Type Qualifiers as Composable Language Extensions for Code Analysis and Generation

Travis Carlson, Eric Van Wyk

Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

Abstract

The work presented here reformulates type qualifiers as composable language extensions that can be automatically and reliably composed by the end-user programmer. Type expressions can be annotated with type qualifiers to specify new subtyping relations that are expressive enough to detect many kinds of programming errors. Type qualifiers, as illustrated in our ableC extensible language framework for C, can also introduce rich forms of concrete syntax, can generate dynamic checks on data when static checks are infeasible or not appropriate, and can inject generated code that affects program behavior, for example to log or display program execution or for the run-time conversion of data.

The ableC framework and extensions to it are implemented using context-free grammars and attribute grammars. This provides an expressive mechanism for type qualifier implementations to check for additional errors, *e.g.* dereferences to pointers not qualified by a "nonnull" qualifier, and generate custom and informative error messages. This approach distinguishes programmers that *use* language extensions from language engineers that *develop* extensions. The framework provides modular analyses that extension developers use to ensure that their extension will compose with other extensions that all pass these analyses. Thus, when a programmer selects a set of extensions to use they will automatically and reliably compose to form a working translator for the extended language.

Keywords: type qualifiers, type systems, pluggable types, extensible languages

1. Introduction and Motivation

C and C++ programmers are familiar with type qualifiers such as `const` or `volatile` that allow the programmer to prohibit certain operations on values of qualified types. These qualifiers also introduce subtype relationships. This is typically done to improve the safety and quality of the program. As an example, when the `const` qualifier is used in a variable declaration only initializing assignments are allowed to that variable; all other assignments are disallowed. The subtype relationship restrictions are visible in a function with a single argument declaration `int *x`. Such a function cannot be passed a value of type `const int *` as this would allow changes to the `const int` via the pointer. Here the type `int *` is seen as a subtype of `const int *`.

In their seminal paper "A Theory of Type Qualifiers", Foster et al. [1] provide a formalization of this subtyping relationship as a lattice structure. They also show how user-defined type qualifiers can be added to a language to enable additional restrictions on the use of certain operators based on the qualifiers on types of the arguments. One example is a `nonnull` qualifier for pointer types to indicate that the value of the pointer will not be null. A pointer `p` declared as

```
int * nonnull p = &v;
```

can be passed to a function as an argument with type `int *`, but the reverse of passing an `int *` value as an argument with type `int * nonnull` is disallowed. As an additional operator restriction, pointer dereference is not allowed on pointers whose type is not qualified as `nonnull`. Attempts to do so raise a static error. On the other hand, the qualifier `tainted` induces a different subtype relationship. With it, the type `char *` is a subtype of `tainted char *` and

Email addresses: travis.carlson@cs.umn.edu (Travis Carlson), evw@umn.edu (Eric Van Wyk)

```
typedef  datatype Expr  Expr;

datatype Expr {
  And (Expr * nonnull, Expr * nonnull);
  Or  (Expr * nonnull, Expr * nonnull);
  Literal (bool);
};
```

```
bool value (Expr * nonnull e) {
  match (e) {
    And (e1, e2) -> { return value(e1) && value(e2); }
    Or  (e1, e2) -> { return value(e1) || value(e2); }
    Literal (v)  -> { return v; }
  }
}
```

Figure 1: Algebraic datatype and `nonnull` extensions.

this disallows a value with this qualified type to be used where an unqualified one is expected. We describe these subtype relationships, which form a lattice of qualified types, in more detail in Section 3. In the work of Foster et al. the qualifiers are called "user-defined" and it is a programmer that specifies a new qualifier, the form of subtype relationship that it induces, and the operations that it disallows.

In contrast to these user-defined qualifiers, the approach taken in this paper distinguishes the programmer that uses a qualifier defined in a language extension from the language engineer that implements the language extension defining the new type qualifier. We reformulate the type qualifiers described above and several others as independently-developed language extensions to the ableC extensible language framework for C [2]. We make a clear distinction between the independent parties that may develop various language extensions and the extension users (programmers) that may select the extensions that address their task at hand. These extensions may add new domain-specific syntax (notations) or semantic analyses to their host language, in this case C. While extension developers must understand the underlying language implementation mechanisms used in ableC, the guarantees of composability provided by ableC and its supporting tools ensure that the extension users do not need such knowledge.

The specifications of language extensions for ableC are written as context free grammars (for concrete syntax) and attribute grammars (for semantic analysis and code generation), as described in Section 4. The tools that process these specifications provide *modular* analyses of extension specifications that an extension developer uses to ensure that their extension will automatically and reliably compose with other independently-developed extensions that also pass these analyses. This means that the extension users (programmers) do not need to be language engineers and do not need to know how the underlying tools and techniques work. Thus extension designers are free to write expressive language extensions that introduce new syntax and semantic analysis with the knowledge that their extension will be easily used by programmers as long as it passes the modular analyses.

This does require more sophistication of the extension developer; but this approach does provide them with the tools for writing more syntactically and semantically expressive type qualifiers than possible in other approaches. Extension developers are required to have working knowledge of context free grammars and attribute grammars (AGs); in return it is possible to specify extensions that are syntactically more complex than the introduction of a single new keyword and that can perform more complex analyses.

Syntactically, type qualifiers can define an expressive sub-language used as part of the qualifier. For example, `units(kg*m^2)` specifies a SI measurement in newtons. Figure 1 contains an example program that uses two language extensions: one introducing algebraic datatypes similar to those in ML or Haskell, and a second extension specifying the `nonnull` qualifier as described above. The algebraic datatype extension is used to declare a `datatype` for simple Boolean expressions and to then write a function to compute their value. The `value` function takes pointers to expressions qualified as `nonnull`. The programmer writing this code would have imported both of these independently-developed language extensions into ableC in order to use both extensions in the same program.

Type qualifiers specified as ableC extensions can perform the sort of analysis exemplified by a `nonnull` qualifier. Additionally, in the process of translating the extended C program down to a plain C program, they can generate code that is injected into the final C program. These can be dynamic checks of data when static checks are not possible or appropriate, or computations to perform data conversion or generate print statements to monitor a changing variable as is done with the `watch` type qualifier shown in Figure 2.

The work presented here presents type qualifiers as language extensions in an extensible setting with guarantees of composability for independently-developed type qualifier extensions. Previous work is extended with a more expressive mechanism for specifying new static checks based on type qualifiers and handling of parameterized type qualifiers (Section 4). We describe a refactoring and extension of the original ad-hoc handling of the C type qualifiers

Use of `watch` type qualifier in a program named `demo.xc`.

```
watch int product = 1;
for (int i=1; i < 4; ++i) {
  product = product * i;
}
```

Output of the for-loop:

```
demo.xc:1: (product) = 1
demo.xc:3: (product) = 1
demo.xc:3: (product) = 2
demo.xc:3: (product) = 6
```

The translation to C of assignment to `product`:

```
product = ({
  int tmp = product * i;
  printf("demo.xc:3: (product) = %s\n",
         showInt(tmp).text);
  tmp; });
```

Figure 2: Code insertion by the `watch` type qualifier, and its output.

in ableC to support

- Mechanisms for distinguishing the behavior carried out in type qualifier checking on code written by a programmer versus code generated by another language extension or included by the C preprocessor (Section 5.1).
- A means for automatically combining type qualifier annotations made to library headers (function prototypes) from multiple extensions to alleviate a manual process in Cqual [1], Foster et al.'s implementation of user-defined type qualifiers in C (Section 5.2).
- Mechanisms for type qualifiers to be specific to a type (and generate errors when applied to other types) and for type qualifiers to be independent of the type they qualify (Section 6).
- Qualifiers that add dynamic checks of data when static checks are not feasible or appropriate (Section 7).
- Qualifiers that insert additional code, for example to perform run-time code insertion (Section 8).

These techniques naturally apply to C, as type qualifiers are already part of the language, but they could be used in other languages as well. Similar ideas could be developed in ableJ [3], our previous implementation of Java in Silver, but the contributions are not limited to attribute grammars in Silver and could be applied in other settings for building languages.

In describing this work, we demonstrate several type qualifiers such as `nonnull` and qualifiers to indicate functions as being pure and associative, qualifiers with richer syntax such as one for dimension analysis to check that physical measurement units (*e.g.* meters, seconds, etc.) are used correctly, extensions that insert code such as dynamic array bound checks and check for consistent use of tags in tagged-unions, the `watch` qualifier (extended to display data flowing into and out of functions), and data conversion (*e.g.* scaling values in millimeters to meters in the dimensional analysis extension). Section 9 discusses related work before Section 10 discusses some future work and concludes.

This paper is an extension of our previous paper [4] on this topic that appeared in the 2017 ACM SIGPLAN Conference on Generative Programming: Concepts & Experience (GPCE 2017). Here we provide a more complete description of the infrastructure provided by ableC for specifying expressive type qualifiers than was possible in the conference version of the paper. The example type qualifiers have been enhanced; specifically the `watch` qualifier can be used to display values of arguments on function calls and the return values on function returns, in addition to the tracking of changes made by assignment statements as illustrated in Figure 2. The `check` qualifier for generating array bounds checks has been refactored into 1) a base extension that provides the `check` qualifier, and 2) extensions that build on this base to inject dynamic checks on additional kinds of data. We illustrate this with an extension for C tagged unions; these are `struct` types with an enumerated type tag that indicate which element of a contained `union` is to be used. The extension injects code to dynamically check that all accesses to the `union` are consistent with the value in the tag, thus detecting common errors when these two fields in the `struct` are used in an inconsistent manner.

2. Background: Type Qualifiers

Our contributions with regard to type qualifiers build on previous work that formulates type qualifiers as user-defined constructs that introduce a subtype relationship between qualified and unqualified versions of a type. We thus review this previous work and how it is incorporated into ableC. Following this, in Section 3, we describe the ableC extensible language framework [2].

2.1. Signed type qualifiers in C

The type qualifiers described by Foster et al [1] are *signed*, to induce subtype relationships in a particular direction on qualified types. The sign of their `nonnull` qualifier is *negative*. This indicates that for some type τ, the qualified type `nonnull` τ is a subtype of τ. Thus one can use a `nonnull` pointer in any place where an unqualified pointer can be used, but not the reverse. The `tainted` qualifier is an example of a *positive* qualifier, this is meant to indicate data that may come from a possibly-malicious or un-trusted source and so should not be used in exploitable functions. This type of qualifier has been shown to be effective in detecting format string vulnerabilities [5]. Positive qualifiers induce the opposite subtype relation so that a type τ is a subtype of `tainted` τ. Thus tainted values cannot be passed into functions that do not explicitly accept them. The standard C qualifiers `const`, `volatile`, and `restrict` are positive. In general, for a (possibly qualified) type τ and a positive qualifier pq and negative qualifier nq the following subtyping relations hold:

$$nq\ \tau \preceq \tau \quad \text{and} \quad \tau \preceq pq\ \tau.$$

Type are associated with an (unordered) sets of qualifiers and thus the order in which they are written does not matter. The subtype relation induced by the composition of multiple qualifiers is modeled as a lattice [1, Section 2]. For each type qualifier q, we define a corresponding two-point lattice L_q. If q is positive, we define $L_q = \bot_q \sqsubseteq q$; otherwise if q is negative, we define $L_q = q \sqsubseteq \top_q$, where the absence of a qualifier is denoted by $\bot_q$ if q is positive and by $\top_q$ if q is negative. A lattice for type qualifiers $q_1, \cdots, q_n$ is then defined as $L = L_{q_1} \times \cdots \times L_{q_n}$. For sets of qualifiers Q_1 and Q_2, it is the case that $Q_1\tau \preceq Q_2\tau$ if and only if the lattice element corresponding to Q_2 dominates the lattice element corresponding to Q_1.

A special point must be made regarding pointers and their qualifiers. A pointer type may contain qualifiers on the pointer type itself as well as on the type that is pointed to. To determine if one qualified pointer is a subtype of another, the first step considers the outer-level qualifiers on the pointers; these are compared based on the subtype relation induced by their signs. Care must then be taken in comparing the inner-level qualifiers. In general, it is unsound to compare the inner-level qualifiers using subtyping rules, sometimes referred to as *deep subtyping* [6, Section 3.6]. These qualifiers are checked for equality instead. As an example, `char *x; tainted char *y = x;` should not be allowed even though `char` is a subtype of `tainted char`. The exception to this rule is when the inner type is qualified as `const` and hence cannot be updated.

When used as an l-value (as on the left-hand side of an assignment), a C variable refers to the memory location where the value of that variable is stored. When used as an r-value (as on the right hand side), it refers to the value itself. Thus a variable declared to be of type τ can be thought of as being of type *reference* τ and of being automatically dereferenced when used as an r-value. Because this is left implicit, a programmer wishing to qualify a type has no way to specify whether the qualifier should apply to this outermost reference or to the value it refers to, and so this behavior is set as a property of each qualifier. For example, declaring a variable as type `const` τ specifies that the memory location that the variable refers to cannot be written to. In contrast, declaring a variable as type `nonnull` τ specifies that the value stored at the memory location that the variable refers to is not null. A qualifier like `const` that applies to the implicit outermost reference is specified to apply at the *ref level*, while a qualifier like `nonnull` that applies underneath it is specified to apply at the *value level* [6, Section 5.2].

2.2. Additional forms of static checks

While the subtypes induced by qualifiers are checked and enforced throughout the entire program, it is possible to specify additional static checks on specific language constructs. As mentioned above, the pointer dereference operator `*` performs an additional check to verify that the type of the pointer being dereferenced is qualified as `nonnull`. If this is not the case, an error message is generated and reported. This ensures that all pointer references are done safely.

The Cqual system allows users to define these sorts of checks in what is called a *prelude* file. Programmers specify these by writing with which qualifiers must appear on the arguments to operators. For example, to require a `nonnull` qualifier on pointer dereferences the user would write the following: `$$a _op_deref($$a *$nonnull);`

2.3. Flow-sensitive type qualifiers

A useful feature of CQUAL is the flow sensitivity of qualifiers. This allows a type system to infer different type qualifiers for a variable at different program points [7] based on the control and data flow of the program. It is sometimes possible to determine at certain points that a pointer cannot possibly be null, for example, but not at others. Consider a simple example; a declaration of a pointer `p` such as `int *p = &v;` that is followed immediately by a dereference of `p` could be allowed because it can be inferred that at that dereference point the pointer `p` will always be non-null. By inferring the locations at which a type is known to be a subtype of the type it was declared to be, the number of annotations required of the user can be reduced while retaining the error-checking benefits [8].

3. Background: ableC

In this section we describe ableC [2], an extensible C compiler front-end at the C11 standard that allows programmers to import new language features into C.[1] ableC is implemented using the Silver[2] attribute grammar system [9] and the Copper [10] parser and context-aware scanner generator embedded in Silver.

Language extensions, also written in Silver, can introduce new concrete and abstract syntax, definitions for host language semantic analyses (such as type checking and error-generation) over the new syntax, as well as new semantic analyses over the host language and extension syntax (such as a new pointer analysis). Below we describe language extension from the perspective of the programmer that uses language extensions and then from the perspective of the developer that implements language extensions.

3.1. From the programmer's perspective

Consider again the example in Figure 1 in which two language extensions, the algebraic data type extension and non-null extension, are used in the same extended C program. To make use of these extensions in a program such as this, the programmer must first direct Silver to compose the extensions with the host language specification to create a translator for this custom language. To do this, the programmer needs to do little more than give a name for the generated translator and list the desired extensions. In this case, the composition specification on the left of Figure 3 contains the required information.

On the right of this figure is an illustration of the process of using ableC. The composition specification directs Silver to type-check the extension and host language specifications for static errors, translate the specifications to Java, and generate additional Java code to combine the separately-compiled specifications. From this specification, a translator is generated, here named `MyAbleC.jar`. A sample program containing the contents of Figure 1 may be named `example.xc`. The program is first passed through the C pre-processor. Next, the generated translator scans and parses the file to generate the abstract syntax tree (AST) for the program. Attribute evaluation is done on the AST; this is the process by which static analyses such as type checking are carried out. The ableC attribute grammar specification implements the C type checking rules and language extensions should also define type checking rules (as attribute equations) for the constructs they introduce. This ensures that type errors are reported on the code that the programmer wrote, not on the generated C code. This is quite important, otherwise language extensions would essentially be only macros where static analysis takes place on the expansion of the macro; we avoid this in ableC.

Additionally in this process, operator overloading is resolved and the language extension constructs are translated into plain C code. In the case of the algebraic data types, the data type declaration is translated into `struct` and `union` constructs and the `match` statement is translated into a nested *if-then-else* statement that uses code generated from the patterns to inspect the data and determine which statement to execute. Finally, the plain C code is compiled using a standard C compiler such as GCC. In our experience developing `ableC` extensions, we have frequently found GCC extensions useful, specifically the statement-expression construct which allows an expression to contain statements to be executed prior its evaluation; thus we require a compiler that supports such features, such as GCC or Clang.

Because of the modular analyses described below, the programmer has the assurance that the composition of these (potentially) independently-developed language extensions will be well-formed. The only caveat is that the programmer may need to resolve a lexical ambiguity that is very similar to what arises in Java when two packages

[1] Available at `http://melt.cs.umn.edu/ableC`, archived at `https://doi.org/10.13020/D6VQ25`

[2] Available at `http://melt.cs.umn.edu/silver`, archived at `https://doi.org/10.13020/D6QX07`

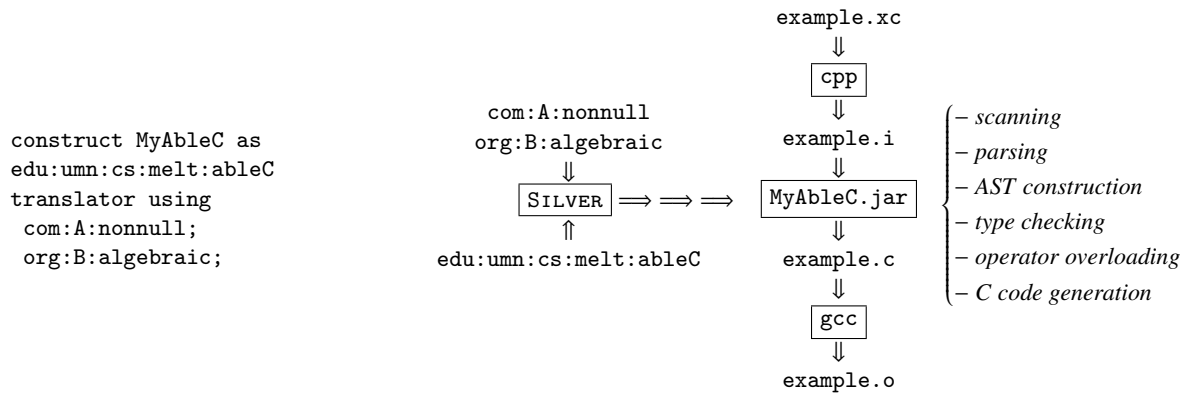

Figure 3: The programmer-written specification for composing the language extensions used in Figure 1 (left) and an illustration of the ableC build and use processes (right).

define a class with the same name: a "full name" for the ambiguous classes must be used. These ambiguities are easily resolved by the programmer using a mechanism called *transparent prefixes* [11]. This amounts to specifying a prefix for each marking terminal to be typed before that keyword in the program disambiguating it. For example, suppose we wish to modify the composition specification on the left of Figure 3 to use an additional extension for regular expression matching, called `edu:C:regex`, that introduces a `match` marking terminal conflicting with that of the `org:B:algebraic` extension (which provides both the algebraic datatype and the associated pattern-matching construct). A clause such as `prefix with "rx:"` would need to be added specifying the prefix to be written before each use of this new terminal; thus the programmer writes `rx:match` when they mean the regular expression `match` keyword. A similar prefix specification may be written for the algebraic data types extension. Alternatively, as shown below, the marking terminal for one extension can be the default over another if the `prefer over` clause is used. This modified specification may be written as follows:

```
construct MyAbleC as edu:umn:cs:melt:ableC translator using
  com:A:nonnull;
  org:B:algebraic  prefer over edu:C:regex;
  edu:C:regex      prefix with "rx:";
```

3.2. From the language extension developer's perspective

Developers of language extensions do need some understanding of language design and implementation, specifically they must understand context-free grammars and attributes grammars and how they are used in the Silver attribute grammar specification language. In Silver, developers specify the concrete and abstract syntax of their language extensions. They define the attributes and attribute equations that implement the static semantics, such as type checking, and also define the translation of the extension down to plain C code. The concrete syntax specifications are extracted and processed by Copper.

Because these specifications are written in terms of context free grammars (both for concrete and abstract syntax) and sets of attribute equations associated with grammar productions, the composition of the host language and several independently-developed extensions is a straightforward process. Although straightforward, this process is only guaranteed to succeed if the language extensions individually meet the composability criteria enforced by the modular determinism analysis [11] (MDA) in Copper and the modular well-definedness analysis [12] (MWDA) in Silver. While these impose some restrictions on language extensions, we have found that they still allow quite expressive and useful extensions to be specified [2].

Language extension developers are responsible for running the modular determinism analysis on the concrete syntax specifications of their extension; this analysis is done with respect to the host language being extended. If the analysis detects composability issues and fails, the extension developer will need to modify their extension to fix the problems. This modular analysis guarantees that any collection of language extensions that pass the analysis (in isolation from one another) can be automatically composed by Copper to create a context-aware scanner specification

with no lexical ambiguities and a parser specification (a context free grammar) that is in the class of LALR(1) grammars. This means that there are no ambiguities and a deterministic parser can be constructed for it [13]. One of the restrictions imposed by the MDA on concrete syntax is that new productions that extend a host language construct (that is, they have a host language nonterminal on their left-hand side) begin with what is called a *marking* terminal. Another restriction is that extension productions cannot extend the *follow* sets [13] of host language nonterminals; that is, they cannot specify that new non-marking terminals now follow host language nonterminals in valid programs.

In the context of type qualifiers, the concrete syntax introduced is traditionally very simple, consisting of a single new identifier. In this case, the only possibility for conflict is in overlapping identifiers between extensions. But, qualifiers are not required to be identifiers. A qualifier defining an expressive sub-language, such as that of the dimensional analysis qualifier, must take care to follow the restrictions imposed by the MDA.

The single caveat to these guarantees is that there may be marking terminals that are valid in the same parsing context (for example, the context of type qualifiers) that have overlapping regular expressions. As described above in Section 3.1, these ambiguities are easily resolved by the programmer using transparent prefixes.

It is possible for an extension to depend on and extend another extension. Recall the example of a regular expression extension introducing a `match` terminal requiring disambiguation with an overlapping terminal in the algebraic data types extension. If the two extensions are not developed independently, the regular expression extension may extend and reuse the `match` terminal in the algebraic data types extension, effectively viewing its host language as being the composition of the `ableC` host language with the algebraic data types extension.

The modular well-definedness analysis (MWDA) provides a similar style of modular guarantee for the attribute grammar specifications that define the static semantics of language extensions. Specifically, it ensures that the composition of any collection of extensions that pass the MWDA independently will form a *well-defined* [14] attribute grammar. This ensures that attribute evaluation will not terminate abnormally because of missing (or duplicate) attribute-defining equations.

The MDA and MWDA can be used in any language developed using Silver. The ableC specification provides various extension points and mechanisms that extension developers can use to implement expressive language extensions as discussed below. Essentially, the nonterminals, productions, and attributes that a host language defines can be considered to be the API that is presented to its extensions. In ableC this includes, for example, an attribute providing location information that is used in Figure 2, an inherited attribute representing the environment that allows types of identifiers to be looked up and that can be contributed to by extensions, and a synthesized attribute representing a list of errors that, again, extensions can contribute to.

Note that in the remainder of this paper the two extensions used in Figure 3 are referred to using their actual names: `edu:umn:cs:melt:exts:ableC:nonnull` and `edu:umn:cs:melt:exts:ableC:algebraicDataTypes`. Above and in Figure 3 we have used these alternative names to highlight the fact that they can in fact be developed by independent parties.

Support for operator overloading, error insertion, and code injection. Besides these facilities, ableC also provides a sequence of extension points along the process of transforming the original extended program down to plain C code. The extension points primarily used in this paper act on host-language operators, allowing language extensions to detect static errors on operators, add type qualifiers to the resulting type of an operator, and inject code into the final C translation on an operator use.

An extension that introduces a new type can overload host-language operators on that type. Error checking and code insertion making use of the extension points can be specified to be performed either prior to or after operator overloading is resolved. To support this, ableC uses a four-stage pipeline (represented by different groups of productions) to translate a programmer-written operator to its host-language AST representation:

1. the concrete productions used to construct the parser, *e.g.* `add_c`,
2. the *overloadable* abstract productions, *e.g.* `addOverloadable`,
3. the *injectable* abstract productions, *e.g.* `addInjectable`, where code is inserted to, for example, scale values in the case of the dimensional analysis extension or to insert a print statement in the case of the `watch` extension,
4. the final abstract productions, *e.g.* `add`, which is the final C host language production on which no more transformations take place.

```
int negate (int n) <pure>
{ return 0 - n; }

vector<int> negate_all (vector<int> xs)
{ return map (negate, xs); }
```

```
int mul (int m, int n) <pure associative>
{ return m + n; }

int product (vector<int> xs)
{ return fold(mul, 1, xs); }
```

Figure 4: Use of `pure` and `associative` type qualifiers with `map` and `fold` parallel programming extension. This example also uses an extension that provides templated vectors in the style of C++.

We will see various example uses of these extension points throughout the paper, where additional background and details of this process will be provided as needed. In Section 6.2 in Figures 14 and 15, we will see how a dimensional analysis type qualifier can detect the incompatible addition of different units (*e.g.* a measurement in meters added to a measurement in seconds) over a numeric interval type introduced by a separate extension. Also in this example we see how the appropriate unit qualifiers are added to the resulting type of the operator. In Section 4.2 in Figures 8 and 9, we will see the `dereferenceInjectable` production for dereferences on pointer types and how errors are detected if the pointer type is not qualified as `nonnull`. We later see in Section 7.1 in Figures 16 and 17 a situation where static checking for `nonnull` is not feasible and the code for a run-time check is injected into the plain C code.

4. Type Qualifiers as Extensions in ableC

In this section we describe how type qualifiers (traditional qualifiers, those as found in Cqual, and more expressive qualifiers than allowed in Cqual) can be implemented as ableC language extensions. In the following sections then we describe the new capabilities that are possible for type qualifiers implemented as ableC language extensions.

Figure 1 has already provided a motivating example of language extensions providing new type qualifiers and another extension that adds new syntactic constructs (in this case algebraic data types and pattern matching statements) to the host language. Another example is the qualifiers `pure` and `associative` as applied to function types in Figure 4. Note that qualifiers on function types are written between angle brackets (< and >) to syntactically distinguish them from the (now seldom used) C syntax of writing declarations of function parameters after the parenthesis and before the function body. The `pure` and `associative` qualifiers are part of a language extension that also introduces `map` and `fold` constructs that generate parallel implementations of these common functional programming concepts. Because the parallel implementations of these constructs make no guarantees about the order in which the applied functions are applied, they must be qualified as pure functions to indicate that they have no side-effects or only ones whose order of execution does not invalidate the execution of the program. Similarly, to parallelize a fold operation, the binary function used to fold up the vector elements must be associative.

Our original proto-type specification of ableC supported the standard C qualifiers in a rather ad-hoc manner. In recent work [2], we have modified the ableC host language specification to implement qualifiers according to the model presented here and to add extension points to allow code generation features. This involved refactoring the type-checking code and adding attributes on the `Qualifier` nonterminal, and several collection attributes presented in the following sections. Now future type qualifiers can be implemented as pure extensions with no further modifications needed to ableC itself, and non-qualifier language extensions can also make use of these code-generation extension points.

4.1. Concrete syntax for qualifiers

We begin by describing how extension developers specify the concrete syntax for type qualifiers as language extensions. For many qualifiers, the concrete syntax is quite simple. This can be seen in Figure 5 which shows the specification of the concrete syntax for the `nonnull` [15] qualifier. Extension specifications begin by declaring the grammar name (`grammar`) followed by the grammars that they extend (`import`). In this case the `nonnull` extension only extends the ableC host language. This specification defines a new `nonnull` keyword as a terminal symbol whose associated regular expression matches the string "`nonnull`." It is a `marking` terminal because it marks the beginning of the extension's syntax. This terminal is also a member of the the `Ckeyword` lexer class to indicate that it has lexical precedence over the C identifier terminal whose regular expression overlaps with the regular expression for `nonnull`. To specify that this new keyword is a qualifier, a concrete production is declared with the host language `Qualifier_c`

```
grammar edu:umn:cs:melt:exts:ableC:nonnull;
import edu:umn:cs:melt:ableC;

marking terminal Nonnull_t 'nonnull' lexer classes {Ckeyword};

concrete production nonnull_c
q::Qualifier_c ::= 'nonnull'
{ q.ast = nonnull(); }
```

Figure 5: Concrete syntax specification of the `nonnull` type qualifier.

nonterminal on the left-hand side and the new terminal on the right-hand side. By convention we suffix names of concrete syntax elements with "`_c`" to distinguish them from their abstract syntax counterparts. A single attribute equation within the production body specifies that the abstract syntax tree for this qualifier be constructed using the `nonnull` abstract production that will be defined below in Figure 7.

Concrete, Abstract, and Aspect Productions in Silver. A production in a Silver specification may be one of three kinds: concrete, abstract, or aspect. A *concrete* production, as seen in Figure 5, is indicated by the `concrete` keyword and contributes to the definition of the context-free grammar used to generate the parser; concrete productions often contain only a single attribute equation that constructs the AST. An *abstract* production, as seen in Figure 7, is indicated by the `abstract` keyword and has no effect on parser generation; by convention, semantic analyses are primarily specified by the attribute equations in abstract productions. An *aspect* production, as seen in Figure 11, is indicated by the `aspect` keyword and allows a language extension to provide additional attribute-defining equations to the existing (concrete or abstract) production with the same name. Aspect productions are often used by extensions that introduce new attributes on host-language nonterminals in order to perform a new semantic analysis. We will see in Section 4.2 how aspect productions are used together with collection attributes to contribute additional information to an existing attribute.

While the syntax of traditional qualifiers such as `nonnull` is quite simple, our dimensional analysis qualifier is parameterized by an arithmetic expression over the SI units in Table 1. For example, a floating point variable whose value is a measure of acceleration may be declared as

```
units(m/s^2) float acceleration;
```

in which the unit of measurement is expressed using the division and power symbols over the base units. Unit symbols may be preceded by standard metric prefixes such as *k* for kilo or *m* for milli as in `units(kg*m^2/s^2)` to express the derived unit of joules for energy.

The Silver specification of the concrete syntax for the dimensional analysis qualifier is shown in Figure 6. As with the `nonnull` specification, a marking terminal is declared to introduce a new keyword and a production is declared to specify a new kind of qualifier. Here, the right-hand side of the new qualifier production follows the `units` marking terminal with a `UnitExpr_c` nonterminal enclosed in parentheses.

Table 1: Unit symbols and meanings.

Symbol	Name	Dimension
`m`	meter	length
`g`	gram	mass
`s`	second	time
`A`	ampere	electric current
`K`	kelvin	thermodynamic temperature
`mol`	mole	amount of substance
`cd`	candela	luminous intensity

This sub-language for unit expressions consists of productions defining arithmetic operations over the base SI units (`BaseUnit_c`). These SI units may optionally be prefixed by a scaling factor (`UnitPrefix_c`). From these expressions an AST of type `UnitsExpr` is extracted as the `ast` attribute, as seen in the equation on the `units_c` production. The `ast` attribute is parameterized by its type which allows the type of the attribute to vary based on the nonterminal that it decorates.

Terminals for base units from Table 1 follow, though some are elided. Note that line comments in Silver begin with two dashes (`--`). Terminals for operators are declared next with appropriate precedence and associativity and are used in the concrete productions for `UnitExpr_c` below. ASTs of derived units are constructed from abstract productions `mulUnit`, `expUnit`, and `scaledUnit` defined in the abstract syntax specification.

An important feature of Copper, the scanner and parser generator used by Silver, is its use of context-aware scanning. To see this, we note that there is no lexical ambiguity between the new base unit and prefix terminals and the host language identifier terminals even though their regular expressions overlap. For example, there is no issue in differentiating the token types of the two occurrences of `m` in the declaration `units(m) float m;`, the first being `Meter_t` and the second `Identifier_t`. Nor are there lexical ambiguities between the new operator terminals here and the arithmetic operator terminals for C in the host language. This is because Copper generates *context-aware* scanners [10]. When the scanner is called for the next token, it uses the current LR parsing state to identify those terminal symbols that are valid (those with a *shift*, *reduce*, or *accept* action in the current state). The scanner then only scans for these terminal symbols. Thus in the parsing context of `UnitExpr_c` the host language terminals mentioned above are not valid and thus there is no ambiguity. Context-aware scanning plays an important role in making both lexical ambiguities and LR table conflicts less common and thus makes the restrictions imposed by modular determinism analysis practical.

There are limitations to the disambiguating capabilities of context-aware scanning, however. This can be seen in the use of a single terminal, `Meter_t ('m')`, that is used to represent both the base unit meter and the prefix milli. Both uses are valid in the same context and so using a second terminal `Milli_t` would introduce a lexical ambiguity even in the presence of context-aware scanning. This type of reuse is common in LALR(1) parser specifications and is a bit of an annoyance. It is important to note that this annoyance affects the extension developer (who is expected to understand these issues) and not the extension user (who is not). The modular analyses described earlier ensure that the composition of different language extension specifications is automatic and reliable so that the extension users need not know of these concerns.

The specifications of this section have shown that the concrete syntax of type qualifiers can be very simple or can include a rich sub-language as in the case of the dimensional analysis extension.

4.2. Abstract syntax and attributes for qualifiers

The scanner and parser for the composed language will construct the abstract syntax tree for the extended language program. It is on this AST that attribute evaluation takes place. This computes static semantic information such as the types of expressions or the list of errors found on a statement. Here we describe how type qualifiers fit into this process.

Attributes for qualifiers. As an example, the abstract syntax specification for the `nonnull` qualifier is shown in Figure 7. This production was used in the concrete syntax specification of Figure 5 to construct an (abstract) `Qualifier` in the AST. Qualifiers such as this are associated with a type. Types in ableC are represented by a `Type` nonterminal that has an attribute that is a list of qualifiers of this sort. The `Qualifier` nonterminal is decorated with attributes to define the static semantics of a qualifier. Equations associated with a production (and written in between the curly braces) define the values that these attributes will take. For the `nonnull` qualifier there are equations defining values such as `isPositive` on the qualifier `q`. Similar productions for the `pure` and `watch` qualifiers are defined in their language extensions.

Recall from Section 2.1 that qualifiers are associated with a sign and are specified to apply at either the reference level or at the value level. In ableC we specify the sign of a qualifier by using two Boolean attributes on the `Qualifier` nonterminal, `isPositive` and `isNegative`. The sign of `nonnull` is negative. A third Boolean attribute, `applyAtValLevel`, specifies whether the qualifier should be applied at the value level, as `nonnull` does, or at the reference level.

```
grammar edu:umn:cs:melt:exts:ableC:dimensionalAnalysis;
import edu:umn:cs:melt:ableC;

marking terminal Units_t 'units' lexer classes {Ckeyword};
concrete production units_c
q::Qualifier_c ::= 'units' '(' u::UnitExpr_c ')'
{ q.ast = units(u.ast); }

nonterminal UnitExpr_c with ast<UnitsExpr>;
nonterminal BaseUnit_c with ast<BaseUnit>;
nonterminal UnitPrefix_c with ast<UnitPrefix>;

terminal Meter_t   'm';
terminal Gram_t    'g';
terminal Second_t  's';
terminal Ampere_t  'A' ;
-- other base unit terminals are elided

terminal Mul_t '*' precedence = 1; associativity = left;
terminal Div_t '/' precedence = 1; associativity = left;
terminal Pow_t '^' precedence = 2; associativity = right;

concrete productions u::UnitExpr_c
 ::= l::UnitExpr_c '*' r::UnitExpr_c { u.ast = mulUnit(l.ast, r.ast); }
   | l::UnitExpr_c '/' r::UnitExpr_c { u.ast = mulUnit(l.ast, expUnit(r.ast, -1)); }
   | l::UnitExpr_c '^' i::IntLiteral { u.ast = expUnit(l.ast, toInt(i.lexeme)); }
   | '(' e::UnitExpr_c ')'           { u.ast = e.ast; }
   | p::UnitPrefix_c b::BaseUnit_c   { u.ast = scaledUnit(b.ast, p.ast); }

concrete productions b::BaseUnit_c
 ::= 'm' { b.ast = meterUnit(); }
   | 'g' { b.ast = gramUnit(); }
   -- other unit productions are elided

terminal Kilo_t  'k';
terminal Centi_t 'c';
-- other prefix terminals are elided

concrete productions p::UnitPrefix_c
 ::= 'k' { p.ast = sciExponent(3); }
   | 'c' { p.ast = sciExponent(-2); }
   | 'm' { p.ast = sciExponent(-3); }
      -- Note the reuse of the Meter_t terminal symbol
   |     { p.ast = sciExponent(0); }
      -- empty, the prefix is optional
```

Figure 6: The SILVER specification of the dimensional analysis qualifier `unit()` and unit expressions.

```
grammar
  edu:umn:cs:melt:exts:ableC:nonnull;

abstract production nonnull
q::Qualifier ::=     -- empty
{ q.isPositive = false;
  q.isNegative = true;
  q.applyAtValLevel = true;
  q.qualCompat =
    \ qualToCompare::Qualifier ->
        case qualToCompare of
        | nonnull() -> true

        | _         -> false ;
}
```

```
grammar
  edu:umn:cs:melt:exts:ableC:dimensionalAnalysis;

abstract production units
q::Qualifier ::= u::UnitsExpr
{ q.isPositive = false;
  q.isNegative = true;
  q.applyAtValLevel = true;
  q.qualCompat =
    \ qualToCompare::Qualifier ->
        case qualToCompare of
        | units(u2) -> unitsCompatible(u.canonical,
                                       u2.canonical)
        | _         -> false ;
}
```

Figure 7: Abstract syntax specification of `nonnull` type qualifier (left) checking qualifier compatibility entirely by tree structure, and fragment of abstract syntax specification of `units` type qualifier (right) checking qualifier compatibility through canonical representations.

Qualifiers that are positive or negative will set only one of `isPositive` and `isNegative` to `true`. Some qualifiers do not introduce a subtype relationship. Setting both attributes to `true` on a qualifier q indicates both $q\ \tau \preceq \tau$ and $\tau \preceq q\ \tau$, and thus $q\ \tau = \tau$. The `watch` qualifier is configured in this way so that no subtyping restrictions are introduced. Alternatively, setting both attributes to `false` effectively creates a new type by introducing the subtyping restrictions $q\ \tau \not\preceq \tau$ and $\tau \not\preceq q\ \tau$, and thus $\tau \neq q\ \tau$. Since ableC already has mechanisms for defining new types as language extensions we have not yet found a compelling use for this capability.

Type expressions and type checking. To determine whether a qualified type $Q_1\ \tau$ is a subtype of $Q_2\ \tau$, ableC checks if all qualifiers in Q_1 but not in Q_2 are negative, and if all qualifiers in Q_2 but not in Q_1 are positive. This requires some method to determine equality of qualifiers. For most qualifiers, such as `nonnull`, this equality relation can be implemented using pattern matching to check if two qualifiers were constructed by the same abstract production. But, this might not be sufficient for more complex qualifiers in which two qualifiers may be compatible even if they are represented by different trees. Consider the dimensional analysis qualifier. The qualifier `units(m/s^2)` represents the same units of acceleration as `units(s^-1 * s^-1 * m)`. Further, we may even wish to treat distinct units as compatible if it is possible to convert between the two, such as `units(m)` and `units(km)`.

The final attribute defining the semantics of a qualifier in our implementation, `qualCompat`, allows such notions of compatibility to be expressed per qualifier. This attribute is a function that takes another `Qualifier` to compare itself to and returns a Boolean value. In most cases this function is specified as a lambda expression (written `\` *v*`::`*t*`->` *expr*) that simply pattern matches to determine if both qualifiers were constructed using the same abstract production. An example of this is seen in the `nonnull` extension in Figure 7. The dimensional analysis extension sets `qualCompat` to a function which compares canonical representations of units, as seen in the same figure. The `canonical` attribute on `UnitsExpr` is used to convert these expressions into a canonical form that is more easily checked for equality.

Beyond subtyping, type qualifiers in the style of Foster et al. [1] can require that certain qualifiers are present on the types of arguments to specific host language operations. For example, pointer dereferences can be required to be qualified as `nonnull`. In ableC, these kinds of restrictions are implemented by augmenting features of the abstract productions in the host language using aspect productions and collection attributes.

Collection attributes in Silver. A *collection* attribute allows aspect productions to contribute additional information to be folded into an initial value of the attribute. An abstract or concrete production specifies this base value of a collection attribute using the `:=` operator, and an aspect production then contributes to the attribute using the `<-` operator. The method of contribution is specified per collection attribute; in the case that the type of the attribute is a list, contributions are typically combined using the list append operator, written `++`. The contribution order is undefined when multiple contributions exist; thus the operator should be commutative (or the operand order unimportant to the application), but this is not verified by Silver.

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

abstract production dereferenceInjectable
e::Expr ::= d::Expr
{ attr lerrors :: [Msg] with ++ ;
  lerrors := d.errors;
  forwards to if null(lerrors)
              then dereference(d)
              else errorExpr(lerrors);
}
```

Figure 8: Partial specification of ableC pointer dereference construct. This production will be enhanced in Section 7.

```
grammar edu:umn:cs:melt:exts:ableC:nonnull;

function isNonNullQualified
Boolean ::= t::Type
{ return containsQualifier(nonnull(), t); }

aspect production dereferenceInjectable
e::Expr ::= d::Expr
{ lerrors <- if isNonNullQualified(d.type)
             then [ ]
             else [ err(e.location, "dereference on pointer "
                                 ++ "without 'nonnull' qualifier") ] ;
}
```

Figure 9: Contribution of errors on dereference by `nonnull` extension. This production will be enhanced in Section 5

In the pointer dereference production of the ableC host grammar shown in Figure 8, errors are represented by a local collection attribute `lerrors`. This is a list of messages (`[Msg]`) initially populated by the errors on the dereferenced expression `d`. Though not shown here, this production will also add appropriate errors if the type of the dereferenced expression `d` is not a pointer type. Additional error messages may be added by extensions using aspect productions. If any errors are found then the dereference operation translates into an error construct that contains the errors. Otherwise it is translated to a final, post-processing version of the construct. This translation is done via *forwarding*.

Forwarding in Silver. Forwarding [16] is a key feature for language extensions such as the algebraic data types of Figure 1 and is described in more detail in our paper on ableC [2], but it plays a less substantial role in type qualifiers. Forwarding essentially indicates what AST the "forwarding" AST is to be translated to. A forwarding production is allowed to be missing attribute equations. When a request is made for an attribute without a defining equation, the value is computed on the *forwarded-to* tree. This tree is computed by the expression written after the `forwards to` keywords. This supports our goal of automatic composition of extensions, *i.e.* that no "glue code" is required to compose independent extensions. Consider, for example, the case that one extension introduces a new attribute on a host-language nonterminal and another extension introduces a new forwarding production for that same nonterminal; forwarding provides a definition for the new attribute on the new production.

To see how an extension uses aspect productions, collection attributes, and forwarding, consider how the `nonnull` extension raises errors on pointer dereferences not qualified by `nonnull`. This is shown in Figure 9. An aspect of the host-grammar production `dereferenceInjectable` contributes new errors to `lerrors` if the type of the expression being dereferenced (`d.type`) is not qualified as `nonnull`. The function `containsQualifier`, defined in the host grammar and shown in Figure 10, is used to determine whether a type is qualified as `nonnull`. This function checks if the qualifier `q` is contained in the list of qualifiers of the type `t` using a helper function `containsBy` that in turn

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

function containsQualifier
Boolean ::= q::Qualifier t::Type
{ return containsBy(qualifierCompat, q, t.qualifiers); }

function qualifierCompat
Boolean ::= a::Qualifier b::Qualifier
{ return a.qualCompat(b); }
```

Figure 10: The ableC host language implementation of `containsQualifier` in using the `qualCompat` attribute.

uses the `qualifierCompat` function to check containment in the list. This function calls the `qualCompat` attribute on one qualifier (this will be `q` from above) passing it a qualifier from the qualifiers on `t` as its argument.

The abstract productions for other host language operators in ableC follow the general pattern exemplified by `dereference`. Thus they also support the addition of new error messages by language extension grammars through aspect productions. The host language `const` qualifier raises errors on assignments using a similar implementation to check that the type of the assigned-to operand is not qualified by `const`. Similarly, the dimensional analysis extension uses an aspect production on the addition operator to raise errors on attempts to add incompatible units.

5. Type Qualifier Analysis in the Presence of Other Language Extensions and Libraries

A central theme of our work in language extension has been composability — that is, the independently-developed language extensions will compose and they all work together. Beyond the composability issues solved by the modular analyses discussed in Section 3, type qualifiers must also work as expected with other extensions and libraries used by the programmer. This raises two challenges whose solutions are discussed in this section. The first is that it will not be useful if a type qualifier generates error messages based on code *generated* by another language extension. The programmer cannot annotate that generated code with necessary type qualifiers, such as `nonnull` for pointer dereferences. We must assume the generated code is correct and thus the analysis introduced by type qualifiers needs to take its context (programmer written code versus extension generated code) into account when doing error checking. Second, source code in library header (`.h`) and source (`.c`) files may not be ones the programmer can (or would want to) edit. Thus adding qualifiers to library header files without editing them should be a simple process.

5.1. Context-aware type qualifier analysis

Recall the example of Figure 1 in which the `nonnull` extension is composed with an algebraic datatype extension. Annotations of `nonnull` on pointers to Boolean expression constructs (`Expr * nonnull`) help to ensure that such expression trees are constructed only from valid variants. The use of pattern matching to extract sub-expressions then is assured that a null dereference will not occur at run-time.

A potential problem arises in the generated plain C code translation. Language extensions specify their translation via forwarding. Algebraic data types translate to tagged unions: collections of C `struct` and `union` types for representing `Expr` values in the expected, but less convenient, manner. A tag, as an enumerated type, in the `struct` indicates which element of the `union` is to be used. Pattern matching translates to `if-else` statements. The code generated from the pattern will inspect these tags and make available the fields of the appropriate union member. A local pointer, named `_current_scrutinee_ptr`, is generated to track the expression being matched and is repeatedly dereferenced in that process. The problem is that the algebraic data type extension, developed independently of the `nonnull` extension, will not qualify this pointer as `nonnull`; errors would be raised on the dereferences to `_current_scrutinee_ptr` since the programmer has no control over the qualifiers used on this generated code.

Language extensions typically check for and raise errors in terms of programmer-written code. Although this is preferable over raising errors on generated code, it is not required. The more serious issue in the example above is that it violates an unenforced invariant asserting that if no errors are found in error-checking the programmer-written code, then there should be no errors on the generated C code.

```
grammar edu:umn:cs:melt:exts:ableC:nonnull;

function suppressError
Boolean ::= loc::Location
{ return  endsWith(".h", loc.filename) || endsWith(".xh", loc.filename)
       || case loc of generatedLoc(_) -> true | _ -> false ;
}

aspect production dereferenceInjectable
e::Expr ::= d::Expr
{ attr msg::Msg = "dereference without 'nonnull'";
  lerrors <- if isNonNullQualified(d.type) || suppressError(e.location)
             then [ ]
             else [ err(e.location, msg) ];
}
```

Figure 11: Context-aware analysis of the `nonnull` qualifier, a revision on Figure 9.

Standard (.h) and extended (.xh) header files are another example of code that might not be possible for the programmer to edit. Similarly, the raising of errors by qualifier extensions on such code that the programmer has no control over can present problems. Instead of generating an error, we might choose to trust that the code not written by the programmer is safe and not perform the null dereference analysis, for example, on headers.

Extension writers can use location information to suppress such errors. Locations in Copper and Silver, and thus ableC, include the original filename and this is maintained by the C preprocessor using # tags. Further, locations are structured data created by either a `loc` or a `generatedLoc` production. Extensions can pattern match on these productions and access original filenames of productions to determine if errors should be suppressed or not, as done by the `suppressError` function in Figure 11. The aspect production for `dereference` from Figure 9 can then be updated to only add error messages when errors are not to be suppressed.

As will be described in Section 7, a possible alternative to suppressing errors in cases where compile-time errors are undesirable may be to generate code to check for such errors at run-time.

5.2. Working with libraries

We now consider issues that may arise through the use of libraries in addition to the need for context-aware error suppression addressed above in Section 5.1. Recall that only library header files presented problems. This is because library source files are compiled with a standard C compiler and thus are not analyzed by the type qualifier extensions of `ableC`.

Extensions may wish to annotate library functions with new qualifiers in order to prevent improper use of such functions. For example, the behavior of standard I/O functions such as `fgets` is undefined when passed a null file pointer, and thus annotating such parameters as `FILE * nonnull` would be beneficial. More seriously, the lack of qualifier annotations on the signature of existing library functions can introduce a gap in the analysis performed by extension qualifiers. For example, not qualifying as `tainted` a string read from a file, again using `fgets`, could unsafely allow such strings to be passed to a function that may be exploited by carefully-constructed input.

This issue is addressed in Cqual by allowing the programmer to annotate library functions with additional qualifiers in a special "prelude" file that is read before and takes precedence over prototypes of the same name in standard headers. Annotations for all new qualifiers on any given function must be together in a single prelude file. This is not satisfactory for our user model in which extension writers develop new qualifiers independently, two or more of whom may wish to add annotations to the same library function. Therefore, ableC allows a function prototype to be declared multiple times with different qualifiers as long as the types are otherwise identical, and accumulates all such qualifiers on each re-declaration. Recall that qualifiers associated with a type are modeled as an unordered set, and so the order of this accumulation is undefined; outstanding issues arising from the ordering of type qualifiers are briefly discussed in Section 9.3 and followed by a discussion in Section 10 of a need for composability guarantees over a notion of semantic composition.

```
#include "nonnull.xh"
           // contains:         char *fgets(        char *s, int size, FILE * nonnull stream);
#include "tainted.xh"
           // contains: tainted char *fgets(tainted char *s, int size, FILE *         stream);

//  combined prototype: tainted char *fgets(tainted char *s, int size, FILE * nonnull stream);

// An original implementation with type qualifier errors.
char *read_from_file(...) {
  FILE *fp = fopen(...);
  char *buf = malloc(...);
  fgets(buf, BUF_SIZE, fp);  // type errors: unsafe tainting of buf, fp might be null
  return buf;
}

// A corrected implementation that adds qualifiers and qualified casts.
tainted char *read_from_file(...) {
  FILE * nonnull fp = (FILE *nonnull) fopen(...);
  tainted char * buf = malloc(...);
  fgets(buf, BUF_SIZE, fp);  // now no type errors
  return buf;
}
```

Figure 12: Composition of `nonnull` and `tainted` qualifier extensions, both of which provide header files that add qualifiers to `fgets`.

Consider the example program of Figure 12 that makes use of an ableC compiler extended with the `nonnull` and `tainted` qualifier extensions. The `nonnull` extension provides a header annotating `fgets` to ensure that the file pointer being read from is not null. Similarly, the `tainted` extension provides a header also annotating `fgets`, in this case to denote that the string comes from an un-trusted source. These prototypes are then composed automatically in a program that includes both headers. Uses of `fgets` by the programmer benefit from both analyses without requiring the programmer to write this combined prototype, as in Cqual.

This figure shows an original implementation of a function to read a file that raises type-qualifier induced error messages. The original call to `fgets` violates the subtyping induced by the `tainted` and `nonnull` annotations on the combined prototype. Below is a corrected version of the function that qualifies the types of `fp` and `buf` appropriately. The cast to `FILE * nonnull` on the value returned from `fopen` ensures that the type qualifiers are consistent with the declaration. As we will see in Section 7, this cast can also inject code to dynamically check that the file pointer is not null.

Since the implementation of `fopen` cannot be changed the programmer should not annotate the return type in the function prototype with `nonnull`. This is an instance in which static checking of `nonnull` cannot be done and dynamic checking is required.

Another issue that may arise through the use of libraries occurs when two language extensions introduce overlapping marking terminals that are used in their respective header files to be included (`#include`) in the programmer's code. Suppose, for example, that in addition to the `tainted` extension presented thus far, the programmer wishes to use a second extension also introducing its own `tainted` qualifier that is used in a header file the programmer is expected to include. Recall that this ambiguity is resolved in ableC through the use of transparent prefixes, requiring the programmer to type a chosen prefix to disambiguate the keywords. Here ableC falls short of our requirement that we be able to use library header files without editing them. The terminal symbols to be disambiguated with a transparent prefix are in library header files - not the programmer's own code. We discuss this issue in Section 10 as remaining as future work.

```
grammar edu:umn:cs:melt:exts:ableC:associative;
import edu:umn:cs:melt:ableC;

abstract production associative
q::Qualifier ::=
{ q.isPositive = false;
  q.isNegative = true;
  q.errors := case q.typeToQualify of
              | functionType(_, _) -> []
              | _ -> ... generate appropriate error ... ;
}
```

Figure 13: Checking that `associative` qualifies only function types.

6. Type-Specific Qualifiers and Type-Independent Qualifiers

Some qualifiers can only be reasonably applied to certain types. For example, `nonnull` only makes sense on pointers, while `pure` and `associative` are intended to qualify functions. In contrast, other qualifiers are useful across a range of types, including types introduced by other extensions. For example, `units(m)` can qualify `float` to represent length, and also can qualify an independently-developed `interval` type to represent a range of lengths. This section discusses how type qualifier extensions can limit their application to certain types.

6.1. Checking errors on type expressions

Qualifiers can limit their application to certain types by raising an error when applied to a disallowed type. The `associative` qualifier, which annotates binary functions, should raise an error on the declaration

```
associative int x;.
```

To support this, we introduce two new attributes on the `Qualifier` nonterminal: `errors` of type `[Msg]` that is non-empty when a declaration violates an extension-specified policy, and an `inherited` attribute named `typeToQualify` that is passed down to a `Qualifier` from the enclosing type and can be inspected to detect errors. Figure 13 shows the specification that `associative` may only be applied to a type matching the pattern `functionType(_, _)`. The `nonnull` specification of Figure 7 can be extended similarly by pattern matching against `pointerType(_)`.

Qualifiers may be more sophisticated in the kind of error checking that they do on type expressions. The dimensional analysis qualifier extension checks that only one `units` qualifier decorates a type. The analysis filters the list of qualifiers on the type (`q.typeToQualify`) and checks if there are two or more that match the pattern `units(_)`. There is no restriction made, however, on the type that is qualified and thus, any type may be qualified by `units(_)`. When used in combination with operator overloading this allows unit qualifiers to qualify numeric types introduced by other extensions, as we see below. Left as future work and discussed further in Section 10 is the ability to restrict such a qualifier only to types that truly are in some sense numeric; we would prefer to prevent `units(_)` from qualifying `String`, for example, but currently have no means of expressing this.

6.2. Type qualifiers and operator overloading

Extensions that introduce new types in ableC can overload operators such as addition, multiplication, and many others. For example, an extension may introduce a new numeric `interval` type representing a range of floating point values and indicate that the addition operator be overloaded at this type, see [2]. The overloaded expression is translated, again via forwarding, to a function call that adds the lower and upper bounds of two intervals. That is,

```
interval[1.2, 3.4] + interval[5.6, 7.8]
```

evaluates to `interval[6.8, 11.2]`.

The `units` qualifier does not restrict the types that it can qualify, and could thus be used to qualify the `interval` type. The `units` qualifier compares qualifiers of types on addition (and other numeric operators) and thus a function to compute the sum over meter-qualified intervals could be written as follows:

```
units(m) interval sum(units(m) interval x, units(m) interval y) { return x + y; }
```

```
grammar edu:umn:cs:melt:ableC;
abstract production addOverloadable
e::Expr ::= l::Expr r::Expr
{ attr lerrors :: [Msg] with ++ ;
  lerrors := [ ];
  attr injectedQualifiers :: [Qualifier] with ++;
  injectedQualifiers := [ ];
  attr runtimeMods :: [RuntimeMod] with ++;
  runtimeMods := [];
  attr modL :: [Expr] = applyMods(runtimeMods, l);
  attr modR :: [Expr] = applyMods(runtimeMods, r);
  forwards to if ! null(lerrors)
              then errorExpr(lerrors)
              else case getAddOverload(l.type, r.type) of
                   | just(p) -> p(modL, modR)
                   | nothing() -> qualifiedExpr(injectedQualifiers, addInjectable(modL, modR)) ;
}
```

Figure 14: Extension points in the ableC host grammar for overloading addition.

```
grammar edu:umn:cs:melt:exts:ableC:dimensionalAnalysis;
aspect production addOverloadable
e::Expr ::= l::Expr r::Expr
{ attr compat::Boolean = unitsCompatible(l.type, r.type);
  lerrors <- if compat then [ ]
             else [ err(e.location, "incompatible units on addition") ];
  injectedQualifiers <- if ! compat then [ ]
                        else [ getUnits(l.type) ];
}
```

Figure 15: Type-independent dimensional analysis error checking. This production is later enhanced in Section 8.

This support for overloading addition by `interval` works in tandem with the dimensional analysis extension's check for unit compatibility on addition.

To see how this cooperation is possible, we consider the overloadable production for addition in the ableC host language. First, the concrete syntax of ableC creates the AST for addition using the `addOverloadable` production, partially shown in Figure 14. Local errors (`lerrors`) are collected and, if there are any, the AST is translated to an erroneous `Expr` production that we have seen before. Otherwise, a function in the ableC host grammar (`getAddOverload`) is called to obtain the production that overloads addition for some new type, wrapped in a *maybe* type. If an overloading production has been registered (a process we elide here) then it is returned wrapped in a `just` production. If `l` and `r` are intervals, this production `p` will be the abstract production for adding interval values. If no overloads exist, this function returns `nothing()` and `addOverloadable` translates via forwarding to the next translation stage `addInjectable`, which in turn translates to the default addition (`add`). We will see in Section 7 how the `runtimeMods` collection attribute is used to generate run-time code.

The default behavior of the addition operator is to drop all qualifiers from the operands. Qualifiers that extensions wish to include in the type of the result must be explicitly added to an `injectedQualifiers` attribute.

The dimensional analysis extension raises unit-incompatibility errors on addition in an aspect production on `addOverloadable` as in Figure 15, making use of the extension points provided by the identically-named abstract production in the host grammar in Figure 14. The elided function `unitsCompatible` performs an analysis to determine if error messages should be added to `lerrors` using the `<-` contribution operator, and also specifies that units qualifiers, if compatible, should be retained on the resulting type of addition. In contrast, the aspect production on multiplication performs no compatibility analysis, and specifies that the qualifiers on the result should be the product of the units on the types of the `l` and `r` operands.

```
// generated from "*p"
({ int * _tmp = p;
   int done = 0;
   if ( _tmp == 0 ) { printf("%s\n", "null dereference");  done = 1; }
   if ( ... _tmp ... ) { printf("%s\n", ... );   done = 1; }
   if ( done ) exit(1);
   * _tmp; })
```

Figure 16: Generated dynamic checks on dereferences.

Because the dimensional analysis error checking is done in an aspect of `addOverloadable`, it is performed before operator overloading and thus is performed even on interval addition as we have shown. Alternatively, creating a similar aspect production instead on `addInjectable`, the next stage of translation, would cause this analysis to be bypassed when addition is overloaded. Thus, error checking for type qualifiers can work on any type (possibly developed by other extensions) that overloads operators. Qualifier extension designers can decide if their qualifier is specific to a type, or not.

7. Dynamic Type Qualifier Checking

In cases where static error checking is not feasible, a possible alternative is for qualifier language extensions to generate code to provide similar checks at run-time. Recall the example composition of `nonnull` with algebraic data types in Figure 1 and the need for suppression of `nonnull` errors on generated code discussed in Section 5.1. As opposed to doing nothing, the `nonnull` extension can generate code to check at run-time if a pointer is null before dereferencing. Another example of the generation of dynamic checks can be seen in a generic `check` qualifier. Here, an inert qualifier in a base extension acts as a platform for further extensions to generate run-time checks of `check`-qualified variables of a certain type. For example, subscripting of a `check`-qualified array can generate simple array bounds checking. Independently, dynamic checks can be generated to ensure that `check`-qualified tagged unions access proper variants.

7.1. Dynamic nonnull checks

Consider the code that needs to be generated to perform arbitrary checks on an expression prior to its dereference. Figure 16 shows sample code generated from the dereference of an integer pointer `p`. Multiple extensions may generate checks on the same expression. Here, the `nonnull` extension verifies that the pointer is not null, and a second extension performs a check that is elided. Thus, care must be taken to avoid evaluating the expression more than once in case it has side effects. We initialize a temporary variable `_tmp` to hold the value of the expression and allow each extension to specify an error message and a conditional over this variable that, if true, will cause the message to be printed and the program to exit after all checks are performed. This code relies on the GCC *statement-expression*, enclosed in `({` and `})`, to allow declarations and if-statements within a dereference expression.

Dynamic checks can be generated on particular operators in a way similar to that used in Section 4.2 to add new errors. Support in ableC for this feature can be seen in the host dereference production of Figure 17, a modification of Figure 8 enhanced with a collection attribute `runtimeMods` to allow run-time modifications to be performed. Such modifications may include the dynamic checks of this section as well as arbitrary code insertion to be seen in Section 8. The process of applying modifications is handled by an `applyMods` function, elided here, that builds appropriate conditional expressions (as seen in Figure 16) for each dynamic check.

As seen in previous examples, extensions can then contribute to these collection attributes in an aspect production. The aspect production on dereference in the `nonnull` extension is updated in Figure 18 to generate dynamic checks. Dynamic checks are constructed using a `runtimeCheck` function that is given an error message and a function from the generated temporary variable to a conditional expression. The run-time checks in Figure 16 are generated by this function and all are then inserted into the translation. In that example, these checks are generated by the `nonnull` extension on pointer dereferences that are not qualified as `nonnull` and are at a location in which static errors are not suppressed.

```
grammar edu:umn:cs:melt:ableC;
abstract production dereferenceInjectable
e::Expr ::= d::Expr
{ attr lerrors :: [Msg] with ++ ;
  lerrors := [];
  attr runtimeMods :: [RuntimeMod] with ++;
  runtimeMods := [];
  forwards to if null(lerrors)
              then dereference(applyMods(runtimeMods, d))
              else errorExpr(lerrors);
}
```

Figure 17: Extension points in the ableC host grammar for injecting run-time code on pointer dereference in ableC, an enhanced version of that shown in Figure 8.

```
grammar edu:umn:cs:melt:exts:ableC:nonnull;

aspect production dereferenceInjectable
e::Expr ::= d::Expr
{ lerrors <- ... as before ...
  runtimeMods <- if isNonNullQualified(d.type) && suppressError(e.location)
                 then [ ]
                 else [ runtimeCheck(checkNull, "null dereference") ];
}

function checkNull
Expr ::= tmp::Expr
{ return equalityOp(tmp, intConst(0)); }

aspect production castInjectable
top::Expr ::= ty::Type e::Expr
{ runtimeMods <- if isNonNullQualified(ty) && !isNonNullQualified(e.type)
                 then [ runtimeCheck(checkNull, "null dereference") ]
                 else [ ];
}
```

Figure 18: Adding dynamic non-null checks on pointer dereference and casting. A revision on Figure 11.

A similar check is generated on casts that add `nonnull` to a type. Recall the initialization

```
FILE * nonnull fp = (FILE * nonnull) fopen(...);
```

of Figure 12. This appears to unsafely disregard the possibility that `fopen` will fail. In fact, due to the generation of a dynamic check, the invariant that a dereference of `fp` will never result in a null dereference at run-time continues to hold. In the lower portion of Figure 18, the injectable version of the cast operator production (`castInjectable`) is used in an aspect production in the nonnull extension. This checks if the type being cast to is qualified with `nonnull` and if the type of the expression being cast is not so qualified. In this case, which is what we see in Figure 12, a run-time check is inserted.

7.2. A generic qualifier for dynamic checks

In this section we describe two types of dynamic checks added by type qualifiers. Both are specified in extensions that build on top of a generic `check` qualifier extension that is extended to check various properties. Here these are that array accesses are within bounds and that tagged unions are accessed in a manner consistent with the value of the tag. The `nonnull` qualifier described earlier could have also been part of this scheme, but we kept it as its own extension to illustrate how a qualifier found in Cqual can be implemented in ableC.

```
// malloc_bc updates a global structure to map x to its allocated size,
//   in this case 20 bytes, assuming 4 byte integers
int * check x = malloc_bc(5 * sizeof(int));

// generated from: x[7] = 1;
x [ ({ int _tmp = 7;
       int done = 0;
       if (_boundsmap_find(x) / sizeof(int) <= _tmp) {
         printf("%s\n", "array subscript out of range\n");
         done = 1;
       }
       if (done) exit(1);
       _tmp;
    }) ] = 1;
```

Figure 19: Code generated on array subscripting by bounds-checking extension.

```
grammar edu:umn:cs:melt:exts:ableC:checkBounds;
import edu:umn:cs:melt:exts:ableC:check;

aspect production arraySubscriptInjectable
e::Expr ::= l::Expr r::Expr
{ attr upperBound :: Expr = ...; -- AST for _boundsmap_find(x) / sizeof(int)
  attr checkBounds :: (Expr ::= Expr) = \tmpRhs :: Expr -> lteExpr(upperBound, tmpRhs);

  runtimeMods <- if isCheckQualified(l.type)
                 then [ runtimeCheck(checkBounds, "array subscript out of range") ];
                 else [ ]
}
```

Figure 20: Adding a dynamic bounds check on array subscripting.

Dynamic array bounds checking. Type qualifiers that generate dynamic checks can be used in a simple approach to prevent array bounds problems in C. This extension creates a custom memory allocation function `malloc_bc` that maintains a global map from allocated pointer addresses to the size allocated. An extension introducing a new qualifier `check` may then generate dynamic checks on array subscript expressions of `check`-qualified pointers to detect if the index is greater than that stored in the global map. An example of such a generated check is shown in Figure 19.

An extension `edu:umn:cs:melt:exts:ableC:check` specifies a `check` qualifier in the same fashion as we have seen with others such as `nonnull`. This function defines an `isCheckQualifier` function to detect if a type is qualified by `check`. The intention is that other extensions can build on top of this extension to provide *dynamic checking* behavior on various types and operators. We can see a use of this in Figure 20. Here the extension for array bounds checking generates dynamic checks on the array subscript operator (`arraySubscriptInjectable`) using an aspect production and a contribution to `runtimeMods`.

A more sophisticated version of this array bounds checking extension may attempt some static analysis to determine if the dynamic checks could be avoided, but even a simple qualifier like this could be useful in testing. The reuse of the `check` qualifier with tagged unions, as described below, will illustrate a use of the kind of static information that is available to extensions. Recall that the API presented to an extension is essentially the nonterminals, productions, and attributes that the host language defines, and that this provides, for example, the ability for an extension to look up the type of an identifier in an inherited attribute representing the environment. More useful to this particular extension would be access to the control-flow graph of the program; although control-flow infrastructure does not currently exist in ableC, we have begun work on this infrastructure and discuss it as future work in Section 10.

Dynamic tagged union checking. Independent extensions may reuse the `check` qualifier to introduce additional analyses. Consider the reuse of the `check` qualifier on tagged unions in Figure 21. Here the Boolean-expression algebraic data type example of Figure 1 is re-implemented in plain C using the standard `struct` and `union` combination with a tag as an enumerated type to indicate which element of the `union` is to be used. Boolean expressions are represented as tagged unions and thus care must be taken to only access the union member corresponding to the current value of the tag. The `check` qualifier is used to ensure this invariant holds at run-time by creating a correspondence between values of the tag (`AND`, `OR`, and `LITERAL`) and the elements of the `union` named `variant` in the `struct Expr`. Dynamic checks are then generated on member access expressions accessing tagged union variants such as `e.variant.and`. The figure contains two examples of erroneous access to `variant` that would be detected by the code generated by the `check` qualifier.

To inject the dynamic checks as seen in Figure 21, the extension provides an aspect production on the field access operator production. A pattern similar to that described above is followed to insert the dynamic checks into the `runtimeMods` collection attribute on the host language production.

This method of generating dynamic checks is similar to that of the bounds checking extensions already seen in Figure 20. What is new here is the complexity in determining the code to be generated.

Member access expressions (*e.g.* `x.fieldname`) that we wish to check dynamically are of the following form. The left-hand side is an access of the member named `variant` on an expression that is a tagged union qualified by `check`. We define a tagged union strictly here to be a struct with a `tag` field of enumerated type and a `variant` field of union type, where the number of enumerated variants is equal to the number of members of the union. To determine if an expression is a tagged union, its type attribute is examined and, if found to be of some struct type, the full declaration of the struct is found by looking up the struct's name in the environment. Information about enumeration variants and union members may then be extracted from this declaration.

If the member access expression is found to meet the conditions above, for example if it is `e.variant.and`, we then wish to generate a conditional expression comparing the value of the `tag` field to the corresponding enumerated variant. The desired variant is computed by examining the tagged union declaration found in the environment, searching for the position of the member name accessed and using the enumerated variant declared in the same position. For example, for the expression `e.variant.and` we find that the `and` field is declared in the same position as the `AND` variant. This can be seen in the generated code in the bottom of Figure 21.

8. Type Qualifier Directed Code Insertion

Type qualifier extensions can generate arbitrary run-time code using methods similar to those used in the generation of the dynamic checks of Section 7. This section presents an enhancement to the dimensional analysis extension to support automatic conversion of unit measurements, and presents a `watch` qualifier used to monitor changing state and that relies in part on independent extensions to determine the code to be generated.

8.1. Automatic conversion of units

A potential application of code insertion comes from the dimensional analysis qualifier. Consider the example in Figure 22 in which the sum of two durations, one in units of seconds and the other in milliseconds, is computed. Rather than raising an incompatible units error, the programmer might instead prefer for the units to be automatically scaled at run-time so that they are in fact compatible.

Support for the injection of run-time modifications on addition expressions, both at the `addOverloadable` (shown in Figure 14) and `addInjectable` translation stages, is similar to that of the dereference expression in Figure 17. Again, a new `runtimeMods` collection attribute is used and the process of applying modifications is handled by the `applyMods` function.

The dimensional analysis extension then may contribute to `runtimeMods` in its `addOverloadable` aspect production, as in Figure 23. The extension determines that a conversion is needed if the units on the types of the operands are compatible but not identical. A new abstract production `rhsRuntimeConversion` is then used to construct a `RuntimeMod` to be collected into `runtimeMods`. The original right-hand operand is then replaced by the `applyMods` function with a scaled expression.

```
struct Expr {
  enum {AND, OR, LITERAL} tag;
  union ExprVariant {
    struct { struct Expr *lhs; struct Expr *rhs; } and;
    struct { struct Expr *lhs; struct Expr *rhs; } or;
    bool literal;
  } variant;
};

bool value (check struct Expr * e) {
  switch (e->tag) {
  case AND:
    return value(e->variant.and.lhs) && value(e->variant.and.rhs);
  case OR:
    // run-time error: copy/pasted from above but did not change to variant.or
    return value(e->variant.and.lhs) || value(e->variant.and.rhs);
  case LITERAL:
    return e->variant.literal;
  }
}

int main (void) {
  check struct Expr true_expr = {LITERAL};
  true_expr.variant.literal = true;

  check struct Expr false_expr = {LITERAL};
  false_expr.variant.literal = false;

  check struct Expr e = {OR};
  // run-time errors: accessing variant.and when tag is OR
  e.variant.and.lhs = &true_expr;
  e.variant.and.rhs = &false_expr;

  printf("true || false = %d\n", eval(&e));
}
```

Generated from `e.variant.and.lhs = &true_expr`:

```
({ union ExprVariant * _tmp = &e.variant;
   int done = 0;
   if ( e.tag != AND ) {
     printf("%s\n", "attempted access of tag-mismatched variant");  done = 1; }
   if ( done ) exit(1);
   _tmp; })->and.lhs = &true_expr;
```

Figure 21: Reuse of the `check` qualifier by a tagged union checking extension.

```
units(s)  float time1 = (units(s) float) 1.8;
units(ms) float time2 = (units(ms) float) 92;
units(s)  float total = time1 + time2;   // generated code for addition: time1 + (time2 * 1E-3)
```

Figure 22: Example use of type qualifier code insertion by the dimensional analysis extension.

```
grammar edu:umn:cs:melt:exts:ableC:dimensionalAnalysis;
aspect production addOverloadable
e::Expr ::= l::Expr r::Expr
{ attr need_convert::Boolean = unitsCompatible(l.type, r.type)
                            && !unitsIdentical(l.type, r.type);
  attr convertUnits::(Expr ::= Expr) = \exprToConvert::Expr ->
                                  multiply ( exprToConvert, getUnitsScaleFactor(l.type, r.type) );
  lerrors <- ... as before ...
  runtimeMods <- if need_convert
                 then [ rhsRuntimeConversion(convertUnits) ]
                 else [ ];
}
```

Figure 23: Scaling of units on addition, an enhanced version of that shown in Figure 15.

It is worth noting that scaling over `interval` values is possible since the process queries if the multiply operation is overloaded by the two types of the components. So if multiply is overloaded by `float` and `interval` types, this will return a `just` value with the production to use that implements this operation.

8.2. Code insertion that builds on other extensions

Recall the `watch` qualifier of Figure 2 which was used to monitor a changing variable. On assignment statements, the expression on the right-hand side is wrapped in a GCC statement-expression that stores the result in a temporary, prints it, and then returns that value. When this qualifier is applied to a function, the `watch` extension generates similar print statements around function calls that display the arguments to the function prior to the call and the return value after. In this section we briefly describe an additional aspect of this extension — the ability to "watch" data passed into and back from function calls — and its implementation.

Figure 24 shows the use of the `watch` qualifier to monitor recursive calls of merge sort over a `vector` of integers. As calls are made to `merge` (which merges two sorted vectors) and `merge_sort`, the values passed in and returned are displayed, as seen in the bottom of the figure. Additionally, a simple `sorted` qualifier with a negative sign is used to enforce the invariant that the two vectors passed to the `merge` function have been sorted. Print statements generated by the `watch` qualifier make use of a `_show_vector_int` function to convert the independently-developed `vector` to a `string`, part of a string extension described below.

Admittedly, the output of the `watch`-generated print statements is verbose and quickly grows unwieldy as the number of `watch` qualifiers in a program increases. A more sophisticated extension might track information about monitored values in order to generate output in a more human-readable format. We have, for example, developed simple systems to monitor this output and generate dynamic call graphs of the calls to the function.

With regard to the implementation of the `watch` qualifier, the `watch` qualifier is neither positive nor negative; it is not signed. Thus no subtype relation is induced and watched values can be passed into functions as arguments to parameters that are not qualified by `watch`, but the changes to the values that occur inside that function are then not printed. A programmer interested in monitoring updates to a variable may wish to be made aware of all such updates, even those that occur in generated code. To accomplish this, the `watch` qualifier does not use the context-aware mechanisms described in Section 5.1 and inserts the same code on programmer-written and extension-generated code so that all changes can be observed. Recall from Section 5.1 that locations are structured data that extensions can pattern match on to determine if they were constructed by either a `loc` or a `generatedLoc` production; thus in the case that a `watch`-generated print statement is generated by some other extension, the line number could be displayed as "generated from line 42", for example.

Use of `watch` type qualifier:

```
sorted vector<int> merge(sorted vector<int> v1, sorted vector<int> v2) <watch> {
  ...
}

sorted vector<int> merge_sort(vector<int> v) <watch> {
  if (v.length <= 1) {
    return (sorted vector<int>) v;
  } else {
    sorted vector<int> v1 = merge_sort(slice(v, 0, v.length / 2));
    sorted vector<int> v2 = merge_sort(slice(v, v.length / 2, v.length));
    return merge(v1, v2);
  }
}

int main(void) {
  vector<int> items = vec<int> [3, 2, 1];
  sorted vector<int> sorted_items = merge_sort(items);
}
```

Translation to C of call to `merge_sort(items)`:

```
({ struct _vector_int _tmp_arg = v;
    printf("sort.xc:51: calling merge_sort(%s)\n", _show_vector_int(_tmp_arg).text);
    struct _vector_int _tmp_result = merge_sort(_tmp_arg);
    printf("sort.xc:51: returning   merge_sort(%s) = %s\n",
      _show_vector_int(_tmp_arg).text, _show_vector_int(_tmp_result).text);
    _tmp_result; });
```

Output:

```
sort.xc:51: calling   merge_sort([3, 2, 1])
sort.xc:42: calling   merge_sort([3])
sort.xc:42: returning   merge_sort([3]) = [3]
sort.xc:43: calling   merge_sort([2, 1])
sort.xc:42: calling   merge_sort([2])
sort.xc:42: returning   merge_sort([2]) = [2]
sort.xc:43: calling   merge_sort([1])
sort.xc:43: returning   merge_sort([1]) = [1]
sort.xc:44: calling   merge([2],[1])
sort.xc:44: returning   merge([2],[1]) = [1, 2]
sort.xc:43: returning   merge_sort([2, 1]) = [1, 2]
sort.xc:44: calling   merge([3],[1, 2])
sort.xc:44: returning   merge([3],[1, 2]) = [1, 2, 3]
sort.xc:51: returning   merge_sort([3, 2, 1]) = [1, 2, 3]
```

Figure 24: The `watch` extension generates code to display vectors by using their `vector`-extension specified `string` representation. A `sorted` qualifier is used on the parameters of a `merge` function.

```
grammar edu:umn:cs:melt:exts:ableC:watch;
import edu:umn:cs:melt:exts:ableC:string;

aspect production assignInjectable
e::Expr ::= l::Expr r::Expr
{ attr insertPrint::(Stmt ::= Expr) = \ tmpRhs::Expr ->
    call("printf", foldExpr([stringLiteral("$file:$line:($l.pp) = %s"), showExprText(tmpRhs)])):
  runtimeMods <- if isWatchQualified(l.type)
                 then [ rhsRuntimeInsertion(insertPrint) ]
                 else [ ];
}

aspect production callInjectable
e::Expr ::= f::Name args::Exprs
{ attr insertPrePrint::(Stmt ::= Exprs) = \ tmpArgs::Exprs ->
    ... similar to insertPrint above, map showExprText over tmpArgs ...;
  attr insertPostPrint::(Stmt ::= Exprs Expr) = \ tmpArgs::Exprs tmpResult::Expr ->
    ...;
  runtimeMods <- if isWatchQualified(l.type)
                 then [ preRuntimeInsertion(insertPrePrint),
                        postRuntimeInsertion(insertPostPrint) ]
                 else [ ];
}
```

Figure 25: Insertion of run-time print statements by the `watch` qualifier.

The generation of print statements on assignment to a `watch`-qualified type follows a process similar to that of the `unit`-conversion code generation of Section 8.1 above. Aspect productions for `assignInjectable` (and `assignOverloadable`) make use of host-language provided productions such as `rhsRuntimeInsertion` as the extension hook for injecting code around the right-hand side of the assignment. Likewise, the `preRuntimeInsertion` (and `post` version) production provide the hooks for injecting code before (or after) a function call.

A complication arises when the `watch` extension is composed with an extension that introduces a new type. It is preferable that the `watch`-generated print of a value of type `interval` or `vector`, for example, be displayed in terms of the programmer-written code rather than its plain C translation. This general issue is addressed by a `string` extension that introduces an overloadable unary `show` operation that extensions can overload for types they introduce that specifies how those values can be converted into a string. The process is not dissimilar to that used to implement the operator overloading discussed in Section 6.2.

The `watch` abstract syntax specification in Figure 25 imports the `string` extension and makes use of this mechanism. Equations in aspect productions of assignment and function call expressions contribute to the `runtimeMods` collection attribute. The `rhsRuntimeInsertion` abstract production is used to generate a print statement to be executed prior to the right-hand side of an assignment, while leaving the expression's value unchanged. Similarly, the `preRuntimeInsertion` and `postRuntimeInsertion` productions are used to insert print statements prior to and after a function call, respectively. The `string` representations of expressions are computed using a `showExprText` function provided by the `string` extension.

The `watch` extension is just one example of how qualifiers can be used to animate the execution of a program. The extension is kept relatively simple by only tracking changes based on variable names, so changes to a value via pointer aliasing do not cause any print statements to display the change. But even in this stage, something like this can be useful; it seems an improvement over the common practice of adding various print statements manually in the code. There are several ways in which qualifiers like this could be used. The `watch` qualifier could come with compile-time directives to turn on the code generation for production versions of the code, but leave them in for debugging purposes. Other similar qualifiers could be used for logging purposes. There are many possibilities. Our contribution is to provide the framework in which these sorts of qualifiers can be easily used by the programmer as they see fit.

9. Related Work

Below we discuss related work in the specific area of type qualifiers, but also more general approaches found in extensible type systems and extensible languages and frameworks.

9.1. Type qualifiers

As mentioned earlier, CQUAL is a tool based on the work of Foster et al. [1, 7] that analyzes C programs that have been extended with user-defined type qualifiers. CQUAL has been shown to be useful in detecting many kinds of errors in real-world programs; these include detecting lock-related bugs in the Linux Kernel [7] and also format string vulnerabilities [5].

In terms of concrete syntax, CQUAL differs from ABLEC by requiring user-defined type qualifiers to begin with a dollar sign. This convention allows the implementation of CQUAL's lexer to be relatively simple, and — because CQUAL itself does not generate a translation to plain C after performing its analysis — this convention eases the task of a separate tool to identify and remove user-defined type qualifiers so the code can be passed to a standard compiler. The downside of this simplicity is that more syntactically expressive qualifiers are not supported, such as the parameterized `units` qualifier for dimensional analysis (Section 4.1); and Foster concedes that the code is more readable without the dollar signs and often omits them in their descriptions of their work [6, Section 5.1].

User-defined type qualifiers in CQUAL introduce a subtyping relation and, optionally, can specify that certain operators must be annotated with certain qualifiers, *e.g.* pointers being dereferenced must be qualified as nonnull. Semantics beyond this, such as more sophisticated errors found by examining the AST directly (Section 4.2), or the generation of run-time code (Sections 7 and 8), are not supported. No method is provided to specify the types that a qualifier is allowed to annotate (Section 6.1). Nor can the content of error messages be specified as in ABLEC.

CQUAL allows qualifiers to be polymorphic by prefixing their names with an underscore, as in the function signature `$_1 int foo($_1 int)`. Although an ABLEC templating extension, in the model of C++, supports polymorphism of types, we do not support polymorphism at the qualifier level. CQUAL also supports the specification of more-complex constraints among polymorphic qualifiers. For example,

```
char $_1_2 *strcat(char $_1_2 *dest, const char $_2 *src);
```

specifies that the qualifiers other than `const` on `src` must be a subtype of the qualified `dest`, which is identical to the qualified return type.

No consideration is made by CQUAL for the composition of independently-developed type qualifiers. Although the chance of two qualifier names conflicting is low, if such conflict occurs CQUAL provides no mechanism to address it such as the transparent prefixes of ABLEC (Section 3.2). A more likely area of conflict is in the annotation of library functions in so-called prelude files that are similar but take precedence over regular `.h` files. A programmer wishing to use two or more independent qualifiers that annotate the same library function must manually rewrite these function headers to use all relevant qualifiers. In contrast, ABLEC automates this process by accumulating qualifiers for each header file that is processed (Section 5.2).

9.2. Extensible type systems

Pluggable types systems, such as the Checker Framework [8, 17] and JavaCOP [18, 19], allow multiple user-defined type systems, expressed via type qualifiers, to refine the built-in type system of a language. Like the type qualifiers of CQUAL, pluggable types provide a means for finding additional compile-time type errors, and have no effect on the run-time semantics of the program. In addition, more-expressive static type checking can be specified in terms of a program's AST and dataflow.

In introducing pluggable types, Bracha [20] argues that for multiple type systems to be used interchangeably or simultaneously in a language, it is important that the type systems be neither syntactically nor semantically required, *i.e.* that type annotations are not mandated and have no effect on run-time semantics. While in our framework it is possible for extension writers to restrict themselves to such *optional type systems*, we neither restrict nor advocate any particular approach. Users are free to use, or not use, extensions that affect the run-time semantics of their programs.

The Checker Framework. The Checker Framework allows users to implement pluggable type checkers as Java plug-ins. It supports subtyping, flow-sensitivity, and qualifier polymorphism. Examples of checkers implemented in the Checker Framework include nullness, fake enumeration, and units *i.e.* dimensional analysis checkers among many others [17].

The concrete syntax of type qualifiers in the Checker Framework make use of Java's annotation syntax. Each checker is then able to select the annotations of interest and ignore the others. These annotations support parameters, though in a manner less expressive than what was demonstrated with our sub-language in `unit` qualifiers. Use of these annotations means that type qualifiers begin with an 'at' symbol (`@`), an inconvenience similar to CQUAL's requirement that they begin with a dollar symbol.

Although a distinction is made between extension developers and programmers, consideration is not made for the reliable composition of independently-developed extensions. Reuse of an annotation between two type systems can lead to a situation in which some programs cannot be successfully type-checked under both systems. It is possible for difficulties to arise from the composition of independently-developed annotated libraries, as the following three approaches are taken to annotating libraries with qualifiers: 1) suppression of warnings from un-annotated libraries, 2) so-called stub files that annotate method signatures but not bodies, analogous to the prelude files of CQUAL, and 3) annotation and recompilation of the library source.

JavaCOP. JavaCOP is also a pluggable type system for Java. It is implemented as an extended version of the `javac` compiler, with an API for describing semantic rules for user-defined types. The modified compiler calls into the API in a new pass that is run prior to code generation. Type-checking extensions are specified in a declarative domain-specific language as rules which specify constraints to apply to certain constructs in the abstract syntax. This is more convenient than the attribute grammar equations in ABLEC but also more limiting.

9.3. Aspect-oriented programming

There are similarities between Aspect-oriented programming (AOP) [21] and some work presented here, especially the mechanism for injecting code as seen in the `watch` qualifier and in the dimension analysis qualifier when it injects code for scaling values. AOP is similar to many approaches to writing modular programs in that it attempts to separate different concerns of a program into modules. What distinguishes it from others is the recognition that some concerns will be spread across several modules and cannot be easily isolated into a single module. These features are written as *aspects* over the underlying program modules and an AOP system *weaves* code from these aspects into that program. A high-water mark in these efforts was ASPECTJ [22], a language that added various AOP features to Java, but other languages languages were extended with notions of aspects, for example C in ASPECTC [23]. In ASPECTJ, the most important parts of an aspect are the code and the *join points*, a specification of the underlying program points into which the code is to be woven. Aspects also specify if the aspect code is to be woven *before*, *after*, or *around* the join point. For example, a join point might be all calls to methods whose name matches the regular expression `set*`. An aspect may then specify some logging operation to take place before a set-method is called to record the previous value that is intended to be set by the method call.[3] This was sometimes called *implicit invocation* [27] since the underlying program was meant to be *oblivious* to the aspects that might be woven into it.

Aspects, at least in ASPECTJ and ASPECTC, and our qualifiers that inject code both suffer from a similar problem of interference or unexpected interaction of separate aspects or qualifiers. When arbitrary imperative code can be woven or injected into the underlying program there is no guarantee that the code from different aspects or extensions will be independent. For example, if a variable is qualified by `watch` and `units(m)` how does any scaling of values injected into the code affect what values are printed out? Is the printed value the one before or after the scaling has been performed? Similar questions arise in AOP systems. What is needed is a mechanism for language extensions of all forms, not just for type qualifiers, to specify what meta-theoretic properties they bring to the host language and

[3] As an aside, we note that ASPECTJ presents some challenges to scanner and parser generators. The original ASPECTJ implementation used a parser with some hand-written components. The AspectBench compiler [24] use a scanner with hand-written modes. Bravenboer et al. [25] provide the first *declarative* specification of ASPECTJ using their scannerless generalized LR parsing framework. Using context-aware scanning in COPPER we were able to provide the first declarative and *deterministic* specification. Context-aware scanning could distinguish terminals based on context and thus an LALR(1), and thus unambiguous, grammar could be written for the language [11, 26].

an assurance that those properties are still valid when other independent language extensions are composed. This is briefly discussed in Section 10.

In another connection, the `aspect` productions in SILVER get their name from AOP. These productions associate new attribute equations with existing concrete or abstract productions and thus cut across the primary structure of grouping equations with productions to allow them to be introduced in other modules.

9.4. Extensible languages and language frameworks

More generally, the area of extensible languages and language frameworks aims to allow a much wider range of language features to be added to a base language than the techniques described above, which are aimed extensions to the type system. Among the many varieties of systems are SugarJ [28] and MetaBorg [29] frameworks for extending Java, the JastAdd [30] extensible Java Compiler [31], and the Xoc [32] and XTC [33] frameworks for extending C, support the composition of independently-developed language extensions but lack the guarantees of composability as provided by the MDA and MDWA, though without the restrictions imposed by these analyses more expressive language extensions can be specified.

There are a few other systems that do provide guarantees of composability similar to the MDA and MWDA analysis, but they give up some of the expressibility that is possible with ABLEC. For example, mbeddr [34], Wyvern [35], and VerseML [36] do not suffer from challenges in parsing composed languages since mbeddr uses a projectional editor and thus there is no parsing phase. The Wyvern and VerseML languages use a form of "structured literals" to limit the parsing problem to areas between a pair of many pre-defined beginning and end markers, such as "`{`" and "`}`". But these do not provide the ability to add new semantics to host language constructs, for example to provide a global analysis or translation to another language that have been found useful in some language extensions. DeLite [37] uses a type-based staging approach but is limited syntactically since its extensions are limited to the concrete syntax of Scala. This is actually rather flexible — for example, Scala allows punctuation and operator characters to be used as method names and the dot in method invocation is optional — but is more limited than what is possible with ABLEC.

10. Discussion and Future Work

In this paper we have presented a reformulation of type qualifiers as composable language extensions. We have also extended the notion of type qualifiers in several ways. A crisp distinction is made between the language extension developers and the end-user programmer. While extension developers need some knowledge of the underlying tools, programmers do not; they can easily import their chosen set of language extensions with the assurance that their composition will be successful. Our extensions to previous work provide for more syntactically expressive type qualifiers, as can be seen in the `units` qualifier for dimensional analysis (Section 4.1). We also provide more semantically expressive qualifiers. These can read and define attributes on the AST for rather general error checking; they can also insert code into the final translation of the extended C program, see Sections 7 and 8.

Constructs similar to those presented here have been implemented monolithically as built-in features of — as opposed to extensions to — languages. For example, the F# programming language includes built-in support for dimensional analysis by allowing values to be associated with units of measure. Osprey [38] adapts the parser of CQUAL to add concrete syntax for base and derived SI units as type qualifiers, and to perform dimensional analysis. These demonstrate the desire for such features and the need for language extension capabilities more generally.

Empirical evaluation. This paper aims to show that interesting and expressive qualifiers can be implemented as extensions — not that any specific qualifier is good or useful. We leave that for programmers to determine, much as programmers decide to use or not use a particular library. We would like to do some empirical analysis of the use of extensions more broadly, not only for type qualifiers as extensions, but have not yet undertaken this effort.

Debugging ABLEC *programs.* The entirety of the communication between ABLEC and GCC consists of calling the traditional C compiler to compile the generated plain C code; no interoperability with mainstream C toolchains is available beyond this. Although ABLEC and its extensions can ensure that type errors are reported on the programmer-written code, there is no support for debugging in terms of programmer-written code and this remains future work.

Handling overlapping marking terminals in included header files. Recall that the single caveat of the composability guarantees of the MDA is that lexical ambiguities between marking terminals need to be disambiguated through the use of transparent prefixes. In most cases this is an acceptable, straightforward solution similar to the method a Java programmer uses to disambiguate two classes of the same name. But this is not satisfactory when different libraries, specifically the header file to be included from those different libraries, were written using language extensions that have overlapping marking terminals. Doing so would require the programmer to modify third-party library code, for example the header files as discussed in Section 5.2. The need to write a prefix on one extension's marking terminal can be avoided by selecting it as the default, but this default is compiler wide. A solution to this problem would require the ability for the programmer to specify marking terminal defaults per included file or extension. Copper does process the information that the C pre-processor adds when including files to keep track of name of the included file so that accurate error messages can be generated by traditional C compilers. Copper could take this information into account and only scan for (non-conflicting) marking terminals specific to the included file. But this remains as future work.

Flow sensitivity and qualifier polymorphism. While ableC provides some capabilities not present in other work, the reverse is also true. There are some capabilities of systems like Cqual that we have not yet implemented and are left as current and future work. Currently we have a very limited implementation for flow-sensitive type qualifiers inference based only on the syntactic structure of the program, not on the structure of control and data flow. This has been found to reduce the number of qualifiers that programmers need to type in their programs and thus increases the utility of this general approach, but it is still limited. We are currently designing a more general form of control-flow analysis in ableC that can be used for multiple purposes. The first is for flow-sensitive qualifier inference. But our control-flow framework can also be used by the host language, and language extensions, to detect optimization opportunities in the program. It is aimed as a general purpose infrastructure, much like the symbol table in ableC, that extensions can utilize in a number of different ways.

ableC also lacks support for qualifier polymorphism. There is an extension that introduces C++-style templates that provides some polymorphism at the type level, so a `sum` function such as

```
template<a>  a sum (a x, a y) { return x + y; }
```

correctly requires that, when adding types qualified by the `units` dimensional analysis qualifier, the qualifiers on the inputs are the same and are also what is returned. But this is not sufficient for multiplying units in which case there are no restrictions on the inputs but a new qualifier for the output type must be computed. This is another point of future work.

Hygiene. Note that ableC provides no built-in support for avoiding name capture in the generated C code such as that provided by hygienic macros [39]. Name clashes between extensions can be prevented by following a convention of naming generated variables based on the fully qualified grammar name, which should be unique, *e.g.* `edu_umn_cs_melt_exts_ableC_nonnull_tmp`. This is unsatisfactory and proper support for hygiene remains as future work.

A need for type classes in ableC. An outstanding issue arising from combination of the `units` qualifier and operator overloading remains as future work. On a `units`-qualified multiplication expression, the dimensional analysis extension injects the product of the units on the operands as a qualifier on the type of the result. For example, multiplying two floats qualified as `units(m)` gives a result qualified as `units(m^2)` as expected. This behavior may also make sense on extension-defined types. Suppose, for example, an extension introduces a `FloatList` type that overloads multiplication with a float to map scalar multiplication over the list. Then the result of multiplying a `FloatList` and a float both qualified as `units(m)` should again give a result qualified as `units(m^2)`. But the injection of qualifiers by the dimensional analysis extension is syntax directed, and it is possible for an extension to overload the syntactic `*`-operator with behavior that does not "mean" multiplication, for example if it were overloaded on `FloatList` to perform `cons` (prepend the element to the front of the list). To address this, we would need some way to indicate that the `units` qualifier only applies to "numeric" operations and for new types (such as `interval`) and operations to indicate that they are, in some sense, numeric. Types may then be in a `Num` numeric type class, and the dimensional analysis extension could detect this and behave accordingly.

Semantic composition of extensions. Even though the MDA and MWDA analyses ensure that independently-developed extensions can compose, this is essentially a syntactic notion of composition; it only guarantees that the composition is well defined and the generated compiler will not terminate abnormally. We are currently investigating different notions of semantic composition that ensures meta-theoretic properties of language extensions are maintained when composed with other extensions. For example, the `nonnull` qualifier may come with a verified property stating that there are no run-time dereferences to null pointers. We would like this property to continue to hold when other extensions are composed with it and the host language. The aim is that language extensions be able to specify (and prove) relevant semantic properties with the assurance that other extensions (that satisfy some form of semantic composability restrictions) do not invalidate those properties. One approach to this requires a strict semantic equivalence between an extension language construct and the construct that it forwards to [40]. Another loosens this restriction and instead requires extension constructs to preserve the properties of the constructs it forwards to, but allows additional properties to hold on the extension. Both of these approaches have proven useful in early investigations of semantic composition, but more work needs to be done to better understand the implications to extension design and the scale up to full-featured languages like C.

In our experience developing language extensions we have, on occasion, come across the need to add extension points to the ableC host language, for example to support operator overloading and the injection of generated code. Although this required modifications to be made to ableC, future extensions can reuse these extension points and be implemented as pure extensions with no further modifications to ableC itself. We expect the implementation of features such as flow sensitivity, qualifier polymorphism, and type classes to similarly involve modifications to the ableC host language that then may be reused more broadly.

With these additions we hope to explore more applications of type qualifiers as a syntactically lightweight way to analyze programs and generate code. These type qualifier extensions are a nice complement to the other kinds of extensions, such as the algebraic data type extension, that have previously been the more common form of extension and we continue to explore the opportunities possible with both.

Acknowledgments

This material is partially based upon work supported by the National Science Foundation (NSF) under Grant No. 1628929. 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 thank the members of the Minnesota Extensible Language Tools (MELT) research group at the University of Minnesota for their valuable contributions to the tools and theory underlying the work presented here and for discussions that led to improvements in this paper; in particular, we thank Ted Kaminski for his development and support of ableC, and Lucas Kramer for pointing out the issues arising from the combination of type qualifiers and operator overloading that lead to a need for a concept of type classes in ableC. We thank the reviewers from the 2017 ACM SIGPLAN Conference on Generative Programming: Concepts & Experience (GPCE 2017) and from COMLAN for their comments and suggestions for improving this paper. We also thank James Noble for pointing out the connection between aspect-oriented programming and type qualifiers that inject code into the underlying program.

References

[1] J. S. Foster, M. Fähndrich, A. Aiken, A theory of type qualifiers, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM, 1999, pp. 192–203.

[2] T. Kaminski, L. Kramer, T. Carlson, E. Van Wyk, Reliable and automatic composition of language extensions to C: The ableC extensible language framework, Proceedings of the ACM on Programming Languages 1 (OOPSLA) (2017) 98:1–98:29.

[3] E. Van Wyk, L. Krishnan, A. Schwerdfeger, D. Bodin, Attribute grammar-based language extensions for Java, in: Proceedings of the European Conference on Object Oriented Programming (ECOOP), Vol. 4609 of LNCS, Springer, 2007, pp. 575–599.

[4] T. Carlson, E. Van Wyk, Type qualifiers as composable language extensions, in: Proceedings of the International Conference on Generative Programming: Concepts & Experience (GPCE), ACM, 2017, pp. 91–103.

[5] U. Shankar, K. Talwar, J. S. Foster, D. Wagner, Detecting Format String Vulnerabilities with Type Qualifiers, in: Proceedings of the 10th USENIX Security Symposium, 2001, pp. 201–218.

[6] J. S. Foster, Type Qualifiers: Lightweight Specifications to Improve Software Quality, Ph.D. thesis, University of California, Berkeley (December 2002).

[7] J. S. Foster, T. Terauchi, A. Aiken, Flow-Sensitive Type Qualifiers, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM, 2002, pp. 1–12.
[8] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, M. D. Ernst, Practical pluggable types for Java, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), ACM, 2008, pp. 201–212.
[9] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Science of Computer Programming 75 (1–2) (2010) 39–54.
[10] E. Van Wyk, A. Schwerdfeger, Context-aware scanning for parsing extensible languages, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM, 2007, pp. 63–72.
[11] A. Schwerdfeger, E. Van Wyk, Verifiable composition of deterministic grammars, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM, 2009, pp. 199–210.
[12] T. Kaminski, E. Van Wyk, Modular well-definedness analysis for attribute grammars, in: Proceedings of the International Conference on Software Language Engineering (SLE), Vol. 7745 of LNCS, Springer, 2012, pp. 352–371.
[13] A. Aho, R. Sethi, J. Ullman, Compilers – Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.
[14] H. Vogt, S. D. Swierstra, M. F. Kuiper, Higher-order attribute grammars, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM, 1989, pp. 131–145.
[15] D. Evans, Static detection of dynamic memory errors, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM, 1996, pp. 44–53.
[16] E. Van Wyk, O. de Moor, K. Backhouse, P. Kwiatkowski, Forwarding in attribute grammars for modular language design, in: Proceedings of the 11th Conference on Compiler Construction (CC), Vol. 2304 of LNCS, Springer-Verlag, 2002, pp. 128–142.
[17] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, T. W. Schiller, Building and using pluggable type-checkers, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM, 2011, pp. 681–690.
[18] C. Andreae, J. Noble, S. Markstrum, T. Millstein, A framework for implementing pluggable type systems, in: Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, 2006, pp. 57–74.
[19] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, J. Noble, JavaCOP: Declarative pluggable types for Java, ACM Transactions on Programming Languages and Systems (TOPLAS) 32 (2) (2010) 4:1–4:37.
[20] G. Bracha, Pluggable type systems, in: OOPSLA Workshop on Revival of Dynamic Languages, 2004.
[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Aksit, S. Matsuoka (Eds.), ECOOP'97 Object–Oriented Programming, Vol. 1241 of LNCS, 1997, pp. 220–242.
[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: J. L. Knudsen (Ed.), ECOOP 2001 Object–Oriented Programming, Vol. 2072 of LNCS, 2001, pp. 327–353.
[23] Y. Coady, G. Kiczales, M. Feeley, G. Smolyn, Using aspectC to improve the modularity of path-specific customization in operating system code, in: Proceedings of the European Software Engineering Conference Held Jointly with International Symposium on Foundations of Software Engineering (ESEC/FSE), ACM, 2001, pp. 88–98.
[24] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Transactions on aspect-oriented software development i, Springer-Verlag, 2006, Ch. Abc: An Extensible AspectJ Compiler, pp. 293–334.
[25] M. Bravenboer, E. Tanter, E. Visser, Declarative, formal, and extensible syntax definition for aspectJ, in: Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA), ACM, 2006, pp. 209–228.
[26] A. Schwerdfeger, Context-aware scanning and determinism-preserving grammar composition, in theory and practice, Ph.D. thesis, University of Minnesota, Department of Computer Science and Engineering, Minneapolis, Minnesota, USA, available at `http://purl.umn.edu/95605` (2010).
[27] R. E. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison-Wesley, 2005.
[28] S. Erdweg, T. Rendel, C. Kastner, K. Ostermann, SugarJ: Library-based syntactic language extensibility, in: Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA), ACM, 2011, pp. 391–406.
[29] M. Bravenboer, E. Visser, Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions, in: Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA), ACM, 2004, pp. 365–383.
[30] T. Ekman, G. Hedin, The JastAdd system - modular extensible compiler construction, Science of Computer Programming 69 (2007) 14–26.
[31] T. Ekman, G. Hedin, The JastAdd extensible Java compiler, in: Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA), ACM, 2007, pp. 1–18.
[32] R. Cox, T. Bergany, A. Clements, F. Kaashoek, E. Kohlery, Xoc, an extension-oriented compiler for systems programming, in: Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2008, pp. 244–254.
[33] R. Grimm, Better extensibility through modular syntax, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM, 2006, pp. 38–51.
[34] M. Voelter, D. Ratiu, B. Schaetz, B. Kolb, Mbeddr: An extensible C-based programming language and IDE for embedded systems, in: Proceedings of the Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH), ACM, 2012, pp. 121–140.
[35] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, J. Aldrich, Safely composable type-specific languages, in: Proceedings of the European Conference on Object Oriented Programming (ECOOP), Springer, 2014, pp. 105–130.
[36] C. Omar, Reasonably programmable syntax, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, USA (2017).
[37] T. Rompf, M. Odersky, Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs, in: Proceedings of the Conference on Generative Programming and Component Engineering (GPCE), ACM, 2010, pp. 127–136.
[38] L. Jiang, Z. Su, Osprey: A practical type system for validating dimensional unit correctness of c programs, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM, 2006, pp. 262–271.
[39] E. Kohlbecker, D. P. Friedman, M. Felleisen, B. Duba, Hygienic macro expansion, in: Proceedings of the Conference on LISP and Functional Programming, ACM, 1986, pp. 151–161.
[40] T. Kaminski, E. Van Wyk, Ensuring non-interference of composable language extensions, in: Proceedings of the International Conference on Software Language Engineering (SLE), ACM, 2017, pp. 163–174.