allow programmers to easily and reliably reuse the work of others. In most programming languages, this is supported by a module or package mechanism so that a developer can distribute their work as a library. The programmer using the library need only understand the

Authors' addresses: Ted Kaminski, Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA, tedinski@cs.umn.edu; Lucas Kramer, Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA, krame505@umn.edu; Travis Carlson, Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA, travis.carlson@cs.umn.edu; Eric Van Wyk, Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA, evw@umn.edu.

2475-1421/2017/10-ART98
https://doi.org/10.1145/3138224

exposed interface and can use several in the same program without worrying about their interaction. In short, they "just work" and provide a means for the modular development of software using components developed by independent parties.

This work takes another step towards a vision for programming in which programmers can import new language features into their compilers with the same ease at which they now import libraries into their programs. Language features, packaged as composable language extensions, can provide concrete syntax for new (domain-specific) language constructs, semantic analyses (such as error checking) of these and host language constructs, and optimizations of the new constructs. In this vision, programmers (as users of language extensions) are able to freely and easily use the combination of language features of their choice that address challenges of the task at hand without needing to understand the inner workings of their compiler or language extension specifications.

Specifically, this paper describes ableC, an extensible specification of C, at the C11 standard. It uses the Silver [1] attribute grammar system [Van Wyk et al. 2010] and the Copper parser and context-aware scanner generator [Schwerdfeger 2010; Van Wyk and Schwerdfeger 2007], which is bundled into Silver. Silver specifications define the semantic analysis and code generation tasks as attributes and their defining equations. These specifications also define concrete syntax as regular expressions and context-free grammars that are extracted and processed by Copper to construct a scanner and parser. Language extensions are thus also specified using Silver. Extensions can add new concrete syntax by specifying new productions and symbols (nonterminals and terminals) that are composed with the host language concrete syntax specification and the specifications of other selected language extensions. Corresponding new productions and nonterminals for the abstract syntax, and the mapping from concrete to abstract syntax, is also specified. Extensions can also introduce extended semantics, in the form of new attributes and attribute equations on both existing host-language abstract syntax productions and newly introduced extension productions.

We make use of two modular analyses provided by these tools to ensure that language extensions specify *additions* to the syntax and semantics, and not potentially conflicting *modifications*, of the C host language. The *modular determinism analysis* [Schwerdfeger 2010; Schwerdfeger and Van Wyk 2009] analyzes the concrete syntax specification of an extension in relation to the host language. It ensures that any collection of extensions that individually pass this analysis can be automatically composed to form a specification with no lexical ambiguities and from which a deterministic LALR(1) parser can be generated. The single caveat is that the programmer may need to perform a disambiguation that is similar to what is done in Java when two packages each introduce a different class with the same name. The second is the *modular well-definedness analysis* [Kaminski 2017; Kaminski and Van Wyk 2012] that provides a similar guarantee on the semantics side: that the attribute grammar for the composed language is well-defined [Knuth 1968; Vogt et al. 1989] and thus will not terminate abnormally due to missing equations. In essence, these analyses ensure that language extensions can be reliably and automatically composed, meaning that the resulting specification of an extended language is guaranteed to be unambiguous and complete.

The modular analysis guarantees are made at the meta-language level, on the Silver and Copper specifications. The modular well-definedness analysis, and other static checks performed on Silver specifications, provide the Silver language with a solution to the so-called *expression problem*, as described in Section 2.2. This provides an illuminating framework to discuss distinguishing characteristics of Silver and other approaches to language extension. While these analysis ensure that the composition of the host language and extensions succeed is unambiguous and complete, they do not provide any guarantees about the meaning of the extended language. We discuss work on meta-level *semantic* guarantees in Section 10.3.

[1] See http://melt.cs.umn.edu/silver, archived at https://doi.org/10.13020/D6QX07

The modular analyses guarantees allow this approach to make a clear distinction between extension users, that is, programmers, and extension developers. Extension developers need an understanding of language design and how to implement their extension using Silver. They must understand the modular analyses and adjust their specifications to satisfy them. Extension users, however, do not need this understanding. To create a custom extended language, the programmer need only list the desired extensions. The supporting tools can then automatically compose the specifications of the host language and extensions to generate a compiler for the extended language.

Most previous work on extensible languages and frameworks does not provide the guarantees described above and thus cannot make a clear distinction between extension users and developers. Work that does provide similar guarantees supports language extensions that are less expressive than those supported by ableC. This paper describes ableC as our effort to find a "sweet spot" of language extensibility in which syntactically and semantically *expressive* extensions can *reliably and automatically* be composed to form a *working* translator that will not terminate abnormally.

Contributions. The paper makes these contributions:

- A description of ableC, an extensible specification of C, at the C11 standard.
- Illustration of sample language extensions that pass Copper and Silver's modular analyses and thus reliably and automatically compose. A regular expression language extension further illustrates how extensions can build on top of other language extensions.
- Identification of points of tension where desired syntax was modified to accommodate the modular restrictions.
- Description of refactorings to and features of the host language syntax specification to make it more amenable to additional language extensions.
- Additions to the host language semantic analysis and code generation specification to enable more types of language extensions. Specifically to support operator overloading and non-local code transformations that lift declarations to the global scope.
- Definition of a new criterion for the expression problem useful in framing challenges in language extensibility.

Road-map. Section 2 first provides an overview of ableC that elaborates on the goals of the work (Section 2.1), then frames the discussion of language extensibility in terms of the expression problem and provides a new criterion for it (Section 2.2), and describes the structure of the ableC system (Section 2.3). The remainder is organized around the perspective of language extensibility as seen by programmers that compose language extensions, developers that implement language extensions, and the designers of the host language specification. Section 3 illustrates how programmers direct the supporting tools to compose the chosen set of language extensions in ableC. Sections 4 and 5 describe the specification of the syntax, and respectively, the semantics of language extensions. These serve as background and describe how we use the Silver and Copper tools in ableC. Sections 6 and 7 describe how the host language designer can write the specification of the syntax and, respectively, semantics of the host language to enable additional types of language extensions. Section 8 shows how extensions can use and extend other extensions. Section 9 describes related work before section 10 discusses future work and concludes.

2 EXPRESSIVE LANGUAGE EXTENSIONS IN ABLEC

2.1 Full-featured Language Extensions and Their Composition

Our work on Silver, Copper, and ableC is related to previous work in extensible languages and frameworks in which independently-developed language features can be reused and composed to support a highly modular development of programming languages. Recent examples include

```
1.  #include <stdio.h>
2.
3.  typedef datatype Tree  Tree;
4.  datatype Tree {
5.    Fork (Tree*, Tree*, const char*);
6.    Leaf (const char*);
7.  };
8.
9.  cilk int count_matches (Tree *t) ;
10.
11. cilk int main (int argc, char **argv) {
12.   int count;
13.   Tree *tree;
14.   // initialize tree from a file
15.   spawn count = count_matches(tree);
16.   sync;
17.   printf ("matches = %d\n", count);
18.   cilk return 0;
19. }
20. cilk int count_matches (Tree *t) {
21.   match (t) {
22.     Fork(t1,t2,str) -> {
23.       int res_t1, res_t2, res_str;
24.       spawn res_t1 = count_matches(t1);
25.       spawn res_t2 = count_matches(t2);
26.       if ( str =~ /[1-9]*/ )
27.         res_str = 1;
28.       else
29.         res_str = 0;
30.       sync;
31.       cilk return res_t1 + res_t2 +
32.                   res_str;
33.     }
34.     Leaf(/[1-9]*/) -> {
35.       cilk return 1; }
36.     _ -> { cilk return 0; }
37.   } ;
38. }
```

Fig. 1. An example program, using composable language extensions that provide algebraic data types with extensible pattern matching, Cilk-style parallel programming abstractions, and regular expressions, which extend the host language and the pattern-matching syntax of the algebraic data type extension.

frameworks for extending Java such as SugarJ [Erdweg et al. 2011], MetaBorg [Bravenboer and Visser 2004], the JastAdd [Ekman and Hedin 2007b] extensible Java Compiler [Ekman and Hedin 2007a], frameworks for extending C such as Xoc [Cox et al. 2008], XTC [Grimm 2006], and mbeddr [Voelter et al. 2012], language embedding using staging as in DeLite [Rompf and Odersky 2010], and Wyvern [Omar et al. 2014] and VerseML [Omar 2017], two type-driven extensible languages.

To clarify the general goals of these approaches and identify the distinguishing characteristics of the ableC approach consider the example program in Fig. 1. This ableC program uses 3 independent language extensions to count, in parallel, the number of nodes in a binary tree whose string values match a specified regular expression. Extension syntax begins with a *marking terminal* (see Section 4) and these are underlined in this and all other examples in the paper.

(1) The first extension provides algebraic datatypes (ADTs) as found in ML or Haskell. Lines 3-7 define the structure of a binary tree that contains strings at its nodes. Lines 21, 22, 34, and 36 are the pattern-matching constructs that inspect the tree nodes and extract the values stored on those nodes in a way that is easier and safer to use than C struct and union data structures.
(2) The second provides parallel task abstractions from Cilk [Frigo et al. 1998], a monolithic extension of C. Lines 24 and 25 spawn two parallel Cilk functions to recursively process the tree. The `sync` statement on line 30 waits for the spawned tasks to finish, before computing the sum and returning it.
(3) The third extension provides regular expression matching facilities commonly found in scripting languages such as Perl and Python. On line 26 a new infix operator (=~) is used to match the regular expression. This extension not only extends the host language, but also extends the pattern language introduced in the ADT extensions. This is seen on line 34 where a regular expression pattern is nested in the `Leaf` pattern constructor.

This program shows the programmer experimenting with three different language extensions in a single program. All three extensions provide new syntax (new language constructs) that are, after semantic analysis, translated down to plain C. The resulting C program is then compiled by a traditional C compiler.

These extensions demonstrate two types of extension-based semantic analysis. First, the ADT extension checks that patterns in a match statement are linear (each variable only appears once), an analysis over the pattern constructs introduced by the extension. Second, the Cilk extension does semantic analysis over not only its constructs but also host-language constructs. To translate a Cilk function to C, two versions of it are generated — a fast and a slow clone [Frigo et al. 1998]. These clones implement the work-stealing parallel task abstractions in the Cilk run-time. The slow clone contains multiple entry points for resuming execution when work (*e.g.* from a `spawn` of a function) is stolen to be shared among processors; thus it maintains a so-called cactus-stack of frames similar to those in a standard activation record stack. To generate C code to implement this, information about all local variables in the function must be collected. This requires an analysis of not only the Cilk constructs, but also the C constructs and other extension constructs (such as the `match` construct on line 8 in Fig. 1). As described below, this ability to extend software (in this case a C translator) with new variants (here, syntax) and new operations (here, semantic analysis) over new variants is sometimes referred to as the *expression problem* [Wadler 1998].

In drawing out distinctions between ableC and other approaches to language extensibility we consider two concerns: who composes the language features and how expressive the extensions are.

Who composes the language features? Does someone who understands compiler implementation techniques need to write "glue code" to compose the extensions with each other and the host language? Or, can a programmer, with no knowledge of language implementation techniques, reliably and automatically perform the composition? It is our contention that for extensible languages to have a significant impact on the practice of software development it must be programmers that can perform this composition and that this composition reliably results in a working compiler for the programmer-specified extended language. Some systems, such as SugarJ and JastAdd, support the automatic composition of extensions. However, this composition can fail. In SugarJ, for example, this may result in an ambiguous grammar specifying the syntax of the new language. Even though a generalized parser is used in this system and can thus parse ambiguous grammars, the resulting parse may generate multiple parse trees. Problems like this (a particular form of abnormal termination of the compiler) can leave the non-language expert programmer in a difficult situation. The ableC approach draws a critical distinction between a programmer that *uses* extensions and an extension developer that *writes* them.

How expressive are the supported extensions? While distinctions raised by the previous question can be made objectively, some distinctions about expressiveness are subjective. Nearly all systems place some restrictions on syntactic expressiveness as the parser for the extended language must be able to differentiate new language constructs. (An exception here is mbeddr [Voelter et al. 2012] which uses a projectional editor and all syntactic ambiguities can be resolved by the programmer directly in the editor.) While systems based on generalized parsing (such as SugarJ and Metaborg) allow for a wider variety of syntactic extension, the restrictions imposed by the modular analyses in ableC still leave room for rather expressive syntax without paying the price of possibly ambiguous parses of user programs. The extensions in Fig. 1 can be nested and naturally fit into the host language. In addition to these, several other extensions to ableC have been developed that support the assertion that expressive extensions can be reliably and automatically added to a host language. These include embedding SQL in C for statically type-safe queries, Halide-style parallelization and vectorization transformations [Ragan-Kelley et al. 2012], term-rewriting facilities (as in TOM [Balland et al. 2007; Moreau et al. 2003] and Kiama [Sloane 2011]), Go-style concurrency constructs, C++ style templates for structure and functions, C++ style type-safe enumerated types, numeric interval types that overload arithmetic operators, generic vectors, and lambda-expressions. Many of these are described in the paper. Others can be found in a corresponding technical report [Kaminski et al. 2017] or in the ableC documentation and software[2].

At least some questions of semantic expressivity can be answered more objectively, specifically, does the approach support extensions that can perform semantic analysis of host language constructs, even when they contain constructs introduced by other extensions? Some systems such as JastAdd do support this, but require the writing of additional "glue code" to compose the specifications, thus making it difficult for non-expert programmers to orchestrate the composition of different language extensions. The expression problem provides a clarifying framework for considering these kinds of questions and is discussed in the following sub-section.

2.2 Extensible Language Framework Criteria

ableC language extensions written in Silver can add both new syntactic constructs and new semantic analyses over those new constructs, host language constructs, and constructs introduced by other extensions. Silver provides a solution to an old problem that Wadler [1998] named the *expression problem.* The core of this problem is the ability to modularly add new data variants *and* new operations over the pre-existing and new data variants. In considering the expression problem as applied to the representation of abstract syntax and applying it to Silver, new syntactic constructs correspond to new data variants and new semantic analyses correspond to new operations. Zenger and Odersky [2005] paraphrased Wadler's description identifying four criteria that a language must satisfy to solve this problem and added a fifth. We add another criterion that needs to be solved to support the type of automatic, programmer-driven language feature composition described above. We list all of these below with their implications for extensible languages and will point out in the remainder of the paper how Silver, and thus ableC language extensions written in Silver, satisfy these criteria.

*EP*1: *extensibility in adding new data variants and new operations:* In attribute grammars (AGs), this amounts to adding new productions to the grammar (variants) and new attributes and their defining equations (operations). Many attribute grammar systems, including Silver, satisfy this criterion since new equations can be specified for productions defined in different modules.

*EP*2: *strong static typing:* Besides avoiding typical runtime type errors, of specific interest is the assurance that no operation will be missing an implementation for a variant it applies to. For AGs, this means that no production in the composed attribute grammar will be missing an equation that is needed in computing attribute values since missing equations result in abnormal termination of semantic analysis.
In higher-order attribute grammars with forwarding [Van Wyk et al. 2002], such as Silver, not all productions need a defining equation for all attributes that decorate the nonterminals of the production. Thus the modular well-definedness analysis tracks the flow of attribute values and is more sophisticated than merely checking that all possible equations are present.

*EP*3: *no modification of existing code:* In our setting this means that the composition of the host language and multiple extensions cannot require modifying their specifications.

[2]See http://melt.cs.umn.edu/ableC for sources and artifact, archived at https://doi.org/10.13020/D6VQ25

- *EP*4: *separate compilation and type checking:* Silver (unlike many AG systems) separately type checks and compiles AG modules. The modular well-definedness analysis ensures that strong static typing can be done independently on individual modules before they are composed.

- *EP*5: *Zenger and Odersky's "independent extensibility":* It must be possible to *compose* independent extensions to the base system without one having prior knowledge of the others.

For languages that satisfy the above five criteria, one may still need to write code in that language, sometimes called "glue code", to compose the base artifact and the independently-developed extensions. This makes it much less feasible for a non-expert programmer to direct the composition of language features. We thus introduce an additional criterion:

- *EP*6: *automatic composition:* No "glue code" need be written to compose a collection of independent extensions with the base artifact. While not necessarily applicable in all settings, for extensible languages, satisfying this additional criterion ensures that a non-expert programmer can (direct the tools to) perform the composition of the host language and the language extensions. For ableC and other languages specified using Silver this criterion means that no attribute equations need to be written by the end-user: the programmer trying to use extensions.

Reliable and automatic composition of language extensions. It is our conjecture that for an extensible language or extensible compiler framework to support the composition of independent language extensions by a non-expert programmer and have impact in the mainstream of software development, it must at least solve the following two challenges, and ableC does.

(1) Parsing extended programs must be reliable and always result in either a syntax error or the single expected parse tree. This does not necessarily preclude the use of generalized parsing, but some reliable form of modular ambiguity detection or means for isolating extension constructs (*e.g.* as is done in Wyvern [Omar et al. 2014], VerseML [Omar 2017], or Eco [Diekmann and Tratt 2014]) would be needed. PEGs [Ford 2004] construct a single parse tree but the order in which grammars are composed determines which ambiguities are automatically removed, which results in unexpected shadowing of productions and parse trees and thus may give unexpected results.

(2) The approach for abstract syntax representation and semantic analysis of the extended program needs to meet all the expression problem criteria given above.

The expression problem criteria (*EP*1-*EP*6) apply to the meta-language: to Silver, not ableC. ableC benefits because it is written in Silver, and these criteria provide static guarantees that the compiler can be organized in highly modular ways, supporting extensions as independent modules. Our discussion of the expression problem is not relevant to the semantics of the ableC language. In our context this means that the composition of language extensions (that pass the modular analyses) are unambiguous and complete. This does not provide any guarantees about semantics or meaning of the programs written in the extended language. For example, neither Silver or ableC provide support to ensure that there is no incorrect capture of name uses by declarations generated by language extensions. We discuss work in this area in Section 10.3

2.3 ableC - Attribute Grammar Based Language Extensions for C

ableC is a specification of C, at the C11 standard, written in Silver. The concrete syntax is specified using the familiar formalisms of a context-free grammar in which regular expressions are associated with the terminal symbols. From these a scanner and parser can be generated. The abstract syntax and static semantics are specified as an higher order [Vogt et al. 1989] attribute grammar [Knuth 1968]. These specifications are all written using Silver. Concrete syntax specifications are extracted

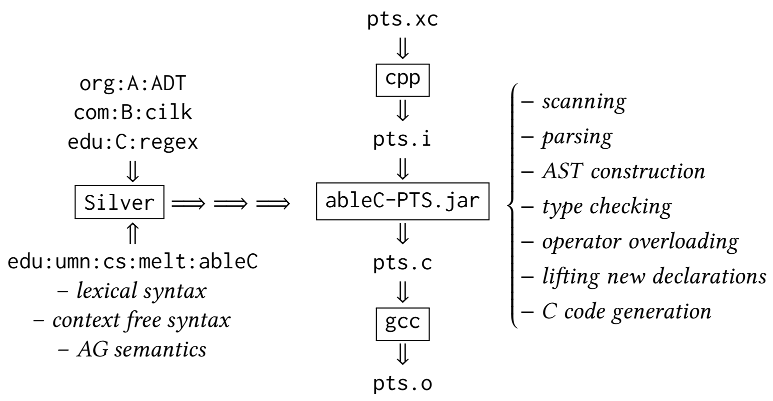

Fig. 2. Structure of ABLEC and example application for the extended program in Fig. 1.

and passed to COPPER, a context-aware scanner and parser generator that is integrated with SILVER, and from the attribute grammar specifications a demand-driven AG evaluator is generated. SILVER orchestrates the translation of these specifications to Java and their compilation into an executable JAR file. The left of Fig 2 shows three language extensions, `org:A:ADT`, `com:B:cilk`, and `edu:C:regex` being composed with the C host language specification for form this executable for this extended language. SILVER grammars are expected to be named using the developer's Internet domain name to ensure grammar elements have unique names when fully-qualified by the grammar name. The three extensions are given names to indicate that they could be developed by independent parties even though we have written them all and their actual implementations used the same `edu:umn:cs:melt` prefix in their names.

The process of using a generated instance of ABLEC (with or without any language extensions) is also outlined in Fig 2. An extended or plain C program is run through the C pre-processor cpp, processed by the ABLEC instance to create a plain C program, which is then finally compiled by GCC. The primary tasks performed by the ABLEC instance are listed in the right of the figure and described in the remainder of the paper. In this regard, ABLEC is not unlike other extensible language frameworks such as Polyglot [Nystrom et al. 2003], an extensible framework for Java in which language specifications/extensions are written in Java. One can write specifications (in SILVER for ABLEC, Java for Polyglot) that use the host language specification to create new dialects of that host language, perform advanced static analyses, or construct translators to other languages.

The novelty of ABLEC is that the supporting tools (SILVER and COPPER) and the design and implementation of the C specifications that make up ABLEC support the vision of reliable and automatic composition on independently-developed language extensions described in the introduction. This is not possible in, for example, Polyglot.

In addition, the host language C specifications in ABLEC provide a number of extension points to enable classes of language extensions that would not otherwise be possible. It is worth noting that most extensions that we have developed for ABLEC do not require these. They were written and passed the modular analyses without any changes needing to be made to the host language specification. These extension points in the host language allow new categories of language extensions to be made. These include mechanisms to support the adding of new infix operators to the concrete syntax, type-based overloading of existing infix operators, non-local transformations that propagate declarations up to an enclosing scope in the syntax tree, and extensible type qualifiers. These are enhancements to the host language that cannot be added as composable language *extensions* as we define them in this paper as they change the behavior of the language. Operator overloading *changes*, for example, the equations on the production for the addition (+) operator to query its sub-expression types for a production that will overload it at those types. Thus, host language designers need to pay some attention to adding extension points such as those described in these two sections. This illustrates the balance that we are attempting to strike that provides strong guarantees of extension composition but still supports expressive language extensions.

3 THE PROGRAMMER'S PERSPECTIVE

The extended C program in Fig. 1 does not make use of any novel language features; it is the use of all of them in a single program that is unique and of interest here. Cilk [Frigo et al. 1998] was developed as a version of C with additional features for parallel programming with the same constructs (though with slightly different syntax) as in the example. Similarly, TOM [Moreau et al. 2003] provides an extended version of C that supports algebraic datatypes and term rewriting, and regular expressions are commonly found in scripting languages. The programmer writing this program may want to use this particular combination of language features, but tools like Cilk and TOM limit them to using just one or ad-hoc combinations of them. ableC allows this programmer to choose the set of language features that suit the task at hand, without (*i*) requiring them to write any "glue code" to compose these extensions, or (*ii*) there being any chance that some particular combination of extensions or programs that use them will cause the automatically-generated translator to terminate abnormally.

Programmer directed composition. To generate an extended ableC translator, the programmer needs to merely select the set of extensions that they desire for the task at hand. Fig. 3 shows the specification that the programmer would write to generate the extended ableC translator that can process the program in Fig. 1. This directs an extension composition DSL to generate the Silver code, simply a declaration of a parser and a main function with a call to a driver, that defines the custom translator (called PTS) for the project based on the ableC host language using the specified language extensions. This is illustrated in Fig 2. This translator uses the generated scanner and parser to detect syntax errors or, in their absence, construct the abstract syntax tree for the extended language program. On this tree attributes are evaluated to perform semantic analyses to check for domain-specific type errors and opportunities for optimization, and if no errors are reported, generate plain C code that can be compiled using a traditional C compiler. Currently we require the use of GCC or Clang as they have a few extensions that simplify code generation for extensions. This is described in Section 7.

```
construct PTS as
edu:umn:cs:melt:ableC
translator using
 org:A:ADT;
 com:B:cilk;
 edu:C:regex;
```

Fig. 3. The programmer written specification for composing the extensions in ableC.

The task of ensuring the selected extensions have passed the modular analyses falls on the extension developers (as described in Sections 4 and 5). If all the extensions pass these analyses then this composition process is guaranteed to succeed. The modular determinism analysis ensures that no lexical or parse table conflicts can occur in the composed language, with the single exception that *marking* terminals may have overlapping regular expressions and the programmer will need to resolve this ambiguity. As described below, each extension production that declares a new type of host language phrase (*e.g.* expression, statement, or declaration) must begin with an initial symbol called a marking terminal. Examples from Fig. 1 include `datatype`, `match`, `cilk`, and the forward slash ("`/`"). Typically these are keywords, but regular expressions use the slash as a marking terminal. For example, if the keyword `match` was also the marking terminal for another extension included in Fig. 3, then a lexical ambiguity would occur and would be reported by Copper. But this is easily fixed by the programmer. The composition DSL supports a mechanism in Copper for specifying a prefix for overlapping terminals that the programmer uses in the program to disambiguate the phrases matched by more than one marking terminal. To do this, the programmer could add the clause `with prefix "ADT"` to the `org:A:ADT` extension entry in Fig. 3 to allow `ADT::match` to be written instead of `match` when using the match statement, thus fixing the lexical ambiguity. A similar specification is written for the other extension with a `match` marking terminal. The programmer can also specify a default to be recognized when no prefix is written. This is similar to what Java programmers do: using the full package name of a class when two or more imported packages define a class with the same name.

In using ableC, programmers need not write Silver or Copper specifications to compose language extensions. Thus they need not write the "glue code" mentioned in expression problem criterion *EP*6. This composition DSL illustrates the extent of concerns the end-user has: extending what host language, with what set of extensions, and providing any prefixes for clashing marking terminals.

4 DEVELOPING CONCRETE SYNTAX FOR ABLEC EXTENSIONS

This section discusses how Copper's context-aware scanning and modular determinism analysis are used in ableC and by extension developers to define their extension's concrete syntax and check that it will compose with other, similarly checked, language extensions. Note that we do not need the extensibility "in both directions" criterion of the expression problem (*EP*1) for concrete syntax specifications. The expression problem and its criteria is used to frame our discussion on the AG semantics specifications, which is discussed in Section 5.

Specification of lexical and concrete syntax. Copper uses the traditional formalisms of regular expressions and context-free grammars to specify the lexical and, respectively, concrete syntax of languages and language extensions. Fig. 4 shows part of the concrete syntax specification (in an abbreviated form) of the `match` construct in the ADT extension. After declaring the grammar name and importing the host language specification, since components of it are referenced, a terminal symbol for the match keyword is declared. It is indicated as a marking terminal. Next is the production for the `match` statement, using host language nonterminals `Stmt_c` and `Expr_c` and the extension defined nonterminal `Clauses_c`, which derives a sequence of `Clause_c` phrases. Three new nonterminals and a new terminal symbol are then defined. The following production specifies a single clause in a `match` statement as being an pattern, a binding symbol, and statements inside curly braces. Note that terminal symbols whose regular expressions are simple strings can be written inside single quotes. (This figure does not show the traditional mechanisms used to indicate that `match` has precedence over host identifiers nor the semantic actions that construct abstract syntax trees from concrete ones since these do not affect composition.)

```
grammar org:A:ADT;
imports edu:umn:cs:melt:ableC;

marking terminal MATCH /match/;
Stmt_c ::= MATCH '(' Expr_c ')' '{' cs::Clauses_c '}'

nonterminal Clauses_c, Clause_c, Pattern_c;
terminal BIND /->/;
Clause_c ::= Pattern_c BIND '{' Stmt_c '}'
```

Fig. 4. Partial abbreviated Silver specification of `match` construct in the ADT extension.

Context-aware scanning. One of the unique features of Copper that we rely on is context-aware scanning [Schwerdfeger 2010; Van Wyk and Schwerdfeger 2007]. This leads to fewer lexical ambiguities and thus fewer LR parse table conflicts. To illustrate how this works, consider the `BIND` terminal. The host language terminal symbols for parentheses and curly brackets are used in this extension, and the host terminal for pointer/field dereference with regular expression `->` could also have been used. Note that this host terminal and `BIND` have overlapping (actually the same) regular expressions. But there is no lexical ambiguity because the context in which the host `->` operator terminal can appear is different from the context in which the `BIND` terminal can appear. The latter is only valid after a pattern, whereas the host `->` is valid after a host expression nonterminal (`Expr_c`). In an LR parser, a terminal is "valid" in a context, represented by a LR parser state, when that terminal has a *shift*, *reduce*, or *accept* action, but not an *error* action. The scanner uses this state to disambiguate the lexical ambiguity that would otherwise occur. A similar lexical ambiguity that is resolved by context is for the "`*`" operator. This can be the C multiplication operator or regular expression closure operator. Regular expressions, in Fig. 1, are parsed just like the other extension constructs and not enclosed in a C string.

Modular determinism analysis. Context-aware scanning thus makes it more likely that concrete syntax specifications of extensions will compose without lexical ambiguities or parse table conflicts, but it does not guarantee it. Such a guarantee is provided by Copper's *modular determinism analysis* which is run on extensions independently, by the extension's developers. It ensures that the composition of the host language and any set of other extensions that pass the analysis will have no lexical ambiguities and no parse table conflicts. The full details of this analysis can be found in the previous work of Schwerdfeger [2010]; Schwerdfeger and Van Wyk [2009], but the two primary conditions it checks for to provide this strong composition guarantee are described here. First, every extension-defined production with a host language nonterminal on the left-hand side must have a marking terminal as its first symbol on the right-hand side. After this, the extension is free to define its syntax. These marking terminals indicate the unambiguous transition (as a shift action) from host language to a particular extension's syntax. As we have seen in Section 3, in the case where these marking terminals do exhibit a lexical ambiguity, the programmer can directly resolve it in the specification of the extended language.

Second, every production in the extension that transitions back to the host language (by having host language nonterminals on its right-hand side) must not add any new terminals (except marking terminals) to the *follow set* [Aho et al. 1986] of any host language nonterminal. In Fig. 4, the `match` production contains an expression (`sc::Expr`) which is followed by a host language right paren and the `Clause_c` construct contains a statement followed by a right curly brace. This is allowed since these terminals are already in the follow sets of these nonterminal symbols. An interesting consequence of this rule is that syntactically rich languages with large follow sets are easier to extend.

The restrictions imposed by the modular determinism analysis are permissive enough to allow rather natural syntactic extensions, but can require more "punctuation", such as the curly braces, than we might prefer. An example of this is the curly braces around statements in match clauses. Without the curly braces in the `Clause_c` production `Stmt` would be followed by the terminals that begin a `Pattern`, thus adding new terminal symbols to their follow set and violating the restrictions. In Section 6 we will see some ways in which the host language syntax can be specified to avoid some of these concerns and allow for more flexible syntax in extensions.

Another implication of the restrictions of this analysis can be seen in the syntax of the Cilk extension. In the Cilk languages [Frigo et al. 1998], the `spawn` keyword was written before the name of the function being spawned, whereas in Fig. 1 one can see that this keyword, the marking terminal, is before the assignment statement and thus precedes, for example, `res_t1` instead of the function call. This is because the transformation of `spawn` down to plain C code involves not only the function call, but the variable into which the return value is copied. And thus the syntax for this extension must include both of these components. A second variation from the Cilk language is the need for the `cilk` keyword before the return statement. This is because the return statement also plays a role in the transformation of a Cilk function down to plain C, and we need to ensure that a special Cilk-specific version of this statement appears in the abstract syntax tree. (If the `cilk` keyword is omitted, the Cilk extension raises an error when the extended program is analyzed and translated to C.)

These illustrate points where the restrictions that ensure composability of independent extensions arise and affect the concrete syntax. The compromises made here seem reasonable, given that we now have a strong guarantee that there will be no parsing issues when the extensions are composed and used by the programmer.

Copper has a relaxed version of its modular determinism analysis that allows parse tables for extensions to be generated separately (thus supporting separate compilation) and composed with the host language later [Schwerdfeger and Van Wyk 2010], but we do not use that feature.

5 DEVELOPING SEMANTICS FOR ABLEC EXTENSIONS

In this section we describe how the semantics (*e.g.* type checking and C code generation) of language extensions are specified as Silver attribute grammars. The discussion highlights how the six expression problem criteria, $EP1 - EP6$, are satisfied by the ableC approach.

Attribute grammars [Knuth 1968] (AGs) have been extended with features like higher-order attributes [Vogt et al. 1989] and reference [Hedin 2000] / remote [Boyland 2005] attributes to result in a useful general-purpose programming model in their own right. They provide semantics for context-free languages by decorating abstract syntax tree (AST) nodes with semantic information. Attributes are associated with nonterminals in the underlying context-free grammar and equations associated with productions define attribute values. The semantics for composition of attribute grammars is straightforward by the union of the sets of nonterminals, productions, equations, etc. in different grammar modules. However, to satisfy the *separate compilation* criteria ($EP4$) of the expression problem this simple process is not used in Silver.

```
grammar org:A:ADT;
import edu:umn:cs:melt:ableC:abstractsyntax;

abstract production match
s::Stmt ::= sc::Expr cs::Clauses
{ e.unparse = "match (" ++ sc.unparse ++ ")"
               ++ "{" ++ cs.unparse ++ "}" ;
  sc.env = e.env;
  cs.env = e.env;
  cs.scrutineeType = sc.typerep;

  forwards to if null( sc.errors ++ cs.errors )
    then stmtSeq ( mkInit(sc), cs.transform )
    else errorExpr( sc.errors ++ cs.errors );
}
```

Fig. 5. Attribute grammar fragment defining part of the algebraic data type language extension.

First, we consider the abstract syntax production `match` in Fig. 5. The first two lines indicate that this is part of the `ADT` extension and builds on (`imports`) the abstract syntax of the host language. This production would be used by the semantic action of the `match` concrete syntax production in Fig. 4 to construct ASTs for match statements. It has equations defining values for two host language–declared attributes. The first is the *synthesized* attribute `unparse`, which constructs the text representation of the program and propagates this information up the syntax tree. The second, `env`, is an *inherited* attribute that propagates the environment (symbol table) that associates names with their declarations and types down the tree. The `forwards to` clause, discussed in more detail below, specifies the translation of the `match` construct down to plain C code if there are no errors detected on the scrutinee (sc) or on the clauses (cs) or, if there are errors, to special *error* node that reports such errors.

This extension also declares a new inherited attribute `scrutineeType` that decorates `Clauses` nonterminals and is used by clauses in a match statement to ensure that the type of the pattern is the same as the type of the scrutinee `sc`. If this is not the case, extension clause productions generate error messages describing the errors in the patterns and pass them up the AST in the `errors` attribute. This shows how extensions can generate error messages that refer to code written by the programmer and *not* in terms of the generated host language C code. Extensions can also perform non-trivial analysis to detect errors that cannot be detected (at least not easily) in the generated C code. For example, the match construct can also check that the patterns in the clauses are exhaustive. The extension that adds syntax for SQL queries, described in a technical report [Kaminski et al. 2017], uses the database schema to check for type errors in SQL queries written in an extended C program.

During attribute evaluation the equations compute values for attributes on the AST. An AG is *effectively complete* if equations exist so that needed attributes can be computed for all possible syntax trees. That is, no required equations are missing. Knuth defined a static analysis to check for AG completeness [Knuth 1968], and extensions such as higher-order attributes extended this analysis [Vogt et al. 1989] accordingly. This analysis (along with traditional typing rules for operations on primitive values such as integers and strings) provide strong static typing of attribute grammars (*EP*2).

```
grammar com:B:cilk;
imports edu:umn:cs:melt:ableC:abstractsyntax;

synthesized attribute slow_clone<a> :: a;
attribute slow_clone<Stmt> occurs on Stmt;
attribute slow_clone<Expr> occurs on Expr;

aspect production while    (note aspect not abstract)
s::Stmt ::= cond::Expr body::Stmt
{ s.slow_clone = while ( cond.slow_clone,
                         body.slow_clone );
}
abstract production sync
s::Stmt ::=
{ s.slow_clone = ...
  forwards to ... call to CILK2C_AT_SYNC_FAST() ... ;
}
```

Fig. 6. Attribute grammar fragment defining part of the Cilk language extension.

While the `match` production shows how new variants are added to the host language, the grammar in Fig. 6 shows how new operations can be added, in the form of new attributes on existing productions. This illustrates how attribute grammars satisfy the extensibility in "both directions" criteria (*EP*1). The original Cilk language translated down to plain C code, and in doing so generated two versions of each `cilk` function: a fast and slow clone that coordinate the stealing and distribution of work across all available processors. The slow clone is created using the `slow_clone` attribute, a higher-order polymorphic attribute that creates the syntax tree of the slow clone. The type of the attribute is the same as the nonterminal type of the node it decorates (when it occurs on a `Stmt` it is a tree of type `Stmt`). The `aspect` production for the host language `while` production has the equation for creating the slow clone code for a while loop. Similar aspect productions exist for all host language productions, though some naturally do more than rebuild the tree as is done here. Aspect productions allow new equations to be associated with existing productions defined elsewhere in this grammar or in an imported one.

Names (for productions, attributes, etc.) in Silver have a short representation that is used in the specifications in this paper, but also have a "fully qualified" representation that is used internally in

Silver when checking names against one another. These fully qualified names are similar to the fully qualified names in Java that are based on the name of the defining package. Similarly the name of the Silver grammar (based on the developers Internet domain name) containing a declaration is used to form the fully qualified name. Thus there are no possible name clashes between different Silver grammars and two extensions cannot, for example, introduce productions that define new types or other constructs that have the same fully-qualified name. While their names used inside a grammar module may be the same, they are distinguished by their full names. If a grammar needs to refer to two names with the same short representation then the fully qualified name can be used. But in Silver specifications that combine grammars there is typically no need to directly refer to these elements and thus no need to write fully qualified names.

To satisfy the automatic composition criteria (*EP*6), ableC uses the Silver feature called *forwarding* [Van Wyk et al. 2002]. This is how, for example, the `slow_clone` attribute (new analysis) is computed for the `match` construct (new syntax). Forwarding allows a production such as `match` to specify a computation that would produce a *semantically equivalent* tree composed of host language constructs. For the `match` construct in Fig. 5 this is the sequence of some initializing statements that refer to the expression being matched against (`mkInit(sc)`), followed by the transformation of the clauses (computed in the attribute `transform`) into a nested sequence of if-then-else statements that implement the pattern matching. Queries for attributes without a defining equation are passed to the *forwarded-to* tree and their values are computed there. Thus, the `slow_clone` attribute for `match` is computed on the host AST that it forwards to, and these host productions have equations for `slow_clone` specified in aspect productions in the Cilk extension, as in Fig. 6. Similarly, the `match` production does not explicitly define the synthesized `errors` attribute but relies on forwarding to get a value for this attribute, either from the list of errors on the `errorExpr` node (which is just the list provided as its single argument) or the empty list of errors from the `stmtSeq` node. For the Cilk `sync` statement in Fig. 6 this is a call to the Cilk run-time. Thus new analyses need only specify equations for the host language, and when computing that analysis on any new syntax (that is, new productions from a different, independent extension) the analysis can always be computed on the forwarded-to tree, rather than directly on the extension node.

Extensions can do sophisticated analysis and transformation over their syntax to compute their forwards-to trees. For example, in our initial regular expression extension implementation we translated regular expression literal (*e.g.* on line 13 of Fig. 1) to values of type `regex_t` in the POSIX regular expression library. This required simply generating the string representation of the literal. In another version, the abstract syntax of the regular expression does the standard transformation into a non-deterministic finite automata and then into deterministic finite automata that is represented as a C `struct` with the expected transition table, list of final states, etc.

Note that Zenger and Odersky's independent extensibility criteria (*EP*5) is also satisfied by this approach. In this example, the programmer can choose to use just one or both extensions. The ADT and Cilk extensions are independent, but the regular expression extension explicitly and intentionally depends on the ADT extension and extends its notion of patterns. It imports both the host language and ADT grammars. While the primary contribution of the ableC approach is to support independently-developed extensions, explicit dependencies on other extensions is supported and discussed further in Section 8.

The *no modification of existing code* criteria (*EP*3) is demonstrated by the use of Silver `imports` statements, both by extension specifications (Figs. 5 and 6) and in the composition specification written by the programmer in the DSL that composes the host language and extensions (Fig. 3).

Finally, although forwarding allows extensions to compose, satisfying *EP*6, the *separate compilation and typing* criteria (*EP*4) is provided by Silver's type checking mechanisms [Kaminski and Van Wyk 2011] and by its *modular well-definedness analysis* [Kaminski 2017; Kaminski and

$$\begin{array}{lcllll}
E & ::= & E \text{ `+' } T & \mid & E \text{ `}-\text{' } T & \mid \; T \\
T & ::= & T \text{ `}*\text{' } F & \mid & T \text{ `/' } F & \mid \; F \\
F & ::= & \text{`(' } E \text{ `)'} & \mid & var &
\end{array}$$

(a)

$$\begin{array}{lcllll}
E & ::= & E \;\; AddOp \;\; T & \mid & T \\
AddOp & ::= & \text{`+'} & \mid & \text{`}-\text{'}
\end{array}$$

(b)

$$\begin{array}{lclllll}
E & ::= & E & AddOp & ETL & \mid & ETL \\
ETL & ::= & ETL & ETLOp & ETL & \mid & ETR \\
ETR & ::= & ETN & ETROp & ETR & \mid & ETN \\
ETN & ::= & T & ETNOp & T & \mid & T \\
T & ::= & T & MulOp & TFL & \mid & TFL
\end{array}$$

(*ETL* (left associative), *ETR* (right associative), and *ETN* (non-associative))

(c)

Fig. 7. Evolution of expression grammar from traditional (a) to allowing infix operators as existing precedence/associativity levels (b) to allowing infix operators at new precedence/associativity levels between host language ones (c).

Van Wyk 2012]. This analysis enforces certain structural properties on the relationship between modules containing certain kinds of declarations to exclude the possibility of having two different equations for a single attribute for a single production. They also ensure syntax extensions actually make use of forwarding, to satisfy criteria *EP*6. It further ensures that the *flow types* of host language nonterminals do not change. Flow types track what inherited attributes may be used to compute a synthesized attribute. For example, the fact that the `env` attribute is used to compute the `errors` attribute is recorded in the flow type for expressions, statements, and other host language nonterminals. This restriction ensures that, in addition to always having a defining equation for synthesized attributes, the attribute grammar will always have a value for each necessary inherited attribute.

This combination of Silver features such as its module system, forwarding, and its modular well-definedness analysis are used to satisfy the 6 expression problem criteria, *EP*1 — *EP*6, given in Section 2.2.

6 ENABLING ADDITIONAL SYNTACTIC EXTENSIONS TO ABLEC

We now turn our attention to the role of host language developer and ways that concrete syntax can be specified to allow a wider class of extensions to pass the modular determinism analysis in Copper. The techniques discussed here, and in the next section on semantics, are applicable to other extensible host languages besides ableC.

Grammar refactoring for new infix operators. The modular determinism analysis requires extension productions with a host language nonterminal on the left-hand side to begin with a marking terminal. For the Cilk and ADT examples in Fig. 1 this seems a reasonable restriction and required only minor adjustments to the Cilk syntax. But this prevents adding new infix operators, such as the regular expression matching operator =~ on line 26, when productions for infix operators use the common LR-style with the operator terminals between expression nonterminals, as illustrated in Fig 7(a). (We abstain from using Silver syntax here as it is a bit verbose and also use short single-letter names for some nonterminals.) Attempting to add a new infix operator, say @, with the same precedence and associativity as addition (+) would entail writing this new production that does not begin with a marking terminal, that is $E ::= E \text{ `@' } T$.

To allow such a new operator, we refactor the grammar to give infix operators (at each precedence level and associativity) their own nonterminals, as shown in Fig. 7(b). Now, the new infix operator can be added in the extension grammar using the new production $AddOp ::= \text{`@'}$. This refactoring requires no change to how the abstract syntax is represented. We can go a step further and refactor the host language grammar to allow new infix operators with either left, right, or no associativity at precedence levels *between* those of existing operators. To allow such operators at the precedence between additive operations (derived from E) and multiplicative operations (derived from T), we can add three new nonterminals between E and T in the grammar for each type of associativity, as shown in Fig. 7(c). Extension writers can then add new productions for these *ETLOp*, *ETROp*, and *ETNOp* operator nonterminals just as was done for *AddOp* above. This technique is used to allow for the new infix operator "=~" in the regular expression extension.

This pattern is used throughout the ableC grammar for all infix operators. Admittedly, writing the expression grammar in this way is rather tedious. However, the strong composability guarantees that we have here make this a small price to pay. It is only an up-front cost to the host language developer, but has many benefits for the end user programmer.

7 ENABLING ADDITIONAL SEMANTIC EXTENSIONS IN ABLEC

This section describes various features in the specification of the host language abstract syntax and semantics of ableC that allow for additional classes of language extensions. All of these enrich the host language by providing semantic extension points or adding host language constructs that go beyond what is needed in a monolithic implementation of C. One simple example of these is the inclusion of the GNU C Compiler extension for statement-expressions which has the concrete syntax of `({` *statement-list* `;` *expression* `; })`. This frequently-used construct executes its statements before evaluating the final expression as its value. For example, a pattern in a `match` statement is translated to an if-then-else condition that uses this to assign values to pattern variables that are then used in the statement part of the clause before evaluating to true or false to indicate a pattern match.

Below we first describe operator overloading, which provides, as an alternative to parsing, a second way to introduce extension productions into the abstract syntax tree. Next, we describe how language extensions can specify non-local transformations and lift new declarations up global-scope of the program, and then describe how the extensible type qualifiers of Foster et al. [1999] are incorporated into ableC. We conclude with a discussion of an extensible environment and mechanism for performing substitutions on abstract syntax trees.

7.1 Operator Overloading

Many languages such as C++ and Python, but not C, allow operators to be overloaded to apply to new types of values. The abstract syntax of ableC is structured so that extensions can overload host language operators for types that they introduce. These new extension types may be associated with syntax for value literals and semantic analyses for domain-specific error checking, thus providing benefits beyond what is available in library-based operator overloading in the languages. But for operator overloading, no new concrete syntax is added, instead, the overloading mechanisms of ableC create the extension abstract syntax.

As an example, consider an extension that provides arithmetic over integer intervals. An *interval* type and value constructor are the only syntactic extensions, but it overloads arithmetic operators. The following fragment negates the first interval, adds the result to the second, and assigns to `x` the interval range from 4 to 8:

```
interval x;        x = - interval[1,3] + interval[7,9];
```

In scanning and parsing this example, it is concrete syntax in the ableC host language that recognizes the unary minus (`-`) and addition (`+`) operators and creates the initial AST for them. For unary minus expressions the so-called *dispatching* production `unaryMinusDispatch` is used, see Fig. 8. This production has a so-called *collection* attribute named `overloads` that is a list of pairs of extension names and productions that may overload this operator. It is initialized to the empty list

```
grammar edu:umn:cs:melt:ableC:abstractsyntax;

abstract production unaryMinusDispatch
um::Expr ::= e::Expr
{ production attribute overloads :: [ Pair<String, Expr ::= Expr> ];
  overloads := [];

  local attribute overloadExpr :: Maybe<Expr>;
  overloads = lookup (extName(e.typerep), overloads);

  forwards to case overloadExpr of
                | just(fwrd_prod) -> fwrd_prod(e)
                | nothing() -> unaryMinus(e) ;
}
```

Fig. 8. Operator overloading: Dispatch production for unary minus.

using the := initial-value operator instead of the definition operator =. This production looks in its `overloads` attribute for a pair with the same extension name as is found on the type (`typerep`) of the child expression `e`; `lookup` returns `nothing()` if no match is found, if one is found it returns the production wrapped in a `just` production. These implement a type similar to `Maybe` in Haskell or `option` in ML dialects. If no overloads are found, `unaryMinusDispatch` forwards to the host language `unaryMinus` of `e`. Otherwise it uses the production found in the `overloads` attribute (`fwrd_prod`) and uses it to construct the tree to forward to.

In Fig. 9 the interval extension specifies that it overloads unary minus by using an `aspect production` for `unaryMinusDispatch` to add the pair containing its name and the production that implements unary minus on intervals (`unaryMinusInterval`). This is done using the `<-` contribution operator instead of the `=` or `:=` operator. The production `unaryMinusInterval` may do additional error checking and will then forward to the C language implementation of interval unary minus. (These equations are elided since our concern is how this production is added to the AST without any new concrete syntax.) Thus, `unaryMinusInterval` is used just like other abstract productions are. The only difference is that it is not the semantic actions in the parser that introduce it, but this operator overloading mechanism. Note that the type of second pair element in `overloads` is the same type as the unary minus dispatching production and the overloading interval unary minus production.

```
grammar org:D:interval;
import edu:umn:cs:melt:ableC:abstractsyntax;

aspect production unaryMinusDispatch
um::Expr ::= e::Expr
{ overloads <- [ pair( "org:D:interval",
                        unaryMinusInterval ) ];
}

abstract production unaryMinusInterval
um::Expr ::= e::Expr
{   ... equations and forwards-to specification...
}
```

Fig. 9. Operator overloading: Extension fragment for overloading unary minus.

ableC supports overloading of many host language operators including array access, function call, field access, using this technique, or a slight variation of it. For binary operators the "double dispatch" problem is solved by first checking if an extension overloads the operator for both types,

```
abstract production while
s::Stmt ::= cond::Expr  body::Stmt
{ s.host = while(cond.host, body.host);
  s.lift = while(cond.lift, body.lift);
  s.gDecls := cond.gDecls ++ body.gDecls;
}
```

(a)

```
template<a, b> struct pair  a fst; b snd; ;
...
int main() {
   template pair<int, float> p;
   ...
}
```

(b)

Fig. 10. (a) Specification of host and lifting attributes for while-loops. Other constructs follow the same pattern. (b) Example use of the template extension that relies on this declaration lifting mechanism.

then checking just the left type, and finally the right type. Three separate collection attributes are used for this. The vector extension described in a technical report for this paper [Kaminski et al. 2017] overloads several of these operators. In addition, extensions that introduce new operators, such as the infix operators discussed previously, can use this technique so that they can be overloaded by other extensions that build on them.

7.2 Non-local Transformations, Lifting Declarations

Part of the reason that ableC is able to solve the extended expression problem described in Section 2.2 is that each extension construct, such as the `match` statement or `cilk` function, forwards to its C-code implementation. This can be seen as a local transformation, although the extended construct is not *replaced* in the AST by what it forwards to but just links to it so that attributes can be retrieved from it during attribute evaluation.

But for some extensions this can be limiting as it is not uncommon for extensions to need to lift new declarations up to the global scope of the program. For example, an extension that introduces λ-expressions (closures) needs to lift the function that implements it up to the global scope. Thus we have added the ableC host language production `injectGlobalDecls` with the signature `Expr ::= Decls Expr` that expressions such as a λ-expression can forward to. The first child is the declarations that are to be lifted; the second is the expression that implements the extension. In the case of the λ-expression the list of declarations contains the implementing function and generated `struct` to implement the environment containing values of free variables. The expression initializes this `struct` and then evaluates to a reference to the lifted function declaration.

This `injectGlobalDecls` construct is in the ableC host language but is not an actual C construct that the vendor C compiler (that compiles the generated host language C program to produce an executable) will recognize and thus it must be translated away by ableC. This is done using two higher-order synthesized attributes that are similar to the `slow_clone` attribute seen previously. The first is `host`. It is defined for all host productions that constitute the abstract syntax of the language and `injectGLobalDecls`. For example, the `host` attribute on the `while` loop production, see Fig. 10(a), simply reconstructs the tree with its host children. Extension productions do not define this attribute and thus the computation of the `host` attribute occurs only on host language productions.

The main driver function, implementing the process described in Fig. 2, gets the `host` attribute off of the AST constructed by the parser generated for the extended language. This tree contains no extension constructs but will contain `injectGlobalDecls` constructs. From this host tree, the driver process gets the `lift` attribute, which computes the tree in which declarations propagated up the syntax tree in the `gDecls` attribute are inserted in the syntax tree above the globally-scoped declaration in which they occur. An invariant maintained by these two attributes is that all declarations in the host tree occur either in lift or in gDecls. Fig. 10(a) shows these attributes for the while-statement.

One extension that utilizes lifting is the template extension, which provides C++-style templated types and functions to be declared and instantiated. For example, a programmer can use this extension to implement a templated struct defining a generic pair, as shown in Fig 10(b). This requires the instantiated version of the pair struct to be lifted to the global level. Note that writing the same instantiation of pair in multiple places should not result in the same struct being lifted to the global score multiple times. It is the responsibility of the extension writer to ensure that duplicate declarations are not created; a declaration passed to injectGlobalDecls is wrapped up in a utility production that first checks the environment to see if the declaration already exists, before forwarding to the empty declaration or the declaration that was intended to be lifted. To enable this, new definitions from gDecls being passed up the tree are inserted into the environment at the global scope for all points in the tree to the right of where the new declaration was injected. Note the second use of template to instantiate a pair struct with int and float in Fig 10(b). C++ does not require template in this location, but since extensions need to begin with a marking terminal, our implementation of templates requires this extra use. This is another example of the adjustments to extension syntax that extension developers must sometimes make to ensure that their extensions will compose with others. More details about the template extension may be found in the technical report for the paper [Kaminski et al. 2017]. C++ templates are very complex and we have not attempted to implement them in their entirety.

Note that this can be generalized to lift declarations to different enclosing scope levels. For example, it could be used in an extensible specification of Java to lift declarations up to the enclosing method, class, or package level.

7.3 Extensible Type Qualifiers

Foster et al. [1999] introduced an elegant and extensible notion of type qualifiers to support a simple notion of subtyping. As an example, consider the C type qualifier const; a value of type int * can be used anywhere that a value of qualified type const int * is expected, but not vice versa. Annotating a type with const thus indicates a supertype; they call this a *positive* qualifier. In contrast, a pointer qualified as nonnull (added as an extension, to indicate that the pointer is not null) can be used anywhere that an unqualified pointer is expected, but not vice versa; this is a *negative* qualifier. In a discussion of practical considerations of extending a language with new type qualifiers, Foster et al. [1999, Section 2.5] assert that the changes to concrete syntax are minimal and could be easily implemented using well-understand techniques to avoid ambiguity. The extensibility and non-ambiguity guarantees of ABLEC make it well-suited for providing support for these kinds of qualifier extensions.

Syntactically, adding extensions like this is simple as the concrete productions consist of simply the marking terminal (*e.g.* nonnull) on the right hand side. These construct a simple AST of host language nonterminal type TypeQualifier. In the initial design of ABLEC, type compatibility involving type qualifiers was done in an ad-hoc manner based on the semantics of the qualifiers built into C. This was generalized to support arbitrary type qualifiers by adding a Boolean isPositive attribute to TypeQualifier that the extension production defines to indicate if a qualifier is positive or negative. Additionally, the type compatibility function was changed so that to determine if type τ_1 is a subtype of type τ_2, it checks that the positive qualifiers on τ_1 is a subset of the positive qualifiers on τ_2, and that the negative qualifiers on τ_2 is a subset of the negative qualifiers on τ_1.

The subtyping semantics of the type qualifiers in C, which consist of const volatile, and restrict, have been implemented in the host language in the manner described. The semantics of const additionally disallows non-initial assignments. This is done by generating an error on the assignment production if the type qualifiers on the type of the assigned-to expression contains the `const` qualifier.

One type qualifier that we have implemented as an extension is `nonnull`. The new production for `nonnull` simply consists of the marking terminal, and constructs an AST of type `TypeQualifier`. The subtyping semantics of `nonnull` are obtained from the checks added to the host language as discussed above. Like `const`, the semantics of `nonnull` extend beyond subtyping. In order to statically detect possible null-pointer dereferences, we may wish to require any pointer that is dereferenced to be annotated as `nonnull`. This is easily accomplished by an aspect production for the pointer-dereference operator that generates an error if the list of qualifiers on the type of the expression being dereferenced does not contain `nonnull`. (This is similar to the aspect production for `unaryMinusDispatch` added by the interval extension on the left in Fig 9.) There are many practical uses of type qualifiers in addition to the few mentioned here. Another extension we have developed is based on the work of Dietl et al. [2011] and utilizes type qualifiers to implement type-safe enumerations like those in C++ (but not C).

Extensible type qualifiers differ from other kinds of extensions in that new checks are performed that may cause previously error-free code to no longer be allowed. This presents some difficulty when using new type qualifiers with libraries that do not use new qualifiers like `nonnull` (and is addressed by previous work [Shankar et al. 2001]) and, as in our case, with independently-developed extensions that may generate (that is, forward to) host language code that does not use new extension-introduced qualifiers. If soundness is required of the semantics of type qualifiers — *e.g.* if every possible dereference of null should be caught — then a certain number of false positives is unavoidable. For example, code generated by an extension that is developed independently of the `nonnull` extension may declare and deference pointers, but has no way to annotate such pointers as `nonnull`. Abstract syntax tree nodes for constructs such as expressions and statements have annotations that indicate their location. This location is structured data, constructed by a `loc` production to indicate a position in a source file or by a `generatedLoc` production to indicate that the construct was generated during forwarding or some other transformation process. This location information can be used by the aspect production for `dereference` to check if the location indicates that the dereference was generated, and if it was it does not produce an error message if the expression to be dereferenced is not qualified by `nonnull`.

The system of type qualifiers that we have added to ABLEC is flow-insensitive. Foster et al. [2002] expand the usefulness of type qualifiers with a notion of flow-sensitivity, allowing qualifiers to be inferred where possible and to differ at each program point. Adding support in ABLEC for flow-sensitive type qualifiers is future work. Type qualifiers in ABLEC can also introduce rich forms of concrete syntax. For example, a qualifier for specifying physical units can include expressions over SI measurement names; for example a type can be qualified by `units(kg*m^2)` to indicate a measurement in newtons. Additionally qualifiers can inject dynamic code checks in static analysis is infeasible or not appropriate for the situation; for example, a runtime check comparing a pointer to null before dereference can be generated where a static check is not possible. They can also inject code for tasks such as logging or automatic conversion of data. Carlson and Van Wyk [2017] provide a more complete discussion of the capabilities of ABLEC type qualifiers.

7.4 Extensible Environment and Substitution Mechanisms

Another feature that extension writers may need is the capability to add additional namespaces to the environment. This is used in the template extension where templated declarations must be able to be looked up to find their parameters and the declaration to substitute. This can be implemented by structuring the environment itself as a tree of definitions, and allowing extensions to add new attributes to represent contributions from new namespaces.

One pattern that may occur for extension writers is to be able to fill in ‘holes’ in a pre-constructed AST. This is the case for the template extension, where a function or type expression must have names substituted for types. ableC also provides, as a utility to extension writers, a way to parse strings into ASTs, which may have names which must be replaced with sub-trees. To allow this, ableC provides as a utility for extension writers a set of attributes which can be used to transform the AST into a substituted version.

8 EXTENSIONS MAY USE OR EXTEND OTHER EXTENSIONS

While the focus on this work is on independently developed language extensions and how they can be automatically and reliably composed, this section describes how language extensions can intentionally depend on other extensions: either to make use of other language extension constructs in their translation or to extend the syntax introduced by some other extension. An example of this first type of dependence can be seen in an extension that adds a simple vector type and overloads the addition operator + for concatenating vectors, as seen in the example below and described in more detail in the technical report for the paper [Kaminski et al. 2017]:

```
    vector<int> a = ...;    vector<int> b = ...;    vector<int> c = a + b;
```

The vector concatenation production forwards to a call to a template function that implements concatenation. Here the vector extension imports the template extension so that it may consider the composition of ableC and the template extension as the “host” language that must forward to.

The second type of dependence is more interesting and can be seen on line 34 of Fig. 1 where the regular expression extension extends the concrete (and abstract) syntax of the pattern language derived by the `Pattern_c` (respectively, `Pattern`) nonterminal defined in the ADT extension. The regular expression `Pattern` production forwards to the pattern expression (shown in concrete syntax form here) “`v where rxmatch(v,` *trx*`) }`” where `v` is a pattern variable that always matches the string scrutinee (in this case the string in the `Leaf` constructor), *trx* is the translation of the regular expression literal to the C host language, and `rxmatch` is a function provided by the regular expression extension that evaluates to true if *trx* matches the string. This value is what determines if the `when` clause will cause the pattern to match or not.

9 RELATED WORK

There are many tools and techniques for extending programming languages and compilers. Below we discuss various categories of these tools, focusing on those most closely related the work presented here and how these address the extensible language challenges discussed in Section 2.2.

Traditional Compilers. These are usually constructed to allow compiler engineers to add new analyses and optimization passes. Examples of these include the Rose compiler framework [Quinlan 2000], the Glasgow Haskell compiler (GHC), Cil [Necula et al. 2002], and LLVM and its associated C compiler, Clang. While some of these even allow programmers to turn on or off various integrated extensions, they do not support extensions that introduce new syntax in any meaningful way beyond directly modifying the compiler nor provide a solution to the expression problem.

Extensible language frameworks. There are several language tools that similarly use declarative formalisms such as context-free grammars, attribute grammars, or term rewriting rules. These include attribute grammar approaches such as JastAdd [Ekman and Hedin 2007b] and Kiama [Sloane 2011], term-rewriting systems such as Stratego [Visser 2001] and Spoofax [Kats and Visser 2010], and general meta-programming systems such as Rascal [Klint et al. 2009]. While these nicely support the modular specification of language features, they generally lack some notion similar to forwarding to satisfy the automatic composition criteria (*EP*6). Most modern attribute grammars do not use a well-definedness check and thus do not satisfy the separate compilation and type checking criteria (*EP*4). In contrast, JastAdd does statically check that grammars are well-defined, but after grammar modules have been composed.

Of particular interest is the work in using attribute grammars for the modular development of languages [Adams 1993; Farrow et al. 1992; Ganzinger 1983; Kastens and Waite 1994; Saraiva and Swierstra 1999], but this work is primarily aimed at helping compiler engineers reuse language specifications and provide no analyses like Silver's modular well-definedness analysis. For example, the JastAdd [Ekman and Hedin 2007b] attribute grammar system, in which reference attributes were developed, also supports higher-order attributes [Vogt et al. 1989], and circular attributes. It was used to specify the ExtendJ extensible Java compiler [Ekman and Hedin 2007a] in which an initial Java 1.4 specification was extendend to Java 1.5, 6, 7, and 8. This demonstrates JastAdd's good support for modular language extensions but, as mentioned above, checks for well-definedness are done after grammar modules have been composed.

The Spoofax [Kats and Visser 2010] language workbench is a similar system but uses Stratego-based term-rewriting and a variety of domain specific languages for specifying language syntax and semantics. It uses scannerless generalized LR (SGLR) parsing, and thus can generate a parser for any context-free grammar, but this approach provides no guarantees that the composed grammar is not ambiguous. However, by eschewing a traditional scanner and parsing down to the character level, SGLR can handle overlapping keywords in a manner not unlike a context-aware scanner. SugarJ [Erdweg et al. 2011] is an extensible specification of Java built using Spoofax that aims to provide language extensions as libraries that are simply imported into a program. SGLR parsers are constructed when the extended program is compiled, and are cached to avoid rebuilding on every compilation. While the library-based model with runtime composition is a major advantage of SugarJ, it suffers the same drawbacks mentioned for the Spoofax language workbench.

XTC [Grimm 2006] and Xoc [Cox et al. 2008] are two notable extensible C compilers. XTC supports modular specification of syntactic extensions to C using parsing expression grammars (PEGs). These are similar in form to context-free grammars but compose using *ordered-choice* for productions with the same left-hand side. These are difficult to reason about since ordered-choice silently removes all ambiguities. Though unlikely, this does present the possibility for one extension's syntax to elide the syntax of another. ableC shares the same goals of programmer-directed composition of language extensions as Xoc [Cox et al. 2008], another extensible C compiler. Xoc uses a GLR-based parser, so extension specifications have a form similar to those in ableC. But Xoc provides no assurances that the extensions will reliably compose and grammar ambiguities are presented to the programmer as a syntax error. Xoc uses an attribute grammar–like mechanism for semantic analysis but neither it, nor XTC, support something like forwarding nor satisfy all of the expression problem criteria *EP*1– *EP*6.

There are other instances of extensible languages constructed using Copper and Silver. But both ableJ [Van Wyk et al. 2007] (for Java) and ableP [Mali and Van Wyk 2011] (for Promela) predate the modular well-definedness analysis (ableJ also predates the modular determinism analysis) and thus do not attempt to find the sweet-spot of extensibility that ableC does and thus many of the problems faced by ableC are not addressed in those earlier works.

Projectional editing systems. MPS [Voelter 2011] and Intentional Programming [Simonyi et al. 2006] are extensible language systems that avoid the need for a parser by using a special editor that lets one alter ASTs directly, even though it looks as if one is editing a text file. While this avoids the challenges of building composable parser specifications, it does lock the programmer into a specific editing tool, whereas ableC allows programmers to use the text editor of their choice. mbeddr [Voelter et al. 2012] is an extensible C translator built using MPS that adds a number of extensions for building embedded systems. It does not offer a solution to all of the expression problem criteria, instead opting for an object-oriented solution, and so adding new analyses may involve changing the host language implementation. Intentional Programming, on the other hand, does satisfy the automatic-composition criteria (*EP*6) since forwarding was originally specified in Intentional Programming. But it does not satisfy the separate type checking criteria (*EP*4).

Additional approaches. Macros systems (traditional, hygienic, etc.) [Weise and Crew 1993] and embedded domain specific languages [Hudak 1996] allow new language constructs to be added but are limited to the existing syntactic forms of the language. Perhaps the most interesting system with macros is Racket [Flatt 2010] which supports hygienic macros [Kohlbecker et al. 1986] and builds a type system on top of an underlying dynamic language through macros (called Typed Racket), demonstrating the semantic analysis capabilities of its macro system. However, it and other macros systems do not offer a solution to all of the expression problem criteria. For embedded DSLs, new analysis is restricted to what can be embedded into the host language's existing type system, and these embeddings typically result in difficult to understand error messages.

The Delite project uses Scala as a host language for embedding domain-specific constructs and lightweight modular staging [Rompf and Odersky 2010], a type-based approach to multi-stage programming, to analyze and optimize DSLs embedded in Scala. Delite comes with no ready solution for the expression problem, but the approach is potentially flexible enough to simply adopt by convention one of Scala's solutions for the problem. While this approach avoids an external translator such as ableC, it does require a sophisticated host language that supports multi-stage programming.

Rendel et al. [2014] explore the relationship between object algebras and attribute grammars in the context of the expression problem. A technique is presented for encoding L-attributed grammars as object algebras in Scala, and claims of the scheme's modularity are supported by a case study in which a monolithic compiler is made extensible by translating it into the encoding. Although *EP*4 is addressed in the sense that attributes can be defined and type-checked separately, it is only at composition-time that errors are raised; without a modular well-definedness analysis there is no guarantee that independently-developed algebras will compose error-free.

There are many solutions to the expression problem in the programming language literature, *e.g.* the very direct Scala solution by Wang and Oliveira [2016]. The composition process here, and in similar approaches, involves writing new classes that thus have new constructor method names that build the AST. This presents a problem here since extensions need a known name to use to construct host language ASTs to forward to.

Safe extensibility. While most extensible language frameworks seems to value expressiveness over reliable composition, not all do. Wyvern [Omar et al. 2013, 2014] and VerseML [Omar 2017] are the only extensible language systems besides ableC, to our knowledge, that supports *reliable* composition of independently developed language extensions, at least syntactically, without abandoning parsing in favor of projectional editing. It uses a white-space sensitive parsing technique that uses indentation to isolate extension fragments to be parsed by a language extension. Or alternatively, Wyvern extension fragments can also be wrapped in balanced braces, parentheses or quotations which ensure the same isolation constraint. Wyvern uses a bi-directional type system to determine the types of these fragments and then chooses the parser to use based on the type. This parser runs after the initial parsing step which skips the language extension fragments. The aim here is to isolate the independently-developed language extensions and is similar in intent to the goals of Copper's modular determinism analysis. The major difference is that Copper distinguishes extensions syntactically, where Wyvern leverages type information to do so. Since this type-based distinction happens during program compilation, and not during the composition of extensions as done by Copper, it is more flexible than what is used in ableC. While Wyvern ensures extension syntax can be composed safely, it does not accommodate introducing new semantic analysis of the host language and thus also does not satisfy the expression problem criteria.

10 CONCLUSIONS, DISCUSSION, AND FUTURE WORK

10.1 Conclusions

In this paper we have described ableC, an extensible language framework for C, and shown several composable language extensions for it. These extensions all satisfy the requirements of the modular determinism analysis (MDA) and the modular well-definedness analysis (MWDA). Doing so ensures that the composition of these, and other extensions that satisfy the analyses, will form a language specification that is unambiguous and complete. A primary contribution of the paper is to demonstrate that the restrictions imposed by MDA and MWDA in a mainstream general purpose host language are not onerous. While they impose some restrictions on language extensions in order to provide certain guarantees on the composability of independently-developed language extensions, they still allow rather expressive and useful extensions to be specified. Additional examples of language extensions can be found in the technical report for the paper [Kaminski et al. 2017] and on the ableC website[2]. ableC is available under an open source license.

Most of the extensions we have developed, such as the Cilk and algebraic datatype extensions, required little forethought in the design of the C host language Silver specification. However, we have found that structuring the host language in certain ways and adding some extension points in the grammar do allow additional kinds of extensions to be made. For example, in Section 6, we refactored the concrete syntax specification of C to allow new infix operators to be added. This did not change the language, but did change the results of the modular determinism analysis to allow new infix operators. We also added, in Section 7.2, a mechanism for lifting new declarations from an extension use point to the global scope of the program.

10.2 Discussion

The goal of the modular analyses is to ensure that the composed translator not terminate abnormally due to unforeseeable interactions between extensions. Extensions can have bugs that, for example, divide by 0 or take the head of an empty list and thus cause the translator to abort. Similarly there are unenforced restrictions that extension writers use their Internet domain name in naming their grammars and that their extension passes the modular analyses. Another unenforced restriction is that extension writers do not claim to overload operators for host language types or extension types that they do not define. These are not enforced but violating these restrictions and assumptions is easily detected and the offending extension is easily identified. The aim is that extensions writers acting in good faith can avoid composition errors that the programmer composing the extension is not prepared to handle.

One question to ask about this work is the following: if there are additional extension-enabling refactorings to make or facilities to add to ableC how would these changes to the host language affect existing language extensions? Would existing working extensions now fail the modular analyses with the new host language? For example, we are in the process of adding attributes to construct program flow graphs to perform control and data flow analysis. This will likely add additional host language inherited attributes potentially causing extensions to fail the modular well-definedness analysis. There is no complete solution; problems with software evolution affect all software, but there are some observations to make. First, we note that some of the features added in Sections 6 and 7 would not have affected pre-existing extensions. For example, adding the non-local lifting transformations in Section 7.2 would not change the flow of attributes in a way that would invalidate extensions that already pass the modular well-definedness analysis. Second, we do anticipate that with more experience, the features in Sections 6 and 7 will form a hopefully small set of "best practices" for designing extensible languages — not an unending slippery slope of modification. We show these new features to illustrate how host language design can impact extensibility. Finally, even though this approach may not be able to avoid breaking changes to programming languages that do occur (*e.g.* migrating from Python version 2 to 3) this extensible approach can smooth out the evolutionary process by allowing host language designers to focus on the core fundamental aspects of the language (*e.g.* its type and module system) while allowing many other design and development issues to be handled by composable language extensions.

When we talk about extensible languages or compilers, we may implicitly evaluate the claim of extensibility from our own perspective as language developers: Is the notation for specifying language extension convenient for me? What limitations are imposed on specifications? But this perspective clouds what arguably matters most: the perspective of the extension user. This is the programmer who is not a compiler engineer, and who cannot write the "glue" code necessary to compose extensions. They do not expect the composition process to fail, neither when extensions are composed, nor when the compiler for the composed language fails to process their program. Thus, we have stressed the importance of composability for the programmer over convenience or simplicity for the language extension or host language developer.

But this does not mean there are not significant benefits to language extension developers as well. Consider the developers of Cilk. Their interest was in language abstractions for parallel programming with sophisticated run-time support. But to realize their ideas they needed to build their own Cilk translator that parses and analyses all of C in addition to the Cilk features. This is no small endeavor and the journey of the Cilk abstractions from a research language to their inclusion in the Intel compilers is a long one. Would it not be more appealing to develop one's ideas for new language abstractions in a framework where one can focus on those abstractions and not need to rebuild so much language infrastructure? It is our conjecture that if, as a community, we can provide language frameworks that have strong guarantees of composition then they are more likely to be adopted by programmers, provide more opportunities for programmers to use the domain-specific abstractions needed for their particular task at hand, and thus provide real user feedback to the abstraction designers.

10.3 Future Work

While we have found ableC to work well for a wide variety of language extensions, there are enhancements that we would like to make. First, neither Silver nor ableC has built-in support for avoiding name capture in the generated C code as found, for example, in hygienic macros [Kohlbecker et al. 1986]. Silver's module system ensures no name clashes between different Silver grammar modules by using fully qualified names that include the unique grammar name of the extension (when that grammar name is based on the users Internet domain). Extension developers can avoid name clashes in the generated C code with other extensions by prefixing their names with a C-allowed version of the grammar name and then avoid generating duplicate names from their extension. This works but is unsatisfactory and we are investigating extensions to ableC to address this, perhaps based on recent algorithms for this [Erdweg et al. 2014; Ritschel and Erdweg 2015].

If the extended ableC program reports no syntax or semantic errors, then an error caused by inadvertent name capture or any other incorrect transformation to C is a serious problem. In fact, we should add a third requirement to the reliable parsing and expression problem challenges in Section 2.2, that if there are no errors in the extended program there should be no errors in the generated program. Lorenzen and Erdweg [2016] have shown how to verify this automatically on a significant subset of Java. While impressive, this primarily shows that the language extensions translation and type checking are in a sense consistent with one another. In the more general setting of attribute grammars, we also need to establish a notion of non-interference between extensions. Specifically we want extension developers to be able to state properties about the language defined by combining their extension with the host language. For example, the extension that introduced the `nonnull` type qualifier may want to state (and prove) the property that no programmer-written pointer dereferences of null pointers will occur at runtime. This property should continue to hold when additional language extensions are used. We are currently investigating two approaches to this, one that requires a strict semantic equivalence between the extension construct and what it forwards to. This approach is described in Kaminski [2017, Chapter 6] and Kaminski and Van Wyk [2017]. This is a testing-based approach but relies on some ideas from formal verification to be able to define a notion of non-interference that can assign blame to language extensions that in some sense interfere. This approach has been successfully applied to some extensions to ableC but does impose additional restrictions on their expressiveness. A second approach that we are investigating relaxes this restriction of semantic equivalence and instead requires only that the extension preserves (or refines) the semantics (that is, attribute values) of what it forwards to. But this approach imposes additional restrictions on what extension productions can forward to, specifically, the child nodes whose nonterminals are declared in the host language must appear unchanged in the forwards-to tree. Both of these approaches make different kinds of compromises to achieve a notion of semantic composition guarantees, as opposed to the essentially syntactic/well-formedness guarantees provided by the modular determinism and well-definedness analyses, and further work is needed.

As mentioned above we are also in enhancing the supporting infrastructure of the ableC host language attribute grammar to include attributes that construct program flow graphs. Besides the obvious uses in performing control and data flow analysis, this could be used to support additional extension optimizations and flow-sensitive type qualifiers as mentioned in Section 7.3.

ACKNOWLEDGMENTS

This material is partially based upon work supported by the National Science Foundation (NSF) under Grant Nos. 1628929, 1047961, and 0905581. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

We also thank August Schwerdfeger for his continued work on Copper and support in the development of ableC.

REFERENCES

S. R. Adams. 1993. *Modular Grammars for Programming Language Prototyping*. Ph.D. Dissertation. University of Southampton, Department of Elec. and Comp. Sci., UK.

A.V. Aho, R. Sethi, and J.D. Ullman. 1986. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA.

Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. 2007. Tom: Piggybacking rewriting on Java. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA '07).*

John Tang Boyland. 2005. Remote attribute grammars. *J. ACM* 52, 4 (2005), 627–687.

Martin Bravenboer and Eelco Visser. 2004. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA '04).* ACM, 365–383.

Travis Carlson and Eric Van Wyk. 2017. Type qualifiers as composable language extensions. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Generative Programming: Concepts & Experience (GPCE '17).* ACM, New York, NY, USA.

Russel Cox, Tom Bergany, Austin Clements, Frans Kaashoek, and Eddie Kohlery. 2008. Xoc, an Extension-Oriented Compiler for Systems Programming. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*

(ASPLOS).

Lukas Diekmann and Laurence Tratt. 2014. *Eco: A Language Composition Editor*. Springer International Publishing, Cham, 82–101.

Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. 2011. Building and Using Pluggable Type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 681–690.

Torbjörn Ekman and Görel Hedin. 2007a. The JastAdd extensible Java compiler. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA '07)*. ACM, 1–18.

Torbjörn Ekman and Görel Hedin. 2007b. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming* 69 (December 2007), 14–26. Issue 1-3.

Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA '11)*. ACM, 391–406.

Sebastian Erdweg, Tijs van der Storm, and Yi Dai. 2014. Capture-Avoiding and Hygienic Program Transformations. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 8586. Springer, 489–514.

R. Farrow, T. J. Marlowe, and D. M. Yellin. 1992. Composable attribute grammars. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 223–234.

Matthew Flatt. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Inc.

Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 111–122.

Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*. ACM.

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*. Berlin, Germany, 1–12.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of Programming Language Design and Implementation (PLDI '98)*. ACM, 212–223.

H. Ganzinger. 1983. Increasing Modularity and Language-Independency in Automatically Generated Compilers. *Science of Computer Programing* 3, 3 (1983), 223–278.

Robert Grimm. 2006. Better extensibility through modular syntax. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '06)*. ACM Press, New York, NY, USA, 38–51.

G. Hedin. 2000. Reference Attribute Grammars. *Informatica* 24, 3 (2000), 301–317.

P. Hudak. 1996. Building Domain-Specific Embedded Languages. *Comput. Surveys* 28, 4es (1996).

Ted Kaminski. 2017. *Reliably Composable Language Extensions*. Ph.D. Dissertation. University of Minnesota, Minneapolis, Minnesota, USA. Available at http://hdl.handle.net/11299/188954.

Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. *Reliable and automatic composition of language extensions to C — Supplemental Material*. Technical Report 17-009. University of Minnesota, Department of Computer Science and Engineering. Available at https://www.cs.umn.edu/research/technical_reports/view/17-009.

Ted Kaminski and Eric Van Wyk. 2011. Integrating attribute grammar and functional programming language features. In *Proceedings of the 4th International Conference on Software Language Engineering (SLE '11) (LNCS)*, Vol. 6940. Springer, 263–282.

Ted Kaminski and Eric Van Wyk. 2012. Modular well-definedness analysis for attribute grammars. In *Proceedings of the International Conference on Software Language Engineering (SLE) (LNCS)*, Vol. 7745. Springer, 352–371.

Ted Kaminski and Eric Van Wyk. 2017. Ensuring Non-interference of Composable Language Extensions. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE '17)*. ACM, New York, NY, USA.

U. Kastens and W. M. Waite. 1994. Modularity and reusability in attribute grammars. *Acta Informatica* 31 (1994), 601–627.

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*. ACM.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: a Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of Source Code Analysis and Manipulation (SCAM)*.

D. E. Knuth. 1968. Semantics of Context-free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. Corrections in **5**(1971) pp. 95–96.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and Functional Programming*. ACM Press, 151–161.

Florian Lorenzen and Sebastian Erdweg. 2016. Sound Type-dependent Syntactic Language Extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, New York, NY, USA, 204–216.

Yogesh Mali and Eric Van Wyk. 2011. Building Extensible Specifications and Implementations of Promela with AbleP. In *Proceedings of the International SPIN Workshop on Model Checking of Software (LNCS)*, Vol. 6823. Springer, 108–125.

Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. 2003. A Pattern Matching Compiler for Multiple Target Languages. In *Proceedings of the 12th International Conference on Compiler Construction (LNCS)*, Vol. 2622. Springer, 61–76.

G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. 2002. Cil: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction (Lecture Notes in Computer Science)*, Vol. 2304. Springer-Verlag, 213–228.

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myer. 2003. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (LNCS)*, Vol. 2622. Springer, 138–152.

Cyrus Omar. 2017. *Reasonably Programmable Syntax.* Ph.D. Dissertation. Carnegie Mellon University, Pittsburgh, USA.

Cyrus Omar, Benjamin Chung, Darya Kurilova, Alex Potanin, and Jonathan Aldrich. 2013. Type-Directed, Whitespace-Delimited Parsing for Embedded DSLs. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages (GlobalDSL '13).* ACM, New York, NY, USA, 8–11.

Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2014. Safely Composable Type-Specific Languages. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP '14)*, Richard Jones (Ed.). Lecture Notes in Computer Science, Vol. 8586. Springer, 105–130.

Dan Quinlan. 2000. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters* 10, 02n03 (2000), 215–226.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Transactions on Graphics* 31, 4, Article 32 (July 2012), 12 pages.

Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2014. From object algebras to attribute grammars. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 377–395.

Nico Ritschel and Sebastian Erdweg. 2015. Modular Capture Avoidance for Program Transformations. In *Proceedings of the 2015 ACM SIGPLAN Conference on Software Language Engineering (SLE '15).* ACM, New York, NY, USA, 59–70.

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ACM SIGPLAN 2010 Conference on Generative Programming and Component Engineering (GPCE '10).* ACM, New York, NY, USA, 127–136.

Joao Saraiva and Doaitse Swierstra. 1999. Generic Attribute Grammars. In *2nd Workshop on Attribute Grammars and their Applications.* 185–204.

August Schwerdfeger. 2010. *Context-Aware Scanning and Determinism-Preserving Grammar Composition, in Theory and Practice.* Ph.D. Dissertation. University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA. Available at http://purl.umn.edu/95605.

August Schwerdfeger and Eric Van Wyk. 2009. Verifiable Composition of Deterministic Grammars. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '09).* ACM, 199–210.

August Schwerdfeger and Eric Van Wyk. 2010. Verifiable Parse Table Composition for Deterministic Parsing. In *Proceedings of the 2nd International Conference on Software Language Engineering (LNCS)*, Vol. 5969. Springer, 184–203.

Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. 2001. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium.* Washington, D.C., 201–218.

Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional software. *SIGPLAN Notices* 41, 10 (2006), 451–464.

Anthony M. Sloane. 2011. Lightweight language processing in Kiama. In *Proceedings of the 3rd summer school on Generative and Transformational Techniques in Software Engineering III (GTTSE '09) (LNCS)*, Vol. 6491. Springer, 408–425.

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1–2 (January 2010), 39–54.

E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. 2002. Forwarding in Attribute Grammars for Modular Language Design. In *Proceedings of the 11th Conference on Compiler Construction (CC) (LNCS)*, Vol. 2304. Springer-Verlag, 128–142.

Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. 2007. Attribute Grammar-based Language Extensions for Java. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP) (LNCS)*, Vol. 4609. Springer, 575–599.

Eric Van Wyk and August Schwerdfeger. 2007. Context-Aware Scanning for Parsing Extensible Languages. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE '07).* ACM, 63–72.

Eelco Visser. 2001. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA'01) (Lecture Notes in Computer Science)*, A. Middeldorp (Ed.), Vol. 2051. Springer-Verlag, 357–361.

Markus Voelter. 2011. Language and IDE development, modularization and composition with MPS. In *Proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE '11) (LNCS)*, Vol. 7680. Springer, 383–430.

Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. 2012. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH Wavefront '12).* ACM, New York, NY, USA, 121–140.

H. Vogt, S. D. Swierstra, and M. F. Kuiper. 1989. Higher-order Attribute Grammars. In *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI).* ACM, 131–145.

Phil Wadler. 1998. The expression problem. (December 1998). Discussion on the Java-Genericity mailing list, available at http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

Yanlin Wang and Bruno C. d. S. Oliveira. 2016. The Expression Problem, Trivially!. In *Proceedings of the 15th International Conference on Modularity (MODULARITY '16).* ACM, New York, NY, USA, 37–41.

D. Weise and R. Crew. 1993. Programmable Syntax Macros. *ACM SIGPLAN Notices* 28, 6 (1993).

Matthias Zenger and Martin Odersky. 2005. Independently extensible solutions to the expression problem. In *Proceedings of the Workshop on Foundations of Object-Oriented Languages, FOOL '12.*