

# Semantic Type Qualifiers

Brian Chin

Shane Markstrum

Todd Millstein

Computer Science Department  
University of California, Los Angeles

{naerbnc,smarkstr,todd}@cs.ucla.edu

## ABSTRACT

We present a new approach for supporting user-defined type refinements, which augment existing types to statically ensure additional invariants of interest to programmers. We provide an expressive language in which users define explicit type rules for new refinements. These rules are automatically incorporated by our framework’s *extensible typechecker* during static typechecking. Separately, our framework’s *soundness checker* automatically guarantees, once and for all, that a refinement’s type rules ensure the intended invariant, for all possible type-correct programs. We have formalized our approach and have instantiated it as a framework for adding new type qualifiers to C programs. We have used this framework to define and automatically prove sound a host of type qualifiers of different sorts, including `nonnull`, `nonzero`, `untainted`, and `unique`, and we have applied our qualifiers to ensure important invariants on open-source C programs.

## 1. INTRODUCTION

Type systems are a natural discipline for ensuring that a program maintains certain run-time invariants. As a simple example, if an expression can be given the type `int`, the programmer is assured (modulo type-unsafe features like casts) that the expression will only ever evaluate dynamically to an integer value. Recent work in our community has shown how to refine the types in traditional type systems to ensure other important kinds of run-time invariants, including memory safety (e.g., [37, 21]), invariants about pointers and their aliasing relationships (e.g., [7, 5, 1, 14]), and invariants about the interactions of threads in concurrent programs (e.g., [15, 4, 17]). The refinements are achieved by augmenting standard types with annotations that represent the additional properties of interest and augmenting standard type rules to check these annotations statically.

Of course, language designers cannot anticipate all the run-time invariants that programmers will want to specify and check. They also cannot anticipate all of the practical ways in which types may be refined in order to enforce a particular invariant. Therefore, rather than providing a fixed type system, it is desirable to provide a framework for *user-defined type refinements*, whereby program-

mers can easily augment a language’s type system with new type annotations to ensure invariants of interest. For example, Foster *et al.* [18, 19] describe CQUAL, a system that allows programmers to define new type *qualifiers*, such as `nonnull` and `const`, for C programs, and Mandelbaum *et al.* [26] provide a theory of type refinements to specify and check properties, like temporal protocols, that depend on “effective” computations.

Despite their benefits, existing frameworks for user-defined type refinements have important limitations that hinder their utility. First, they use a fixed set of generic type rules across all type refinements. Type-refinement-specific information is instead provided by annotating each program to indicate when the typechecker should assume or check that a refinement holds on some program fragment. While such annotations can simulate a limited form of refinement-specific type rules, they are not expressive enough to handle many common situations. For example, it would be difficult to simulate a type rule for some expression that depends recursively on the types of subexpressions.

Second, while existing frameworks ensure that programs respect user-defined typing disciplines, the type-refinement designer must take responsibility for ensuring that his new typing discipline in fact establishes the desired run-time program invariants. Any errors in the user-defined typing discipline will go undetected by the framework. For example, it is possible for a program to typecheck in CQUAL but nonetheless cause a variable declared `nonnull` to have the value `NULL` at run time. When a language has a fixed type system, it is reasonable for the language designer to prove once and for all, typically by hand, that the type system is *sound*, guaranteeing that such problems cannot arise. However, it is unreasonable to expect programmers to perform such proofs for each new type refinement in an extensible framework.

In this paper, we explore a novel approach to user-defined type refinements that addresses these limitations, marrying the expressiveness and reliability of fixed type systems with the benefits of user-extensible type systems:

- **A language for type-refinement rules.** We provide an explicit language in which users may write type rules for their new refinements. This language enables the natural expression of user-defined typing disciplines, and it is much more expressive than the program annotations used by prior frameworks. An *extensible typechecker* automatically incorporates user-defined type rules during static typechecking, in order to enforce users’ typing disciplines.
- **Semantic guarantees.** We allow users to explicitly specify the run-time program invariant that a type refinement is meant to ensure. We observe that for many interesting cases such invariants are quite natural and simple. A *soundness checker* uses a refinement’s invariant to *automatically* prove

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

that the associated type rules are sound: the rules ensure the specified run-time invariant will hold, for all possible type-correct programs.

We have instantiated our approach as a framework for adding user-defined type qualifiers to C programs. The framework’s extensible typechecker is implemented in the CIL [31] front end for C. The soundness checker implements a proof strategy that requires a handful of proof obligations to be discharged by an automatic theorem prover. Our implementation employs Simplify [12], the automatic theorem prover from the Extended Static Checker for Java (ESC/Java) [16]. We have used our framework to define and automatically prove sound a host of type qualifiers: qualifiers that restrict the value of an expression, including `nonnull`, `nonzero`, `pos`, and `neg`; qualifiers that restrict the flow of values through a program, including `tainted` and `untainted` [35]; and qualifiers that restrict a program’s aliasing relationships, including `unique`, and `unaliaised`. We also have used our qualifiers to statically ensure important program invariants on open-source C programs.

Our approach combines the benefits of two existing kinds of extensible program checkers. On the one hand, systems like CQUAL allow programmers to define their own syntactic abstractions, which are used to check programs automatically but with no semantic guarantees. On the other hand, systems like ESC/Java [16] directly check semantic properties of programs via user-annotated pre- and postconditions, but this requires additional annotations such as loop invariants in order to be sound and fully automatic. Our approach allows users to define their own syntactic abstractions, which can be checked simply and automatically, while also separately proving the soundness of such abstractions automatically.

This paper represents the first step in a broader research agenda to develop expressive, reliable, and extensible program checkers. We plan to apply our approach beyond the checking of low-level invariants like those enforced by `nonnull` and `unique`. For example, we would like to support program checking with semantic guarantees for design-level properties, including temporal protocols [11] and design patterns [20].

In the next section, we overview our framework for user-defined type qualifiers informally via a number of examples. Sections 3 and 4 describe the implementations of our extensible typechecker and soundness checker, respectively. Section 5 summarizes the formal details of our approach. Section 6 describes experience using our framework to check C programs. Section 7 compares with related work, and section 8 concludes and discusses future directions.

## 2. SEMANTIC TYPE QUALIFIERS

Our framework supports two common classes of qualifiers, which are discussed in turn below. *Value* qualifiers, such as `pos` and `nonnull`, pertain only to the value of an expression. *Reference* qualifiers, such as `unique` and `unaliaised`, (additionally) pertain to the address of an l-valuable expression (or *l-value*). Distinguishing between these classes of qualifiers allows us to provide specialized support for each one, making it easier for users to define qualifiers and for our framework to reason about them automatically.

### 2.1 Value Qualifiers

Figure 1 illustrates a definition of the value qualifier `pos` in our framework, which can be used to statically track positive integers. Line 1 of the figure declares `pos` to be a new value qualifier for the type `int`. It also declares a variable `E`, which is used in the rest of the qualifier’s definition. Each variable declaration includes a type and a *classifier*. The classifier `Expr` indicates that during type-

```

1. value qualifier pos(int Expr E)
2.   case E of
3.     decl int Const C:
4.       C, where C > 0
5.   | decl int Expr E1, E2:
6.     E1 * E2, where pos(E1) && pos(E2)
7.   | decl int Expr E1:
8.     -E1, where neg(E1)
9.   | ...
10. invariant value(E) > 0

```

**Figure 1: A type qualifier and associated type rules for positive integers.**

```

int pos gcd(int pos n, int pos m);
int pos lcm(int pos a, int pos b) {
  int pos d = gcd(a, b);
  int pos prod = a * b;
  return (int pos) (prod / d);
}

```

**Figure 2: Example code using the user-defined `pos` type qualifier.**

checking of a C program, `E` will be instantiated with side-effect-free program expressions. The type `int` constrains these expressions to have type `int`. In addition to the classifier `Expr`, we support the classifiers `Const`, `LValue`, and `Var`, which match against program constants, l-values, and variables, respectively. Our implementation performs qualifier checking over programs in CIL’s intermediate language, which cleanly distinguishes expressions, which are side-effect-free, from instructions [31].

Given the declaration in line 1 of figure 1, programmers may now annotate their programs with the `pos` qualifier, as shown by the C code in figure 2. The `lcm` procedure in the figure computes the least-common multiple of two integers. The `pos` qualifier is used to specify that the two arguments should be positive integers and to ensure that the return value is also positive. To handle nested qualifiers unambiguously, we use a postfix notation, whereby a qualifier qualifies the entire type to its left. A type may be annotated with multiple user-defined qualifiers; their order is irrelevant.

#### 2.1.1 Type Rules

Line 1 of figure 1 declares the new `pos` qualifier, but it does not indicate how this qualifier should be used during typechecking. This is the role of the `case` block beginning on line 2, which uses a form of pattern matching to indicate a subset of expressions that can be given the type `int pos`. For example, the clause in lines 3-4 indicates that a positive integer constant may be given the type `int pos`. The clause first declares the variable `C`, which ranges over integer constants from the underlying program, for use in the rest of the clause. It then specifies the pattern `C`, to indicate the syntactic form of the expression. Finally, the predicate `C > 0` further constrains the expression specified in the pattern.

Type rules like the first case clause of figure 1 can be simulated in systems like CQUAL [18] by annotating all positive integers in a program with a `pos` assumption. However, the case clauses in our framework are more general. For example, the clause on lines 5-6 specifies that an expression that is a product of two expressions of type `int pos` can also be given the type `int pos`. This kind of recursive type rule would be quite difficult to manually encode

using `pos` assumptions. The final case clause illustrates that the definition of a qualifier can depend on other qualifiers. That clause specifies that a negation expression can be given type `int pos` if the negated expression can be given type `int neg`, where `neg` is another user-defined qualifier. In fact, qualifier definitions can be mutually recursive. For example, the definition of `neg` (not shown) has rules that refer to `pos`.

The syntax for expression patterns in case clauses is defined by the following grammar:

$$P ::= X \mid *X \mid \&X \mid \text{new} \mid \text{uop } X \mid X \text{ bop } X$$

Here  $X$  ranges over *variable patterns*, which have an associated classifier upon declaration (e.g., `Expr` or `Const`) that restricts the kinds of program fragments that may match. The pattern `new` matches against calls to memory allocation routines like `malloc`<sup>1</sup>. Various unary and binary operations may also be matched against; for simplicity we restrict their argument patterns to be variable patterns.

The predicate after (the optional) `where` in a case clause may include boolean operations on program constants and variable patterns with classifier `Const`, recursive qualifier typechecks on subexpressions, and conjunctions and disjunctions of these kinds of predicates.

Each clause of a case block can be viewed as an *introduction* type rule for the new qualified type. For example, the second clause is akin to the following rule:

$$\frac{\Gamma \vdash E_1 : \text{int pos} \quad \Gamma \vdash E_2 : \text{int pos}}{\Gamma \vdash E_1 * E_2 : \text{int pos}}$$

As discussed in section 5, we have formalized this semantics of the case clauses.

The extensible typechecker uses the stylized type rules defined by the case block, along with a set of standard rules for typechecking constructs like variable references, procedure calls, and assignments, to perform qualifier checking. For example, consider again the `lcm` procedure in figure 2. As usual, typechecking an assignment statement involves obtaining the types of each side and checking that they match. The assignment to `d` typechecks successfully because both sides of the assignment have type `int pos`: the right-hand side is shown to have this type by the standard type rule for procedure calls, given the declared type signature of `gcd`. The assignment to `prod` also typechecks successfully, because the case clause on lines 5-6 allows `a * b` to be given the type `int pos`. Our rules for `pos` are not able to determine that `prod / d` has type `int pos`, so the programmer must insert a cast to satisfy the typechecker, because of the declared return type of `lcm`.

The case clause allows an expression to be given a qualified type. However, programmers may also want to use qualifiers to statically rule out certain kinds of run-time errors, by restricting the ways in which operations may be used. This is accomplished via a `restrict` clause, as shown in the definition for a `nonzero` qualifier in figure 3. The syntax of a `restrict` clause is identical to that of a case clause. A `restrict` clause specifies that any expression in a given program that matches the clause’s pattern must also satisfy the clause’s predicate. The `restrict` clause for `nonzero` indicates that all division operations in the program must have a denominator of type `int nonzero`. In this way, division-by-zero errors are detected statically instead of dynamically.

<sup>1</sup>Procedure calls, including `malloc`, are not in general side-effect-free and so are not considered expressions by CIL. However, it is critical for common qualifiers like `nonnull` that we allow `malloc` to be matched against in qualifier definitions.

```
value qualifier nonzero(int Expr E)
case E of
  decl int Const C:
    C, where C != 0
  | decl int Expr E1:
    E1, where pos(E1)
  | decl int Expr E1, E2:
    E1 * E2, where nonzero(E1) && nonzero(E2)
  | ...
restrict decl int Expr E1, E2:
  E1 / E2, where nonzero(E2)
invariant value(E) != 0
```

**Figure 3: A type qualifier and associated type rules for nonzero integers.**

For example, consider again the `lcm` procedure in figure 2. If the extensible typechecker is given the definition of `nonzero` in addition to `pos`, it will use the `restrict` clause in `nonzero`’s definition to check the division in `lcm`. The check succeeds because the denominator `d` can be shown to have type `int nonzero`: it is declared to have type `int pos`, so it also has type `int nonzero` by the second clause in the case block of figure 3.

The `restrict` clause plays a role analogous to qualifier assertions in CQUAL [18]. For example, the `restrict` type rule in figure 3 could be simulated by annotating the denominator in each division in a program with a `nonzero` assertion. However, the `restrict` clause is more general, since its predicate may be more complicated than a single qualifier check.

### 2.1.2 Subtyping

It is natural to consider `int pos` to be a subtype of `int`. Subtyping provides more flexibility, for example allowing the following code to typecheck:

```
int pos x = 3;
int y = x;
```

Our extensible typechecker considers *all* value-qualified types to be subtypes of their associated unqualified types. More precisely, if  $q$  is a value qualifier and  $\tau$  is a (possibly qualified) type, then  $q\tau$  is considered to be a subtype of  $\tau$ . The rest of the subtyping rules are standard, for example reflexivity and transitivity. As usual, care must be taken in the presence of pointers [18]. For example, it would be unsound to consider `int pos*` to be a subtype of `int*`, because that would allow the following code, which stores a negative number in a variable of type `int pos`, to typecheck:

```
int pos x = 3;
int* p = &x;
*p = -1;
```

Our language for defining qualifiers does not support explicit subtype declarations between two user-defined qualifiers. However, such subtype relationships can be encoded using the case block. For example, the second clause in the case block of `nonzero`’s definition in figure 3 effectively declares `pos` to be a subtype of `nonzero`: any expression of type `int pos` may also be considered to have type `int nonzero`.

### 2.1.3 Soundness

A user-defined qualifier and associated type rules constitute a typing discipline, which is enforced by our extensible typechecker.

Often such typing disciplines are intended to ensure a particular run-time invariant. For example, the typing discipline defined by the `pos` qualifier and associated type rules in figure 1 is intended to statically guarantee that certain expressions only evaluate to positive integers at run time.

However, the extensible typechecker enforces user-defined typing disciplines in a purely syntactic manner, as usual for typechecking, without regard to their associated invariants. For example, suppose the pattern in the second case clause in the definition of `pos` in figure 1 were erroneously specified as `E1 - E2` instead of `E1 * E2`. In that case, our typechecker would happily use this revised type rule to check programs, even though this can cause `pos`'s intended invariant to be violated at run time.

Rather than forcing users to take responsibility for the correctness of their qualifiers, our framework supports *automated soundness checking*. A qualifier definition may optionally specify the qualifier's associated invariant. The framework then proves, independent of any particular program to be typechecked, that the qualifier's type rules respect this invariant.

For example, consider the definition of `pos` in figure 1. Line 10 uses the `invariant` clause to provide the qualifier's associated run-time invariant. The invariant is a predicate that is implicitly universally quantified over an arbitrary run-time execution state. Let us denote this execution state by  $\rho$ . The `value` predicate is provided by our framework and represents the value of a given expression in  $\rho$ . Therefore, the invariant for `pos` indicates that the value of an expression of type `int` `pos` should be greater than zero, in any run-time execution state.

Given this invariant, our soundness checker generates one proof obligation for each case clause, which is discharged by an automatic theorem prover. The obligation simply requires that if an expression matches the clause's syntactic pattern and satisfies the clause's predicate, interpreted in the context of an arbitrary run-time execution state  $\rho$ , then the associated qualifier's invariant also holds in  $\rho$ . For example, consider the first case clause for `pos` in figure 1. The soundness checker generates the following proof obligation: if an expression `E` is an integer constant that is greater than zero, then the value of `E` in an arbitrary execution state  $\rho$  is greater than zero. This obligation is easily proven, given the evaluation semantics of integer constants.

Now consider the second case clause. The soundness checker generates the following proof obligation: if an expression `E` has the form `E1 * E2` and both `E1` and `E2` satisfy `pos`'s invariant in an arbitrary execution state  $\rho$ , then `E` also satisfies `pos`'s invariant in  $\rho$ . This obligation is easily proven by the semantics of multiplication. On the other hand, if the pattern were erroneously specified as `E1 - E2`, the soundness checker would catch the error and warn the programmer, since the associated proof obligation would fail: it is not possible to prove that the difference of two arbitrary positive integers is also positive. The `restrict` clause does not affect whether or not a qualified expression satisfies its qualifier's invariant, so it is ignored by the soundness checker.<sup>2</sup>

### 2.1.4 Flow Qualifiers

Some common kinds of qualifiers are used solely to restrict the flow of values in a program. For example, a *taintedness* analysis uses qualifiers `untainted` and `tainted` to respectively tag data coming from trustworthy and potentially untrustworthy

<sup>2</sup>It is possible to allow users to explicitly define a new set of "type errors," such as division-by-zero errors, and then have the soundness checker prove that a `restrict` clause in fact rules out such errors. However, these proofs would be fairly trivial and would not be especially useful to programmers.

```
value qualifier untainted(T Expr E)
value qualifier tainted(T Expr E)
case E of E
```

Figure 4: Type qualifiers for checking taintedness.

sources [35]. For soundness, the only requirement is that tainted data never flow where untainted data is expected. Taintedness qualifiers can be used, for example, to detect format-string vulnerabilities, by requiring the format-string argument to `printf` (and related procedures) to be `untainted`. Other common flow qualifiers include `user` and `kernel` to prevent user pointers from being dereferenced in kernel space [22], `low` and `high` qualifiers to ensure secure information flow [38], and `static` and `dynamic` qualifiers for binding-time analysis [23].

Flow qualifiers are a degenerate form of value qualifier in our framework. Since all value-qualified types are considered subtypes of their associated unqualified types, proper value flow is guaranteed "for free." Qualifiers that require no additional properties need not specify any case clauses or any run-time invariant.

For example, figure 4 specifies a taintedness analysis in our framework. The `untainted` qualifier can qualify any type `T`. Since the qualifier has no case block, the only way to introduce an expression of type `untainted T` is by an explicit cast, which denotes that the expression is considered trustworthy. For example, if `buf` is an arbitrary buffer of type `char*` and the format-string argument to `printf` has type `char* untainted`, then the following code typechecks:

```
char* untainted fmt = (char* untainted) "%s";
printf(fmt, buf);
```

However, the invocation `printf(buf)` fails to typecheck, since `buf` is not known to be `untainted`. Indeed, if `buf` contains format specifiers, this call to `printf` will attempt to read nonexistent arguments off the stack.

The `tainted` qualifier is not strictly necessary: because `T untainted` is a subtype of `T` but not vice versa, any expression whose type is not qualified with `untainted` is implicitly considered to be possibly tainted. However, it may be useful to explicitly annotate some variables as `tainted`. The definition of `tainted` in figure 4 has the desired behavior. The lone case clause allows any expression to be considered `tainted`, effectively making `T tainted` a supertype of `T` (and hence also of `T untainted`). Because of the implicit subtyping relation for value qualifiers, it is also the case that `T tainted` is a subtype of `T`, so those types are essentially equivalent.

Although the versions of `tainted` and `untainted` in figure 4 are degenerate, they could easily be augmented. For example, a user could decide that all constants should be trusted, adding a case clause to the definition of `untainted` as follows:

```
case E of decl T Const C: C
```

This rule would, for example, obviate the need for the cast in the assignment to `fmt` in our earlier code snippet.

## 2.2 Reference Qualifiers

Figure 5 defines a reference qualifier `unique`, which intuitively specifies that an l-value either contains `NULL` or contains the only reference to a memory location. Because reference qualifiers may pertain to an expression's address, we require reference qualifiers to be defined only for l-values or variables, rather than for arbitrary expressions.

```

ref qualifier unique(T* LValue L)
  assign L
    NULL
  | new
disallow L
invariant value(L) == NULL ||
  (isHeapLoc(value(L)) &&
   forall T** P: *P = value(L) => P = location(L))

```

**Figure 5: A type qualifier for unique pointers.**

```

int* unique array;
void make_array(int n) {
  array = (int*)malloc(sizeof(int) * n);
  for(int i = 0; i < n; i++)
    array[i] = i;
}

```

**Figure 6: Code that uses the `unique` qualifier.**

### 2.2.1 Type Rules

The `assign` block is analogous to the `case` block of value-qualifier definitions. The `assign` block allows users to specify the allowable right-hand-side expressions in assignments to a qualified l-value. These type rules are also used to typecheck implicit assignments to qualified l-values, through procedure calls and returns.

The first `assign` clause for `unique` in figure 5 specifies that a unique l-value may be assigned the value `NULL`. The second clause similarly specifies that a unique l-value may be assigned the result of memory allocation. Consider the code in figure 6. The assignment to `array` in `make_array` typechecks by the second `assign` clause, since C’s `malloc` function matches the pattern `new`.

Because reference qualifiers qualify an l-value’s address, they do not make sense in the context solely of an l-value’s contents. Therefore, top-level reference qualifiers in an l-value’s type are not considered to be part of the l-value’s *r-type*, which is the type used by the typechecker when the l-value appears on the right-hand side of an assignment (or in other expressions like conditionals). However, it may still be necessary to restrict the ways in which an l-value’s contents may be used, to ensure soundness. For example, without any restrictions, the following code snippet would typecheck:

```

int* unique p = ...;
int* q = p;

```

The second statement is allowed because the r-type of `p` is simply `int*`, but it causes `p` and `q` to point to the same location, violating `p`’s uniqueness.

The `disallow` clause in a reference qualifier’s definition addresses this problem by restricting how a qualified reference may be used on the right-hand side of an assignment (or an implicit assignment via procedure calls and returns). A `disallow` may prevent a qualified reference from being referred to and/or having its address taken. The `disallow` clause for `unique` in figure 5 prevents the associated l-value from being referred to, so the assignment to `q` in our code snippet above now fails to typecheck, as desired. For the purposes of `disallow` checking, we treat dereference expressions as atomic. Therefore, a unique l-value may still be dereferenced, so the following code typechecks and is perfectly safe:

```

int* unique p = ...;
int i = *p;

```

We are actively exploring extensions to our framework that allow `unique` to be more expressive. For example, intuitively we can assign a unique l-value any expression that is *fresh*: its value is a memory location that is not currently referenced. In turn, a unique local variable returned from a procedure may be considered *fresh*. We have formalized and automatically proven the soundness obligations for these rules, but we must relax our notion of value qualifiers in order to allow a *fresh* qualifier to be properly expressed in our framework. We are also considering allowing qualifier definitions to directly refer to liveness information. This would allow, for example, a variable to safely be considered unique as long as all aliases are dead [5].

### 2.2.2 Subtyping

Unlike for value qualifiers, there is in general no sound sub- or supertype relationship between an arbitrary reference-qualified type and its associated unqualified type. Therefore, we assume no relationship between these two types. However, the implicit stripping of reference qualifiers from the r-type of an l-value, as described earlier, allows expressions of these types to interact in useful (and still sound) ways.

### 2.2.3 Soundness

As with value qualifiers, our framework automatically proves the soundness of reference qualifiers. Consider the invariant clause for `unique` in figure 5. Similar to the built-in `value` function, the `location` function returns the address of an l-value in a given execution state. The built-in predicate `isHeapLoc` indicates that a memory location is on the heap (i.e., it was dynamically allocated) rather than the stack. The invariant uses explicit universal quantification over all memory locations `P` of the appropriate type in the execution state `ρ`, and `*P` denotes the contents of location `P` in `ρ`.

The `assign` type rules are proven sound in a manner analogous with how the `case` rules are proven sound for value qualifiers. Each `assign` clause for a qualifier `q` has an associated proof obligation, which is discharged by an automatic theorem prover. The obligation ensures that assigning some l-value `l` an expression matching the `assign` clause establishes `q`’s associated invariant for `l`. For example, consider the first `assign` clause for `unique` in figure 5. The soundness checker must prove that if some l-value `l` is assigned `NULL`, then `unique`’s invariant will hold for `l`. This follows easily by the first disjunct in the invariant. The obligation for the second `assign` clause is proven by the semantics for the `new` construct, since `new` always evaluates to an unreferenced memory location.

The proof obligations for `assign` clauses ensure that a reference qualifier’s invariant is properly *established*. To complete soundness checking, we must show that the invariant is also properly *preserved* across assignments. Accordingly, we generate the following proof obligation: if `l` is an l-value satisfying the invariant of some reference qualifier `q` and we execute an arbitrary assignment to some other l-value `l'`, then `l` will still satisfy the qualifier’s invariant in the resulting execution state. We require the arbitrary assignment considered in the obligation to obey the `disallow` block for `q`, if any is specified.

The automatic theorem prover proves this obligation via a case analysis on the different forms of right-hand sides consistent with a qualifier’s `disallow` clause. If the `disallow` clause for `unique` were erroneously omitted, one case in proving `unique`’s preservation obligation would require showing that if `l` is unique and we store the value of `l` in `l'`, then `l` is still unique. Because this case is not provable, the soundness checker would correctly inform the user of the potential unsoundness.

### 3. EXTENSIBLE TYPECHECKING

Our extensible typechecker takes a C program and a set of qualifier definitions in the language described in the previous section. The extensible typechecker then performs qualifier checking on the program as directed by the qualifier definitions' type rules. A qualifier's declared invariant is completely ignored; it is used only by the soundness checker (described in the next section). Our extensible typechecker is implemented as a module in CIL [31], a front end for C written in OCaml [34]. CIL parses C code into a simple abstract syntax tree (AST) format and provides a framework for performing passes over this AST. After qualifier checking, the AST is written back to C, and the gcc compiler performs ordinary C typechecking and code generation.

In this section, we discuss the implementation of our extensible typechecker. First we describe how C programs are annotated with user-defined qualifiers. Then we illustrate how user-defined type rules are represented and used in our CIL implementation. Finally, we discuss some details of handling C programs.

#### 3.1 Annotating Programs

To annotate a C program with qualifiers, we take advantage of gcc *attributes*, which are tags that can be associated with types (and other things like variable names). CIL supports such attributes and maintains them in the generated AST for a program. A type attribute follows the type name and has the following syntax:

```
__attribute__((attribute name))
```

We typically use macros instead of writing the unwieldy attribute syntax directly. Such macros are used in our examples of section 2. For example, the qualifier `pos` mentioned in figure 2 is defined as follows:

```
#define pos __attribute__((pos))
```

#### 3.2 Qualifier Checking with CIL

Our qualifier checker traverses the given CIL AST, performing user-defined type rules on applicable program fragments. Any type errors found during qualifier checking are provided to the programmer as warnings, but compilation is still allowed to continue.

To implement qualifier checking, we have created OCaml datatypes to represent the expression patterns and predicates that are allowed in user-defined type rules. For example, consider the case clause on lines 5-6 in figure 1. The expression pattern is represented internally as follows:

```
Binop(Mult, WCEXPR("E1"), WCEXPR("E2"))
```

The constructor `WCEXPR` represents "wildcard" expressions, which can match any program expression. The clause's predicate is similarly represented as follows:

```
Binop(And, Qual("nonzero", PatVar("E1")),  
      Qual("nonzero", PatVar("E2")))
```

Consider the application of this type rule to the right-hand side of the assignment to `prod` in figure 2. First we match our expression pattern against the CIL AST for `a * b`. The match succeeds and produces bindings for variables in the pattern: `E1` is bound to the expression `a` and `E2` is bound to the expression `b`. Finally, the rule's predicate is evaluated, after replacing each pattern variable with the C program fragment to which it is bound. In our example, the predicate is satisfied if `a` and `b` can recursively be given the qualifier `nonzero`. The other kinds of type rules are represented and checked similarly.

### 3.3 Interacting with C

We allow types to be annotated with qualifiers wherever they appear. For example, the types of `struct` fields may be qualified, and our qualifier checker will check that they obey the user-defined type rules. Fields of unions may also be given qualified types, but the usual unsoundness for unions makes our qualifier checking in this case unsound as well.

As is often the case for C program analyses, we assume a logical model of memory. In particular, we assume that the type of `p+i`, where `p` is a pointer and `i` is an integer, is the same as the type of `p`. This assumption is unsound, but in practice it removes a large source of spurious type errors, particularly because CIL represents array indexing with pointer arithmetic.

Another large source of spurious type errors arises from invoking procedures in the C standard library, since their argument and result types are not annotated with user-defined qualifiers. We currently solve this problem by writing header files that contain alternate signatures for library procedures, which replace the procedures' ordinary signatures via gcc command-line macros. We plan to develop a standardized mechanism for incorporating qualifier annotations on the standard library via user-provided specification files.

Macros from the standard library are also problematic. When these macros are expanded by the C preprocessor, our qualifier checker produces type errors because the macros' bodies are not properly annotated. Short of modifying the macros in place, we have little recourse.

Finally, our qualifier checker can be unsound because it, like C, allows variables to be used before being initialized.

## 4. AUTOMATED SOUNDNESS CHECKING

Our soundness checker takes a qualifier definition and generates the necessary proof obligations for each user-defined type rule, as described in section 2. These obligations are proven by Simplify [12], a Nelson-Oppen-style automatic theorem prover [33]. Simplify contains decision procedures for several decidable theories, including linear arithmetic, equality with uninterpreted function symbols, and maps. Simplify's input language accepts first-order formulas over these theories.

This section details the implementation of our soundness checker. First we describe the axioms we provide Simplify so it can reason about C programs, and then we describe the proof obligations that are proven in the context of these axioms. We have used our soundness checker to automatically prove the soundness of a variety of type qualifiers. The value qualifiers `nonnull`, `nonzero`, `pos`, and `neg` are each proven sound by our checker in under one second. The reference qualifiers `unique` and `unaliaised` are each proven sound in under 30 seconds.

### 4.1 Axioms

We use axioms to formalize the dynamic semantics of programs in CIL's intermediate language. The state of a program is represented by an execution state  $\rho = (\pi, I, \Gamma, \sigma)$ , where  $\pi$  is a program,  $I$  is an index pointing to the statement about to be executed,  $\Gamma$  is the environment, which maps variable names to locations in the store, and  $\sigma$  is the store, which maps locations to values.

We use several function symbols for constructing and manipulating execution states. The *state* symbol takes a program, index, environment, and store, and it constructs an execution state. Accessors like *getProgram* and *getIndex* take a state and return the appropriate component. Environments and stores are represented as *maps*: Simplify has built-in function symbols *select* and

*update* to access elements and functionally update a map. Program expressions and statements also have associated function symbols. For instance, the statement  $*x := \&y$  is encoded as  $assign(deref(var(x)), ref(var(y)))$ .

Given this representation, we define axioms for a function symbol  $evalExpr$ , which evaluates an expression in a given state. For instance, the following axiom formalizes evaluation of variable references<sup>3</sup>:

$$\forall \rho, e, x. (e = var(x)) \Rightarrow evalExpr(\rho, e) = select(getStore(\rho), select(getEnv(\rho), x))$$

We similarly define axioms for a function  $location$ , which takes an l-value and returns its address, and a function  $stepState$ , which takes a program state and returns the state resulting from executing the current statement.

Our axioms only formalize the subset of the CIL intermediate language necessary for reasoning about expression patterns. For example, we do not currently axiomatize the semantics of procedure calls, since they cannot be pattern-matched against, except as black-box expressions which are known to satisfy particular qualifiers. We do, however, explicitly model memory allocation, via a  $new$  function symbol. Omitting the semantics of procedure calls in our axioms is sound assuming that all procedures meet their declared type signatures, which is ensured by the extensible type-checker.

## 4.2 Proof Obligations

To produce a qualifier’s proof obligations, first we define a predicate symbol to represent the qualifier’s invariant. Built-in function symbols like `value` in qualifier definitions are translated to their counterpart function symbols in the axioms. For example, the invariant for `pos` from figure 1 is defined as follows:

$$pos(\rho, e) = (evalExpr(\rho, e) > 0)$$

Proving the soundness of a qualifier  $q$  also requires access to the invariants of all qualifiers  $q'$  that are referred to in  $q$ ’s type rules.

Given these invariants it is straightforward to represent our proof obligations in Simplify. For example, the obligation for the second case clause of `pos` in figure 1 is defined as follows:

$$\forall \rho, e_1, e_2. (pos(\rho, e_1) \wedge pos(\rho, e_2)) \Rightarrow pos(\rho, multExpr(e_1, e_2))$$

As another example, the obligation for the second `assign` clause of `unique` in figure 5 is defined as follows:

$$\forall \rho, l. (getStmt(\rho) = assign(l, new)) \Rightarrow unique(stepState(\rho), l)$$

## 5. FORMALIZATION

We have formalized our extensible type system and soundness checker in the context of a simply-typed lambda calculus augmented with ML-style references [28] and user-defined type qualifiers. Details are in our companion technical report [6].

The evaluation rules are entirely standard and are formalized via a big-step operational semantics. A machine configuration has the form  $\langle \sigma, e \rangle$ , where  $\sigma$  is a store and  $e$  is an expression, and we define an evaluation relation  $\langle \sigma, e \rangle \rightarrow \langle \sigma', v \rangle$ , where  $v$  is a value.

The static semantics is defined, as usual, by a judgment of the form  $\Gamma \vdash e : \tau$ , where  $\Gamma$  is a type environment mapping variable names to types and  $\tau$  is a possibly-qualified type. The inference rules defining this judgment are the standard ones, augmented with

<sup>3</sup>Throughout this section, we elide portions of axioms and proof obligations that involve “typing predicates,” which are used to restrict the domain of function symbols.

```
value qualifier nonnull(T* Expr E)
  case E of
    decl T LValue L:
      &L
    | new
  restrict decl T* Expr E:
    *E, where nonnull(E)
  invariant value(E) != NULL
```

Figure 7: The nonnull value qualifier.

the subtyping relation described in section 2 and a few “hooks” to incorporate user-defined type rules. We have defined a template inference rule to represent each form of user-defined type rule (e.g., `case`, `assign`). We have also formally defined the associated proof obligation for each kind of type rule, and we assume that any user-defined type rules have been shown to satisfy their corresponding obligation (as is ensured by the soundness checker).

To formalize the notion of type soundness, we define a judgment  $\Gamma; \tau \vdash \langle \sigma, v \rangle$ . Intuitively,  $\Gamma; \tau \vdash \langle \sigma, v \rangle$  holds if  $\Gamma \vdash v : \tau$  and  $v$  additionally satisfies all of the associated invariants for qualifiers in  $\tau$ . For example, if  $\tau$  has a top-level qualifier  $q$ , then it must be the case that  $[[q]](\sigma, v)$  is true, where  $[[q]]$  denotes  $q$ ’s invariant predicate. The inference rules for this judgment also recursively check conformance to nested qualifiers.

Next we define a notion of well-formedness of a store with respect to a type environment. We say that  $\Gamma \sim \sigma$  if  $\Gamma$  and  $\sigma$  have the same domain<sup>4</sup> and for each location  $l$  in this domain,  $\Gamma; \Gamma(l) \vdash \langle \sigma, l \rangle$  holds. In other words, every location in the store is well typed and satisfies its qualifiers’ invariants.

Finally we can state the soundness theorem, which is a “semantic” version of the traditional type preservation theorem [39]:

**Theorem:** If  $\Gamma \sim \sigma$  and  $\Gamma \vdash e : \tau$  and  $\langle \sigma, e \rangle \rightarrow \langle \sigma', v \rangle$ , then there exists some  $\Gamma' \supseteq \Gamma$  such that  $\Gamma' \sim \sigma'$  and  $\Gamma'; \tau \vdash \langle \sigma', v \rangle$ .

We have proven this theorem for the case when qualifiers are restricted to be value qualifiers. The proof for the full language with reference qualifiers is in progress.

## 6. EXPERIENCE

This section reports on experience using our framework for user-defined type qualifiers. We describe its usage on existing C programs to statically detect NULL dereferences, violations of uniqueness invariants, and improper format strings. In all of the experiments described below, the extra compile time for performing qualifier checking is under one second.

### 6.1 Null Dereferences

Figure 7 shows the definition of a nonnull value qualifier, which is automatically proven sound by our soundness checker. The `case` rules indicate that the address-of and memory-allocation expressions can be considered nonnull<sup>5</sup>. The `restrict` clause requires all dereferences in a program to be of nonnull expressions.

We used this nonnull qualifier to statically ensure the absence of NULL dereferences in the `grep` search utility program (version 2.5). We annotated the files `dfa.c` and `dfa.h`, which comprise the core string-matching algorithm and related data structures. The files consist of 2287 non-blank, non-comment lines of code.

<sup>4</sup>We treat locations as if they are variables in the static semantics, as others have done [18].

<sup>5</sup>Our new construct is an idealization of `malloc` which never fails.

**Table 1: Results from the nonnull experiment.**

program:	grep
files:	dfa.c, dfa.h
lines:	2287
dereferences:	1072
annotations:	120
casts:	47
errors:	0

We applied `nonnull` annotations to variables in an iterative fashion. Running our extensible typechecker with `nonnull` on the unannotated files produced error messages for each dereference, due to the qualifier’s `restrict` clause. These errors were removed by annotating some variables with `nonnull`, which could in turn cause error messages on assignments to the newly-annotated variables, leading to more annotations. In addition to formal parameters and local variables, we documented several fields of structures as being `nonnull` through this process.

There were situations where the type rules for `nonnull` were insufficient and we were forced to insert casts. The major source of such imprecision is due to the flow-insensitivity of our type system. A simple example from `grep` follows:

```
if ((t = d->trans[works]) != NULL) {
    works = t[*p];
    ...
}
```

The index into array `t` is safe because it is guarded by the check for `NULL`, but our type system cannot deduce this fact. We plan to extend our typechecking algorithm to incorporate flow-sensitivity, borrowing ideas from `CQUAL` [19].

A related source of imprecision occurs when access to a `NULL`-terminated array is guarded by a test against an integer variable representing the array’s length. Statically deducing the invariant between the array and the integer variable may be difficult. One possibility would be to piggyback our qualifier checker on top of `CCured` [32], which (among other things) can sometimes statically deduce array bounds.

Table 1 summarizes the results of our experiment. We were able to validate the safety of all 1072 dereferences in the files. Eliminating all of our extensible typechecker’s error messages required 120 `nonnull` annotations and 47 `nonnull` casts.

## 6.2 Uniqueness

The implementation of `grep` makes use of several global data structures, which are manipulated by a variety of procedures. It would be nice to statically ensure that each global variable is the sole reference to the data structure to which it points. In this way, we can guarantee that each global data structure cannot be updated in unexpected ways through other pointers.

We annotated several global variables in `dfa.c` with the `unique` qualifier, using the definition from figure 5. We had the most success with the global variable `dfa`, which contains the current deterministic finite-state automaton (DFA) being constructed. Our assign rules were not sufficient to statically validate the initialization of `dfa`, since it is initialized to a pointer passed in from the parser module. However, the extensible typechecker validated all 49 subsequent references to `dfa` as preserving the variable’s uniqueness. A representative procedure manipulating `dfa` is shown in figure 8, along with our annotated declaration of `dfa`.

```
static struct dfa * nonnull unique dfa;

static int charclass_index (charclass s) {
    int i;
    for (i = 0; i < dfa->cindex; ++i)
        if (equal(s, dfa->charclasses[i]))
            return i;
    REALLOC_IF_NECESSARY(dfa->charclasses, charclass,
                        dfa->cindex);
    ++dfa->cindex;
    copyset(s, dfa->charclasses[i]);
    return i;
}
```

**Figure 8: A use of unique in grep.****Table 2: Results from the untainted experiment.**

program:	bftpd	mingetty	identd
lines:	750	293	228
printf calls:	134	23	21
annotations:	2	1	0
casts:	0	0	0
errors:	1	0	0

Other global variables were not able to be proven unique using our qualifier, because the variables were passed as arguments to procedures. This idiom violates the `disallow` clause for `unique` in figure 5. Indeed, such uses of the global variables are violations of uniqueness: inside a procedure where a global is passed, the global is no longer unique. It is possible that we could statically check this idiom by relaxing our `unique` qualifier to support “lent” references [1], which allow a `unique` reference to be temporarily aliased.

## 6.3 Untainted Format Strings

Our final experiment used the `untainted` qualifier to ensure proper format-string arguments to `printf`. We used the simple version of `untainted` defined in figure 4, augmented with a case clause that defines all constants to be `untainted`:

```
case E of decl T Const C: C
```

We used this qualifier to annotate and check three of the programs tested by Shankar *et al.* [35], who performed a taintedness analysis using `CQUAL`. The programs are `bftpd` (version 1.0.11), an FTP server; `mingetty` (version 0.9.4), a remote terminal utility; and `identd` (version 1.0), a network identification service. For all three programs, we were able to reproduce the results of Shankar *et al.*

Our results are shown in table 2. Running our qualifier checker on `bftpd` indicated two procedure parameters that are necessary to annotate as `untainted`, since they are used as format strings to `printf`. Re-running the qualifier checker then revealed an exploitable error that had been previously identified [3, 35]. The offending code is shown below:

```
int sendstrf(int s, char * untainted format, ...);
...
sendstrf(s, entry->d_name);
```

The `d_name` field of `entry` is a file name and should not be considered a proper format string. The extensible typechecker ap-

appropriately signals an error since the field has not been declared tainted.

The other two test programs were verified to have no formatting vulnerabilities. In addition, no casts were required for any of the three test programs; the simple case clause defined above was sufficient to infer the taintedness of all format-string arguments.

## 7. RELATED WORK

Our framework is inspired by the CQUAL system of Foster *et al.* [18]. As mentioned in the introduction, our language for user-defined type rules is novel; a weak form of type rules is simulated in CQUAL via qualifier assertions and assumptions. CQUAL also does not support automated soundness checking to ensure that a qualifier establishes its intended invariant. Like our system, CQUAL distinguishes between value and reference qualifiers. CQUAL also supports subtyping relationships among qualifiers. These relationships are potentially more general than the ones allowed in our framework. However, subtyping in CQUAL is declared by the programmer and trusted to be correct, while we prove the semantic soundness of subtyping in our framework. CQUAL supports qualifier inference, which is lacking in our framework. Follow-on work extended CQUAL's type system to be flow-sensitive [19], while our type system is flow-insensitive.

The type-refinement framework of Mandelbaum *et al.* [26] was also discussed in the introduction. Like CQUAL, their framework lacks a language for specifying type rules and also lacks automated soundness checking. However, the framework supports a sophisticated type system that is flow sensitive and precise in the face of computational effects, allowing protocols similar to those in the Vault language [11, 13] to be checked.

Some type systems, including the calculus of constructions [9], Nuprl [8], and sophisticated type systems [36, 10] for Proof-Carrying Code (PCC) [30] and Typed Assembly Language [29], use a form of dependent types [27] to allow predicates to be directly encoded as types. However, the proof that a predicate holds on some program fragment cannot in general be produced automatically and must instead be supplied by the programmer. Our framework is less expressive than these systems, since predicates can only be proven indirectly via type qualifiers and their associated type rules. However, the separation of typechecking, which is simple and purely syntactic, from soundness checking, which formally connects the type system to the desired predicates, allows these proofs to be performed automatically.

Dependent ML (DML) [40, 41] allows ML types to depend upon integers with linear inequality constraints. This limited form of dependent types can be used to automatically prove arithmetic program invariants, including those provable by our `int` qualifiers like `pos` and `nonzero`. DML's types can also express arithmetic invariants that relate multiple program expressions, which are not currently supported in our framework.

There is an interesting parallel between our work and research on Foundational PCC (FPCC) [2]. Rather than trusting a built-in type system (or separately proving the type system's soundness outside of the PCC framework), as is done in traditional PCC, FPCC allows a code producer to provide its own type system, along with a proof that this type system meets the code consumer's safety policy. These user-defined type systems and associated safety policies are much more general than our user-defined type qualifiers and associated invariants. However, the limited scope of our approach allows us to prove soundness fully automatically.

The technical details of our approach build on our previous work on the Cobalt and Rhodium languages [24, 25]. These languages allow users to implement dataflow analyses, which are automati-

cally proven sound by discharging proof obligations with an automatic theorem prover. Our case and assign rules are analogous to Rhodium's flow functions; the `restrict` and `disallow` rules have no analogue. Because our work is based on type systems rather than dataflow analysis, our framework's implementation and formalization are quite distinct from those of Cobalt and Rhodium. Our framework also must handle new issues, including subtyping and the distinction between value and reference qualifiers. On the other hand, Cobalt and Rhodium, being based on dataflow analysis, are naturally flow sensitive.

## 8. CONCLUSIONS AND FUTURE WORK

We have presented a new approach for supporting user-defined type refinements. We allow programmers to supply explicit type rules for new refinements, enabling the expression of common typing disciplines that would be difficult or unnatural to express in prior frameworks. An extensible typechecker executes these rules, and a soundness checker validates their correctness. We have demonstrated the approach through a framework for adding type qualifiers to C programs. Our framework supports the expression of a variety of qualifiers, and we have used these qualifiers to statically ensure interesting run-time invariants in C programs.

We plan to explore several directions to increase the expressiveness and practicality of the approach. First, we will incorporate techniques to make existing qualifiers more flexible. We are currently extending the pattern language to support special-purpose behavior for implicit assignments through procedure calls and returns, to make use of the semantics of variable scoping. Other extensions include support for qualifier inference to decrease the annotation burden and support for flow sensitivity. We also plan to insert run-time checks for qualifier casts, in order to ensure dynamically that the associated invariant holds.

Second, we will extend our framework to handle new kinds of qualifiers. Flow sensitivity will allow us to explore the specification and checking of temporal protocols. We will also target qualifiers, like `const`, whose associated invariants are naturally expressed as predicates on an entire execution trace. Reasoning automatically about execution traces is a challenge for our soundness checker. We plan to convert trace-based invariants into predicates on a single execution state by allowing users to conservatively instrument a program's dynamic semantics to record extra information [25].

## 9. ACKNOWLEDGMENTS

Thanks to Craig Chambers, Sorin Lerner, Jens Palsberg, and Ben Titzer for helpful comments on the paper.

## 10. REFERENCES

- [1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330. ACM Press, 2002.
- [2] A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 247. IEEE Computer Society, 2001.
- [3] C. Bailleux. More security problems in bftpd-1.0.12. bugtraq mailing list post of December 8, 2000. <http://www.securityfocus.com/archive/1/149977>.
- [4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230. ACM Press, 2002.
- [5] J. Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, 2001.

- [6] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. Technical Report CSD-TR-40045, UCLA Computer Science Department, November 2004. <http://www.cs.ucla.edu/~todd/stq.pdf>.
- [7] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.
- [8] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [9] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.
- [10] K. Crary and J. C. Vanderwaart. An expressive, scalable type theory for certified code. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 191–205. ACM Press, 2002.
- [11] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69. ACM Press, 2001.
- [12] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
- [13] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 13–24. ACM Press, 2002.
- [14] M. Fahndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 302–312. ACM Press, 2003.
- [15] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232. ACM Press, 2000.
- [16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, June 2002.
- [17] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003.
- [18] J. S. Foster, M. Fahndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.
- [19] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12. ACM Press, 2002.
- [20] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.
- [21] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293. ACM Press, 2002.
- [22] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th USENIX Security Symposium*, pages 119–134, 2004.
- [23] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993.
- [24] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 220–231. ACM Press, 2003.
- [25] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2005.
- [26] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 213–225. ACM Press, 2003.
- [27] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.
- [28] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
- [29] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
- [30] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
- [31] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of CC 2002: 11th International Conference on Compiler Construction*. Springer-Verlag, Apr. 2002.
- [32] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139. ACM Press, 2002.
- [33] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
- [34] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1):27–52, 1998.
- [35] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., Aug. 2001.
- [36] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 217–232. ACM Press, 2002.
- [37] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value  $\lambda$ -calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.
- [38] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2,3):167–187, 1996.
- [39] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 Nov. 1994.
- [40] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.
- [41] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.