Type Qualifiers as Composable Language Extensions

Travis Carlson
Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA
travis.carlson@cs.umn.edu

Eric Van Wyk
Computer Science and Engineering
University of Minnesota
Minneapolis, MN, USA
evw@umn.edu

Abstract

This paper reformulates type qualifiers as language extensions that can be automatically and reliably composed. Type qualifiers annotate type expressions to introduce new subtyping relations and are powerful enough to detect many kinds of errors. Type qualifiers, as illustrated in our ableC extensible language framework for C, can introduce rich forms of concrete syntax, can generate dynamic checks on data when static checks are infeasible or not appropriate, and inject code that affects the program's behavior, for example for conversions of data or logging.

ableC language extensions to C are implemented as attribute grammar fragments and provide an expressive mechanism for type qualifier implementations to check for additional errors, *e.g.* dereferences to pointers not qualified by a "nonnull" qualifier, and report custom error messages. Our approach distinguishes language extension users from developers and provides modular analyses to developers to ensure that when users select a set of extensions to use, they will automatically compose to form a working compiler.

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

Keywords type qualifiers, type systems, pluggable types, extensible languages, dimensional analysis

ACM Reference Format:
Travis Carlson and Eric Van Wyk. 2017. Type Qualifiers as Composable Language Extensions. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 13 pages.
https://doi.org/10.1145/3136040.3136055

GPCE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136055

1 Introduction and Motivation

Type qualifiers in C and C++, such as `const` or `restrict`, enable the programmer to disallow certain operations on qualified types and to indicate simple subtype relationships between types. For example, only initializing assignments are allowed on a variable declared with a `const` qualifier, and a function with an argument declaration `int *x` cannot be passed a value of type `const int *` as this would allow changes to the `const int` via the pointer. The type `int *` is considered to be a subtype of `const int *`.

In their seminal paper "A Theory of Type Qualifiers", Foster et al. [12] formalize this subtyping relationship and show how user-defined type qualifiers can be added to a language to perform additional kinds of checking. One example is a `nonnull` qualifier for pointer types to indicate that the value of the pointer is not null. A pointer `p` declared as

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

can be passed to a function as a parameter declared to have type `int *`, but the reverse of passing an `int *` value as a parameter with type `int * nonnull` is disallowed. Furthermore, dereferences to pointers whose type is not qualified as `nonnull` raise errors. On the other hand, the qualifier `tainted` induces a different subtype relationship in which the type `char *` is a subtype of `tainted char *`, thus preventing a value with this qualified type to be used where an unqualified one is expected. These subtype relationships, which form a lattice of qualified types, are described in more detail in Section 2. In their paper, 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 this paper we reformulate these and other type qualifiers as language extensions in the ableC extensible language framework for C [17]. In the approach, 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 users of extensions do not.

```
typedef  datatype Expr  Expr;
datatype Expr {
  Add (Expr * nonnull, Expr * nonnull);
  Mul (Expr * nonnull, Expr * nonnull);
  Const (int);
};

int value (Expr * nonnull e) {
  match (*e) {
    Add (e1, e2) -> { return value(e1) + value(e2); }
    Mul (e1, e2) -> { return value(e1) * value(e2); }
    Const (v) ->    { return v; }
  }
}
```

Figure 1. Algebraic datatype and `nonnull` extensions.

Language extension specifications for ableC are written, as described in Section 3, as context free grammars (for concrete syntax) and attribute grammars (for semantic analysis and code generation). The underlying tools that process these specifications provide *modular* analyses of language 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 relieves the extension users (programmers) of needing to know about the underlying tools and techniques and thus frees extension designers 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.

While this requires more sophistication of the extension developer, this approach does provide them with the tools for writing more syntactically and semantically expressive type qualifiers than possible in other approaches. Syntactically, type qualifiers can define their own sub-language, *e.g.* `units(kg*m^2)` which specifies a SI measurement in newtons. Another example is the sample program in Figure 1 that uses two language extensions: one introducing algebraic datatypes similar to those in Standard ML and Haskell, and another introducing a `nonnull` qualifier like the one described above. Here the algebraic datatype extension is used to declare a `datatype` for simple arithmetic expressions and to write a function to compute their value. This `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 but they can also introduce code into the C program that is generated from the extended ableC program. These can be dynamic checks of data when static checks are not possible

Use of `watch` type qualifier:

```
watch int sum = 0;
for (int i=1; i < 4; ++i) { sum = sum + i;  }
```

Translation to C of assignment to `sum`:

```
sum = ({ int tmp = sum + i;
         printf("example.xc:3: (sum) = %d\n", tmp);
         tmp;  });
```

Output:

```
example.xc:1: (sum) = 0
example.xc:3: (sum) = 1
example.xc:3: (sum) = 3
example.xc:3: (sum) = 6
```

Figure 2. Code insertion by `watch` type qualifier.

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.

Contributions: This paper reformulates type qualifier work of Foster et al. [12] in an extensible setting with guarantees of composability for independently-developed type qualifier extensions, extended with a more expressive mechanism for specifying new static checks based on type qualifiers and handling of parameterized type qualifiers (Section 3).

We describe a refactoring and extension of previously ad-hoc handling of type qualifiers in ableC to support:

- Mechanisms for distinguishing behavior of type qualifier checking on code written by a programmer versus code generated by a language extension or included by the C preprocessor (Section 4.1).
- A technique for automatically combining type qualifier annotations to library headers from multiple extensions to alleviate a manual process in Cqual, Foster et al.'s implementation of user-defined type qualifiers in C (Section 4.2).
- 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 5).
- Qualifiers that add dynamic checks of data when static checks are not feasible or appropriate (Section 6).
- Qualifiers that perform runtime conversion of data or otherwise insert additional code (Section 7).

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, the `watch` qualifier, and data conversion (*e.g.* scaling values in millimeters to meters in the dimension analysis extension).

Section 8 discusses related work before Section 9 discusses some future work and concludes.

2 Background

Our work reformulates much of the previous work on type qualifiers to implement them as composable language extensions and then extends them so that qualifiers have considerable code-generation capabilities. Thus we review this previous work and the ableC extensible language framework in this section.

2.1 Positive and Negative Type Qualifiers in C

Type qualifiers in Foster et al. [12] are associated with a *sign* that is used to induce subtype relationships on qualified types. The sign of their `nonnull` qualifier is *negative*, meaning that for some type τ, the qualified type `nonnull` τ is a subtype of τ. Thus a `nonnull` pointer can be used any place an unqualified pointer can be used, but not the reverse. An example of a *positive* qualifier is `tainted`, intended to represent data that may come from a possibly-malicious user and so should not be passed to exploitable functions, which has been shown to be effective in detecting format string vulnerabilities [26]. 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. Qualifiers then form a lattice of subtypes [12, Section 2] so that, for a type τ and a positive qualifier pq and negative qualifier nq the following subtyping relations hold: $nq\ \tau \preceq \tau$, $\tau \preceq pq\ \tau$, $nq\ pq\ \tau \preceq pq\ \tau$, and $nq\ \tau \preceq nq\ pq\ \tau$.

A pointer type may contain qualifiers on the pointer type itself as well as on the type that is pointed to. To determine whether one qualified pointer is a subtype of another, first the outer-level qualifiers on the pointers 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, so-called deep subtyping. Instead, they must be checked for equality. For 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 [11, Section 3.6].

When used as an l-value, a variable in C refers to the memory location where the value of that variable is stored. When used as an r-value, 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—not that the value stored at that memory location 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—not that the memory location itself 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* [11, Section 5.2].

2.2 Additional Checking

While the subtypes induced by qualifiers are checked throughout the program, there are some constructs in which additional checks are added. For example, the pointer dereference operator `*` performs an additional check to verify that the pointer being dereferenced is qualified as `nonnull`. If it is not, an error message is raised. This ensures that all pointer references are done safely.

Cqual allows users to define these sorts of checks in a so-called *prelude* file. Users would specify these by writing with which qualifiers the arguments to operators must be annotated. For example, to require a that `nonnull` qualifier on pointer dereferences the user would write the following: `$$a _op_deref($$a *$nonnull);`

2.3 Flow-Sensitive Type Qualifiers

Flow sensitivity allows a type system to infer different type qualifiers for a variable at different program points [13]. It may be possible to know at certain points that a pointer cannot possibly be null, for example, but not at others. For 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 [23].

2.4 ableC, Attribute Grammar–Based Language Extensions for C

ableC [17] is a specification of C, at the C11 standard, using the Silver [27] attribute grammar system and the Copper [29] parser and context-aware scanner generator embedded in it. 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). 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.

The composed language specifications defines a translator that scans and parses extended C programs (which have the `.xc` or `.xh` extension), constructs an abstract syntax tree (AST), performs type-checking and potentially generates error messages, and finally translates the extended C program down to plain C to be output and compiled using a traditional C compiler. Extensions can perform error checking on the initial untranslated AST and thus generate error messages that refer to the programmer-written code.

Of special interest are the modular determinism analysis [25] (MDA) in Copper and the modular well-definedness analysis [18] (MWDA) in Silver that provide guarantees on the composability of independently-developed language extensions. While these impose some restrictions on language extensions, we have found that they still allow quite expressive and useful extensions to be specified [17].

The MDA is an analysis run by a language extension developer on the concrete syntax specifications of their extension, with respect to the host language being extended. If the analysis fails, the extension developer can make adjustments to their extension to fix the composition problems. This 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 scanner specification with no lexical ambiguities and a parser specification—a context free grammar—that is in the LALR(1) class, meaning that there are no ambiguities and a deterministic parser can be constructed for it [1]. One of the restrictions of the MDA on concrete syntax that ensures composability of concrete syntax specifications 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 a so-called "marking" terminal. Another restriction is that extension productions cannot extend the *follow* sets [1] of host language nonterminals; that is, they cannot specify that new non-marking terminals now follow host language nonterminals in valid programs. 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. These ambiguities are easily resolved by the programmer using a mechanism called *transparent prefixes* [25]. This amounts to specifying a prefix for each marking terminal to be typed before that keyword in the program disambiguate it. It is similar to using full package names in Java programs when two packages both define a class with the same name.

The MWDA provides a similar modular guarantee for the static semantics specified as attribute grammar fragments. Specifically, it ensures that the composition of any collection of extensions that pass the MWDA independently will form a *well-defined* [31] attribute grammar. This ensures that attribute evaluation will not terminate abnormally because of a missing attribute-defining equation.

```
int square (int x) pure { return x * x; }

int add (int x, int y) pure associative
{ return x + y; }

int vector<int> square_all (vector<int> xs)
{ return map (square, xs); }

int sum (vector<int> xs) { return fold(add, 0, xs); }
```

Figure 3. Use of `pure` and `associative` type qualifiers with `map` and `fold` parallel programming extension.

3 Type Qualifiers as Extensions in ableC

This section describes how type qualifiers, in the spirit of those in Cqual, can be implemented as composable language extensions in the ableC extensible language framework. The following sections describe our enhancements and contributions to that line of work.

Motivating examples featuring the use of type qualifiers together with other extensions include the algebraic data type example in Figure 1 and the qualifiers `pure` and `associative` as applied to function types in Figure 3. These are used in a parallel programming language extension that introduces `map` and `fold` constructs that generate parallel implementations of these common functional programming concepts.

Prior to this work, ableC supported the standard C qualifiers in an ad-hoc manner. The ableC host language specification was modified to treat 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 that this one-time enrichment of the host language specification has been made, future type qualifiers can be implemented as pure extensions with no further modifications needed to ableC itself, and non-qualifier extensions can also make use of these code-generation extension points.

3.1 Concrete Syntax for Qualifiers

Figure 4 shows several parts of the implementation of the `nonnull` [10] qualifier as an ableC extension, written in Silver. Extension specifications indicate the name of the extension, following the `grammar` keyword, and also `import` the host language specification since extensions build on top of many and translate to host language constructs.

The syntax that most type qualifiers add is quite simple, typically introducing a new keyword, such as `pure` or `nonnull`. In Silver this is implemented as a new `terminal` symbol, in this case named `Nonnull_t`, with the constant regular expression `'nonnull'` defining its syntax. Since this terminal marks the beginning of an extension's syntax it is a `marking` terminal, described below. This terminal is in the `Ckeyword` lexer class to give it precedence over variable and

```
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(); }

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

function isNonNullQualified
Boolean ::= t::Type { return ... elided ...; }

aspect production dereference
e::Expr ::= d::Expr
{ lerrors <- case isNonNullQualified(d.type) of
   | true -> [ ]
   | false -> [ err(e.location, "dereference on pointer "
                ++ "without 'nonnull' qualifier") ] ;
}
```

Figure 4. Fragments of `nonnull` type qualifier implementation.

type names whose regular expressions overlap with this one; this is just the common notion of lexical precedence seen in most scanner generators.

To add a new qualifier, extension developers write a new production with a host language nonterminal `Qualifier_c` on the left-hand side. We conventionally suffix names of concrete syntax elements with "_c" to distinguish them from their abstract syntax counterparts. The production named `nonnull_c` defines the new `nonnull` qualifier and indicates how to construct the abstract syntax tree for this qualifier, using the `nonnull` abstract production defined below.

While the `nonnull` qualifier is syntactically rather simple, our dimensional analysis qualifier is parameterized by arithmetic expressions over the SI units in Table 1. For example, a floating point variable to hold acceleration values can be declared as

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

in which the derived unit for acceleration is constructed using the division and power operators over the symbols in Table 1. Standard metric prefixes such as `k` for kilo or `m` for milli can precede unit symbols allowing dimensions such as `units(kg*m/s^2)` for newtons. Here the prefix `k` precedes the base unit `g`.

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

An abbreviated specification of the concrete syntax for this qualifier is shown in Figure 5. Note that line comments in Silver begin with `--`. Of interest here is the sub-language for units rooted at the `UnitExpr_c` nonterminal. Terminals for the base unit symbols in Table 1 are declared, though only two are shown here. Next are terminals for the operators; these are given appropriate precedence and associativity settings and used in the concrete productions for `UnitExpr_c` below. The last one derives base units with optional metric prefixes, whose productions and prefix terminals are shown below. Note that the construction of ASTs is elided in all of these productions.

It is worth noting that there is no lexical ambiguity between the base unit (and prefix) terminals and the host language variable and type name terminal symbols even though their regular expressions overlap. Nor are there lexical ambiguities between the new operator terminals here and the arithmetic operator terminals in the host language. This is because Copper generates *context-aware* scanners [29]. When the scanner is called for the token, it uses the current LR parsing state to identity 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 MDA analysis practical.

Also note that we use the same terminal symbol, `Meter_t` (`m`), for the base unit meter and the prefix "milli". Context-aware scanning is not enough to avoid the ambiguity of introducing another terminal matching `'m'` in this context. This type of reuse is common in LALR(1) parser specifications and is 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.

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

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

nonterminals UnitExpr_c, BaseUnit_c, UnitPrefix_c;
terminal Meter_t 'm';
terminal Gram_t  'g';  --other unit terminals 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 = ...; }
   | l::UnitExpr_c '/' r::UnitExpr_c { u.ast = ...; }
   | l::UnitExpr_c '^' i::IntLiteral { u.ast = ...; }
   | '(' e::UnitExpr_c ')'           { u.ast = ...; }
   | p::UnitPrefix_c b::BaseUnit_c   { u.ast = ...; }
concrete productions b::BaseUnit_c
 ::= 'm' { b.ast = ...; }
   | 'g' { b.ast = ...; }  --other unit productions
                           --elided
terminal Kilo_t  'k';
terminal Centi_t 'c';  --other prefix terminals elided

concrete productions p::UnitPrefix_c
 ::= 'k' { p.ast = ...; }
   | 'c' { p.ast = ...; }
   | 'm' { p.ast = ...; }  --Reuse of Meter_t
   |     { p.ast = ...; }  --empty, prefix is optional
```

Figure 5. Dimension analysis qualifier `unit()` and unit expressions.

3.2 Abstract Syntax for Qualifiers

After the parser has completed and the abstract syntax tree (AST) generated by the specifications in the concrete syntax, the AST is decorated with attributes to compute static semantic information such as the types of expressions or the list of errors found on a statement. Silver is a higher-order [31] attribute grammar system [19] with a simple notion of reference [15] / remote [3] attributes used for this process.

Attributes for qualifiers: Returning to the `nonnull` example in Figure 4, note the `abstract production` named `nonnull` that constructs an (abstract) `Qualifier` in the AST. A list of qualifiers of this sort are associated with each `Type` nonterminal in the ableC abstract syntax to represent types. The `Qualifier` nonterminal is decorated with attributes to define the semantics of a qualifier and equations that define these attribute values for `nonnull` can be seen in the figure. Similar productions for the `pure` and `watch` qualifiers are defined in their language extensions.

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

Figure 6. Partial specification of ableC pointer dereference construct. This production is enhanced in Section 6.

Our implementation splits the sign of a qualifier among two Boolean attributes, `isPositive` and `isNegative`, on the `Qualifier` nonterminal. A qualifier that sets only one of these to `true` works as expected and as described in Section 2. Since `nonnull` is a negative qualifier, it sets `isNegative` to `true` and `isPositive` to `false`. It applies at the value level and thus `applyAtValLevel` is set to `true`.

It is possible to set both `isPositive` and `isNegative` to `true` on a qualifier q, thus indicating that $q\ \tau \leq \tau$ and $\tau \leq q\ \tau$, or $\tau = q\ \tau$. This is done for the `watch` qualifier so that no subtyping restrictions arise. Both attributes can also be set `false`; this effectively creates a new type since $q\ \tau \not\leq \tau$ and $\tau \not\leq q\ \tau$, or $\tau \neq q\ \tau$. Since ableC already has extension mechanisms for creating new types we have not yet found a compelling use for this capability.

Type expressions and type checking: As mentioned above, abstract `Type` nonterminals representing types have an attribute named `qualifiers` that is a list of `Qualifier` trees. To determine whether a qualified type $q_1\ \tau$ is a subtype of $q_2\ \tau$, we check 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. Thus, annotating a type with a qualifier that sets both `isPositive` and `isNegative` has no effect on type compatibility. Each `Qualifier` defines a `qualCompat` attribute that is a function that compares itself to another qualifier for this check. For simple qualifiers like `nonnull` this is just a lambda-expression (written `\ v::t -> expr`) that checks for equality. For more sophisticated qualifiers like `unit`, this function is more complex as it must convert the unit expressions, such as `m * m` and `m^2`, into a canonical form to compare them. These `qualCompat` functions are used to detect that, *e.g.*, an `int const *` typed value cannot be passed into a function parameter with type `int *`. All type checking in ableC performs this subset check on qualifiers.

Another error checking feature in Foster-style type qualifiers is the additional checking for qualifiers on specific host language operations, such as requiring dereferenced pointers to be qualified by `nonnull`. Thus, specifications in the `nonnull` grammar in Figure 4 need to add new errors to the host language dereference construct. In ableC, operations

such as this are implemented by abstract productions in the host language; one example is the pointer dereference production in Figure 6. This production uses a few advanced Silver features. The local errors attribute `lerrors` is a list of messages (`[Msg]`). It is also a *collection* attribute, meaning that additional messages can be added remotely (in this case from the `nonnull` specification) and these are all combined with the list append operator `++`. This attribute is initially populated by the errors on the dereferenced expression `d` using the special collection attribute initializer `:=`. If there are no errors then this construct translates to a final, post-processing version of the construct. Otherwise it is translated into an error construct that contains the errors.

This translation is done via *forwarding* [28]. When an AST node is queried for an attribute for which it has no equation, it constructs the tree defined by the `forwards to` clause and automatically queries that tree for the attribute value. This is primarily used by language extensions to define their translation to host language constructs.

For the `nonnull` specification to add new error messages to `lerrors` it uses an `aspect` production, as seen in Figure 4. Such productions can add new attribute defining equations to an existing `abstract` production. In this case, new error messages are contributed to `lerrors` on `dereference` using the contribution operator `<-` if the type of the dereferenced expression `d.type` is not qualified with `nonnull`. If it is so qualified, then the empty list, `[]`, is contributed.

Productions for host language operators in ableC follow the general pattern exemplified by `dereference` and thus aspect productions in language extension grammars can add new errors to existing operators. The host language `const` qualifier works in this way to check assignments and the dimensional analysis extension works similarly to check for compatible `unit` qualifiers on arithmetic operators.

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

Type qualifiers introduced as language extensions must work well with other language extensions and libraries used by the programmer. This raises two challenges whose solutions are discussed in this section. First, any useful system that allows extensions to work together must not generate error messages on code generated by a language extension since the programmer cannot annotate that code with the necessary type qualifiers. 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, library code in header (`.h`) and source (`.c`) files must also be handled properly so that qualifiers can be easily added to library header files without editing them.

4.1 Context-Aware Type Qualifier Analysis

Consider again the extended ableC code in Figure 1. Language extensions in ableC specify their translation to plain C, via forwarding. The `datatype` declaration is translated to a collection of C `struct` and `union` types for representing `Expr` values in the expected, but less convenient, manner. The pattern matching construct introduced by the extension translates into the expected, but again less convenient, nested `if-else` statements.

A potential problem arises in the pointer declarations and their dereferences in the generated C code translation of the extension; the translation of the `match` construct has a local pointer named `_current_scrutinee_ptr` that adds an additional level of pointer indirection to types in the `datatype` declaration. To match a pattern against the value (in this case `*e`) being scrutinized this pointer tracks the elements being inspected and is frequently dereferenced. But the (generated) type of this pointer is not qualified by the `nonnull` extension and thus dereferences to it are erroneous.

Typically, extensions check for errors so that error messages are in terms of the programmer-written code, not the generated C code. But this is not required and for some forms of extensions the errors found on the generated code lead to reasonable error messages. Thus, it would be possible for `nonnull` dereference errors to be reported on extension generated code that the programmer cannot change. Even if the extension does check for errors and disregards any errors on the generated code, this leads to a violation of an unenforced invariant that if no errors are found in error checking the programmer-written code, then there should be no errors on the generated C code. Although this may not affect the programmer, it is still undesirable.

Qualifiers may also not want to generate error messages on code from standard (`.h`) or extended (`.xh`) header files. Locations in Copper and Silver, and thus ableC, include the original filename and this is maintained by the C preprocessor, using # tags.

To solve this problem, type qualifier analysis can examine the context in which the analysis in being applied by inspecting the location of the code. Locations in ableC are structured data created by a `loc` or a `generatedLoc` production. The second indicates that the code, in this case the dereferences generated by the extension, are generated and thus have a location matching the pattern `generatedLoc(_)`. The function `suppressError` in Figure 7 determines, based on the location, if errors should be suppressed or not.

We can rewrite the aspect production for `dereference` from Figure 4 so that error messages are only added to `lerrors` if the type is not appropriately qualified and errors are not to be suppressed. This can be seen in Figure 7.

As we will see in Section 6, the `nonnull` qualifier extension can also introduce a dynamic check for the pointer to ensure that it is not null during pattern matching.

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

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

Figure 7. Context-aware analysis of the nonnull qualifier, a revision on Figure 4.

4.2 Working with Libraries

First, library source files present no problems since they are compiled with a standard C compiler and not ableC. So no new type qualifier analysis need occur there.

Problems may arise when header files are included by the #include directive. This code is processed by an extended ableC compiler and must act accordingly. In some cases, the lack of extension-defined type qualifiers in library functions simply creates a lost opportunity to prevent improper use of such functions. For example, passing a null file pointer to the standard I/O functions—*e.g.* fgets()—results in undefined behavior at runtime, but could be prevented at compile time if the parameters were annotated to be of type FILE * nonnull. More seriously, in other cases the annotation of library functions with extension-defined type qualifiers may be critical to the usefulness of the qualifiers in practice. Consider the tainted qualifier. Failing to annotate as tainted strings that are read from a file—again, *e.g.* with fgets()—would create a gap in the analysis.

Cqual addresses this by allowing user-annotated prototypes to take precedence over those in standard header files, but provides no mechanism for composing multiple annotated prototypes for the same function for different qualifiers. Thus, to support independent development of type qualifiers, we allow functions to be redeclared with differing extension-defined qualifiers as long as the prototypes are otherwise compatible, and accumulate the type qualifiers as they are seen. The type of the function is then formed from the union of all such added type qualifiers.

For example, the tainted qualifier extension may provide a header file containing the prototype

```
tainted char *fgets(tainted char *s, int size,
                    FILE *stream);
```

and the nonnull extension may provide a header containing

```
char *fgets(char *s, int size,
            FILE * nonnull stream);
```

```
grammar edu:umn:cs:melt:ext:ableC:pfp ;
imports edu:umn:cs:melt:ableC;

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

Figure 8. Checking that pure qualifies only function types.

In a program that includes (#include) both of these, ableC will compute the type of fgets to be

```
tainted char *fgets(tainted char *s, int size,
                    FILE * nonnull stream);
```

This prototype will now ensure that potential incorrect uses of fgets described above are caught statically. Importantly, the programmer did not need to write this combined prototype, as in Cqual. It was composed automatically from the extension-written prototypes in their header files.

5 Type-Specific Qualifiers and Type-Independent Qualifiers

This section discusses how type qualifier extensions can check for errors in their application to a type, *e.g.* some qualifiers such as pure and associative are specified to only be allowed to qualify function types. In contrast, some qualifiers, such as units, can be applied to any type; this allows for additional applications of these qualifiers. We show how the dimensional analysis qualifier can be applied to a separately-developed numeric interval type.

5.1 Checking Errors on Type Expressions

Type qualifiers can limit their application to specific types, *e.g.* the pure and associative qualifiers in the parallel functional programming extension can only be applied to functional types. This extension raises an error on declarations such as pure int x; since this declaration violates this extension-specified policy. To accomplish this, qualifiers can define an errors attribute of type [Msg] that decorates many host language nonterminals to be something other than the empty list. For pure in Figure 8 the *inherited* attribute typeToQualify is passed down to q::Qualifier from the enclosing type so that pure can inspect this type. Here it checks if it is a function type matching the pattern functionType(_,_). If not, an error message is generated.

nonnull performs a similar check. Qualifiers can perform other type expression checks by writing more sophisticated attribute equations. For example, a dimensional analysis qualifier units checks that only one units qualifier decorates a type by filtering the list of qualifiers on the type

```
abstract productions addOverload
e::Expr ::= l::Expr r::Expr
{ attr lerrors :: [Msg] with ++ ;   lerrors := [];
  forwards to if null(lerrors)
   then case getAddOverload(l.type, r.type) of
       | just(p) -> p(l, r)
       | nothing() -> add(l, r)
   else errorExpr(lerrors);
}
```

Figure 9. Overloading addition in ableC.

(`q.typeToQualify`) and checking if there are two or more that match the pattern `units(_)`. It does not however check the type that it qualifies and can thus qualify any type, a feature used in combination with operator overloading to qualify numeric types introduced by other extensions.

5.2 Type Qualifiers and Operator Overloading

In ableC, host language operators such as addition (`+`), multiplication (`*`), and many others can be overloaded by new types introduced by language extensions [17, Section 7.1]. For example, an extension-defined interval type can overload addition so that expressions such as

```
interval[1.2, 3.1] + interval[2.0, 4.4]
```

evaluate to the interval value `interval[3.2, 7.5]`.

The new `interval` type can also be qualified by the dimensional analysis `units` qualifier since `units` does not restrict itself to any specific types in the way that `pure` and `nonnull` do. Thus, a sum function over meter-qualified intervals could be written as

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

The concrete syntax of ableC creates the AST for addition using the `addOverload` production, partially shown in Figure 9. It collects local errors (`lerrors`) and if there are any translates to an erroneous `Expr` production that we have seen before. Otherwise it follows a process we elide here to determine whether it should translate (via forwarding) to an extension-defined overloading production (called `p` in Figure 9) or the default addition production (`add`).

The dimensional analysis extension can contribute new errors to the `lerrors` errors attribute in Figure 9 by writing an aspect production similar to the one in Figure 7 for `nonnull` dereferences. This aspect is shown in Figure 10. It calls an elided function `unitsCompatible` to determine if an error should be added to `lerrors`, again using the `<-` contribution operator. It also specifies what type qualifiers should be collected on the resulting type of this addition. In this case it is just the units on one of the arguments, but for multiplication it would be the product of the unit expressions in the `units` qualifiers on the types of `l` and `r`.

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

aspect productions addOverload
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") ];
  e.collectedTypeQualifiers <- if !compat then []
     else [getUnits(l.type)];
}
```

Figure 10. Type-independent dimensional analysis error checking.

```
({ int * _tmp = d;
   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 11. Generated dynamic checks on dereferences.

Error checking for the dimensional analysis extension happens before operator overloading is resolved because the aspect production shown in Figure 10 is on the `addOverload` production. Alternatively, an extension may choose to write a similar aspect on the default addition production `add` instead. 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.

6 Dynamic Type Qualifier Checking

There are occasions when statically detecting qualifier-based errors is not feasible or appropriate. Recall from Section 4.1 where the `nonnull` qualifier checks its context so as to not generate error messages on code generated by other extensions, in that case the `match` construct from the algebraic data type extension. Instead of doing nothing, the `nonnull` extension could insert code to do a run-time check of the pointer to be dereferenced. Another example is a `check_bounds` qualifier, described below, that adds simple bounds checking to arrays with its qualifier.

Dynamic non-null checks: Before discussing the details, consider the sample code in Figure 11 generated from a dereference to an integer expression `d`. This code uses the GCC *statement-expression*, bracketed by `({` and `})` to contain the if-statements that perform the dynamic checks. A temporary variable `_tmp` is used to avoid evaluating `d` more than once and a `done` flag is set to false. Following are two dynamic

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

aspect production dereference
e::Expr ::= d::Expr
{ e.errors <- ... as before ...
  dynamicChecks <- if isNonNullQualified(d.type) ||
                      ! supressError(e.location)
                   then [ ]
                   else [ (checkNull, "null deference") ];
}
function checkNull  Expr ::= tmp::Expr
{ return equalityOp(tmp, intConst(0)); }
```

Figure 12. Adding a dynamic non-null check to pointer dereference. A revision on Figure 7.

checks that examine the contents of `_tmp` and potentially print an error message and set the flag to true. If any set the flag, the code exits, otherwise the value returned for the statement expression is the dereference of `_tmp`. The first is a check if the pointer is null, added by the `nonnull` extension, the second is elided but added by some other extension.

Host language productions, like `dereference`, have a second collection attribute (to go along with `lerrors`) named `dynamicChecks` that extensions can contribute to in order to add dynamic checks like those in Figure 11. This attribute has the type `[( Expr::=Expr, String)]`. The first element is a function for constructing the checking expressions in the `if` condition whose argument is the temporary `_tmp`. The second is the message to print if the check detects an error. Extensions can contribute elements to this list that inject dynamic checks into the generated C code. In Figure 12 the dimensional analysis extension adds a null check to the `dynamicChecks` collection attribute if the type is not qualified by `nonnull` and errors are to be suppressed.

The host `dereference` production in Figure 6 is extended to initialize `dynamicChecks` to the empty list, like `lerrors`. Instead of forwarding directly to `dereferenceFinal` if there are no errors, it first wraps all the dynamic checks specified in `dynamicChecks` by other extension contributions to construct code like that in Figure 11. It uses `dereferenceFinal` for the last dereference so that these checks are only inserted one time since that production does not inject these checks.

In a similar manner, cast operations that add a `nonnull` qualifier to the type can also generate a dynamic check that the pointer is not null.

Dynamic array bounds checking: Attempting to access array elements outside of the actual bounds is a notorious problem in C. A lightweight dynamic solution is provided by a `check_bound` qualifier on pointer types and a custom memory allocation function `malloc_bc` that maintains a map of allocated pointer addresses and the size allocated in a global map. For example, the statement

```
int * check_bounds x = malloc_bc(5 * sizeof(int));
```

```
units(m) double length = 2.3;
units(mm) double width = 450.2;
/* scaling width from mm to m at runtime */
units(m) double perim = 2. * (length + width);
```

Figure 13. Example use of type qualifier code insertion by the dimensional analysis extension.

maps the allocated memory to the allocated size, here 20 bytes assuming 4 byte integers. On an array access, `x[i]`, a dynamic check is inserted on the host language array access production in much the same fashion as on the `dereference` production above. This compares the index value to the allocated size in the global map to check if it is valid. An error is raised if not. This extension injects the check for all arrays qualified by `check_bounds`, even those inside of loops. A more sophisticated version would 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.

7 Type Qualifier Directed Code Insertion

Besides the dynamic checks that were described above, extensions can insert more general forms of code that encloses another expression or statement. An example of this is the simple `watch` qualifier in Figure 2. On assignment statements, the expression on the right-hand side is "wrapped up" in a GCC statement expression that stores the result in a temporary, prints it, and then returns that value.

The `watch` qualifier is not signed—it is neither positive nor negative. Thus no subtype relation is induced and watched values can be passed into functions not so qualified, but the changes to the value are then not printed.

It makes sense for changes to `watch` qualified values to be printed even if the updates are from extension generated code. Thus the `watch` qualifier does not make use of the context-aware mechanism described in Section 4 and inserts the same code on programmer-written and extension-generated code so that all changes can be observed.

Another potential application of code insertion comes from the dimensional analysis qualifier. Consider the example in Figure 13 in which two distance variables, qualified with different units, are added. The `addOverload` production shown in Figure 9 has an additional collection attributes, `lInserts` and `rInserts`, both of type `[Expr::=Expr]`, that contain code-wrapping functions added by different extensions for the left, and respectively, right children of the production. The equations for the right child are shown in Figure 14. The local attribute `rInserted` is the result of folding up elements of `rInserts` around `r`, similarly for the left child. The `forwards to` clause is updated to use `lInserted` and `rInserted` from the original `l` and `r` in Figure 9. To add a multiplication of a scaling factor around the right child of add to match the units on the left child, the dimensional analysis extension can add the following equation to the aspect production in Figure 10:

```
abstract productions addOverload
e::Expr ::= l::Expr r::Expr
{ attr rhsInserts :: [ (Expr ::= Expr) ] with ++;
  rInserts := [];
  attr rInserted :: Expr =
   foldr(\f::(Expr::Expr) e::Expr -> f(e), r, rInserts);
  forwards to if null(lerrors)
   then case getAddOverload(l.type, r.type) of
       | just(p) -> p(lInserted, rInserted)
       | nothing() -> add(lInserted, rInserted)
   else errorExpr(lerrors);
}
```

Figure 14. Overloading addition in ableC with code insertions, a revision on Figure 9.

```
  e.rInsertions <- if !compat then []
                   else [ convertUnits(l.type, r.type) ]
```

Note 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.

These examples illustrate that qualifiers in ableC can extend host language constructs to wrap up expressions with new code that animates changes going on in the program or adds runtime scaling conversions to data.

8 Related Work

8.1 Type Qualifiers

Cqual is a tool that analyzes C programs extended with user-defined type qualifiers in the model of Foster et al. [12, 13]. It has been shown to be useful in detecting many kinds of errors in real-world programs, including locking bugs in the Linux Kernel [13] and format string vulnerabilities [26].

The concrete syntax of C is modified such that user-defined type qualifiers begin with a dollar sign. This makes it easy to identify and remove user-defined type qualifiers after analysis so the code can be passed to a standard compiler. Foster concedes that the code is more readable without the dollar signs and often omits them [11, Section 5.1]. ableC does not place this restriction on qualifier names because the modular determinism analysis and use of transparent prefixes ensures that there are no lexical or syntactic ambiguities in the concrete syntax specification of an extended ableC instance, except for the unlikely few that are are easily resolved by of use transparent prefixes.

As seen previously, user-defined type qualifiers introduce a subtyping relation, but semantics beyond subtyping is limited. It can be specified that the operands of certain operators must be annotated with certain qualifiers—*e.g.* pointers being dereferenced must be qualified as nonnull—but more sophisticated errors found by examining the AST directly are not supported, nor can the content of error messages be specified as in ableC. No method is provided for the user to specify the types that a qualifier is allowed to annotate.

Qualifier polymorphism is supported by naming qualifiers in function prototypes beginning with an underscore. For example, `$_1 int foo($_1 int);` declares a function that takes an argument with some set of qualifiers and returns a value with the same set of qualifiers. ableC can support something similar to this simple example using the templating extension. Cqual 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` make it a subtype of the qualified `dest`, which is identical to the qualified return type.

The type qualifiers as language extensions in ableC differ from those in Cqual in two ways. First, in Cqual there is no consideration made for the composition of independently-developed type qualifiers. While the chances of conflicting qualifier names are small there is no mechanism in Cqual to address it that is similar to the transparent prefixes in ableC (Section 2.4). Additionally, library functions are annotated with qualifiers through the use of prelude files, which are similar to but take precedence over regular `.h` files. For two or more independent qualifiers to qualify the same library function, the programmer using the qualifiers must manually rewrite these function headers to use all relevant qualifiers. In ableC this process is automated by accumulating qualifiers for each header file that is processed (Section 4.2). The second difference is that qualifiers in ableC can inject code to dynamically check for errors or perform other computations such as runtime data conversions (*e.g.* dimensional analysis) or print changes to a variable (*e.g.* the `watch` qualifier).

8.2 Extensible Type Systems

Extensible types allow multiple type systems, including user-defined type systems, to be used in a single language [6]. Like type qualifiers, pluggable types provide a means for finding additional compile-time type errors, and have no effect on the run-time semantics of the program.

The Checker Framework is a framework for implementing pluggable type systems in Java [23]. User-defined checkers are implemented as plug-ins to `javac`. It contains built-in support for subtyping and flow-sensitivity. Examples of checkers implemented in the Checker Framework include a nullness, fake enumeration, and units *i.e.* dimensional analysis checkers among many others [6]. The concrete syntax of type qualifiers make use of Java's annotation syntax. This means that type qualifiers begin with an 'at' symbol, an inconvenience similar to Cqual's requirement that they begin with a dollar symbol. These also support parameters in a manner less expressive than what was demonstrated with our sub-language in `unit` qualifiers for derived units. Each checker selects the annotations of interest and ignores the others. There is no consideration made to handle conflicts

arising from the composition of independently-developed type qualifiers, other than to advise users to be careful to avoid using two checkers that both use the same annotation.

The Checker Framework takes three approaches to annotating libraries with qualifiers: 1) suppressing warnings from unannotated libraries, 2) so-called stub files that annotate method signatures but not bodies, similar to the prelude files of CQUAL, and 3) annotation and recompilation of the library source. As with the annotations of library functions when using CQUAL, this presents difficulty in composing annotated libraries that have been independently developed.

JavaCOP [2, 20] is another Java system, 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. Users of the API have access to the program's AST, which includes information about program structure including the user-defined annotations discussed above. JavaCOP type checking extensions are specified in a declarative domain-specific language as rules which specify the construct in the abstract syntax to check and the constraints applied to it. This is more convenient than the attribute grammar equations in ABLEC but also more limiting.

8.3 Extensible Languages and Language Frameworks

There has been much work in the area of extensible languages and language frameworks. Various systems, such as the SugarJ [9] and MetaBorg [4] frameworks for extending Java, the JastAdd [8] extensible Java Compiler [7], and the Xoc [5] and XTC [14] 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. Other systems provide guarantees of composability but give up some expressibility. For example, mbeddr [30], Wyvern [22], and VerseML [21] do not suffer from challenges in parsing composed languages but they 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 found useful in some language extensions. DeLite [24] 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, but is more limited that what is possible with ABLEC.

9 Discussion and Future Work

This paper presents a reformulation of and extension to previous work on type qualifiers in which new type qualifiers can be packaged as composable language extensions. Programmers, who need no understanding of the underlying implementation mechanisms, can easily import their chosen set of language extensions with the assurance that their composition will be successful.

Some non-extensible languages, such as F# and Osprey [16], which adapts the parser of CQUAL, provide dimensional analysis capabilities built into the language. These demonstrate the desire for such features and the need for language extension capabilities more generally.

Our extensions to previous work provide for more syntactically expressive type qualifiers, as seen in the `units` qualifier for dimensional analysis (Section 2.4). We also provide more semantically expressive qualifiers as in the general ability to read and define attributes and the code insertion techniques in Sections 6 and 7.

There are some capabilities of systems like CQUAL that we have not yet implemented and are left as current and future work. For example, our current implementation has limited support for flow-sensitive type qualifier inference. 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. 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 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. We also lack support for qualifier polymorphism. Our language extension for templates 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 since 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.

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.

References

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

[2] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. 2006. A Framework for Implementing Pluggable Type Systems. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*. ACM, 57–74.

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

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

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

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

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

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

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

[10] David Evans. 1996. Static Detection of Dynamic Memory Errors. In *Proceedings of the ACM 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, 44–53.

[11] Jeffrey Scott Foster. 2002. *Type Qualifiers: Lightweight Specifications to Improve Software Quality.* Ph.D. Dissertation. University of California, Berkeley.

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

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

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

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

[16] Lingxiao Jiang and Zhendong Su. 2006. Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 262–271.

[17] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and automatic composition of language extensions to C. *Proceedings of the ACM, Programming Languages, OOPSLA* 1, 98 (October 2017).

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

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

[20] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. 2010. JavaCOP: Declarative Pluggable Types for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 2, Article 4 (Feb. 2010), 37 pages.

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

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

[23] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, 201–212.

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

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

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

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

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

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

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

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