Semantic Type Qualifiers

Brian Chin Shane Markstrum Todd Millstein

University of California, Los Angeles

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

Abstract

We present a new approach for supporting user-defined type refinements, which augment existing types to specify and check additional invariants of interest to programmers. We provide an expressive language in which users define new refinements and associated type rules. These rules are automatically incorporated by an *extensible typechecker* during static typechecking of programs. Separately, a *soundness checker* automatically proves that each refinement's type rules ensure the intended invariant, for all possible 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 `pos` and `neg` for integers, `tainted` and `untainted` for strings, and `nonnull` and `unique` for pointers, and we have applied our qualifiers to ensure important invariants on open-source C programs.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: reliability, validation; D.3.m [*Miscellaneous*]; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: invariants, mechanical verification, specification techniques

General Terms Languages, Reliability, Verification

Keywords type qualifiers, extensible typechecking, type soundness

1. Introduction

Type systems are a natural discipline for ensuring that programs maintain 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., [38, 33, 23]), invariants about pointers and their aliasing relationships (e.g., [8, 5, 1, 15]), and invariants about the interactions of threads in concurrent programs (e.g., [16, 4, 18]). 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.

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.

Of course, language designers cannot anticipate all the run-time invariants that programmers will want to specify and check. Language designers also cannot anticipate all of the practical ways in which types may be refined in order to enforce a particular invariant. Therefore, it is desirable to provide a framework for *user-defined type refinements*, whereby programmers can easily augment a language's type system with new type annotations to ensure invariants of interest. For example, Foster *et al.* describe CQUAL [19, 20], a system that allows programmers to define new *type qualifiers*, such as `nonnull` and `untainted`, for C programs, and Mandelbaum *et al.* [27] provide a theory of type refinements to specify and check properties, like temporal protocols, that depend on "effectful" 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 type rules across all type refinements. Type-refinement-specific information is 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 guarantees the desired run-time program invariants. Any errors in the user-defined typing discipline are undetected by the framework. For example, a program that is improperly annotated with `nonnull` assumptions can typecheck in CQUAL but nonetheless cause a variable declared `nonnull` to have the value `NULL` at run time.

In this paper, we address these limitations of prior frameworks via a novel approach to user-defined type refinements:

- **A language for type-refinement rules.** We provide an explicit language in which users 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 of programs, in order to enforce users' typing disciplines.
- **Semantic guarantees.** We allow users to explicitly specify the run-time invariant that a type refinement is meant to represent. We observe that for many interesting cases such invariants are quite natural and simple. A *soundness checker* automatically proves that each refinement's type rules ensure the intended invariant, for all possible programs.

We have instantiated our approach as a framework for adding user-defined type qualifiers to C programs. The extensible 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.   invariant value(E) > 0
```

Figure 1. A user-defined 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 `pos` type qualifier.

checker is implemented in the CIL [32] front end for C. The soundness checker implements a proof strategy that requires a handful of proof obligations to be discharged. Our implementation employs Simplify [14], the automatic theorem prover from the Extended Static Checker for Java (ESC/Java) [17], to automatically discharge these proof obligations. 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` [36]; and qualifiers that restrict a program's aliasing relationships, including `unique` and `unaliased`. We have used our qualifiers to ensure important program invariants on open-source C programs.

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 `unaliased`, (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 applicable to expressions of 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 declared classifier `Expr` for `E` indicates that during typechecking of a C program, `E` will be instantiated with side-effect-free program expressions. The declared type for `E` constrains such expressions to have type `int`. In addition to the classifier `Expr`, our framework supports the classifiers `Const`, `LValue`, and `Var`, which represent C 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 [32].

Given the declaration in line 1 of figure 1, programmers may now annotate their programs with the `pos` qualifier, as shown in 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 an expression that matches the pattern.

Type rules like the first `case` clause of figure 1 can be simulated in systems like CQUAL [19] 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:

$$P \quad ::= \quad X \mid *X \mid \&X \mid \texttt{new} \mid \mathit{uop}\ X \mid X\ \mathit{bop}\ X$$

Here X ranges over *variable patterns*, which have a declared type and classifier (e.g., `int Expr`) restricting the kinds of program fragments that may match. The pattern `new` matches against calls to memory allocation routines like `malloc`[1]. 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 operations on constants and on variable patterns with classifier `Const`, qualifier checks on expressions and patterns, and conjunctions and disjunctions of these kinds of predicates.

Each clause of a `case` block can be viewed as an *introduction* type rule for a qualified type, since the clause specifies conditions under which an expression may be assigned that qualified type. As

[1] Procedure calls, including `malloc`, are not in general side-effect-free and so are not considered expressions by CIL. However, it is critical for some qualifiers that pertain to pointers, for example `unique`, 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.

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

Our framework includes an extensible typechecker, which uses the type rules defined by `case` blocks, along with a set of standard rules for typechecking constructs like variable references, procedure calls, and assignments, to perform qualifier checking. Such checking validates the qualifier annotations supplied by the programmer, which represent the programmer's assumptions about when particular invariants hold.

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`. Because of their declared types, we statically know that both `prod` and `d` are positive, but this information is not sufficient to show that the expression `(prod / d)` is also positive. Indeed, the type rules for `pos` are not able to derive the type `int pos` for that expression. Therefore, the programmer must insert a cast to satisfy the typechecker, because of the declared return type of `lcm`.

A `case` block specifies when an expression may be given a qualified type. Users may then want to employ expressions having qualified types to enhance the precision of existing typechecks from the base type system. Our framework provides a `restrict` block for this purpose, an example of which is 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` augments the base rule for typechecking division expressions to require that the denominator have the type `int nonzero` (rather than simply `int`). In this way, division-by-zero errors can be detected statically instead of dynamically.

For example, consider again the `lcm` procedure in figure 2. If the extensible typechecker is given the definition of `nonzero` in addition to that of `pos`, it will use `nonzero`'s `restrict` clause to check the division in the last statement of `lcm`'s body. The `restrict` clause requires that `d` have the type `int nonzero`. By the second `case` clause for `nonzero` in figure 3, any expression of type `int pos` also has the type `int nonzero`. Since `d` is declared to have the type `int pos`, that `case` clause allows the `restrict` check to succeed.

The `restrict` clause plays a role analogous to qualifier assertions in CQUAL [19]. 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. For example, the predicate in a `restrict` clause may contain conjunctions and disjunctions of qualifier checks.

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 τ is a (possibly qualified) type, then $\tau\ q$ is considered to be a subtype of τ.

The rest of the subtyping rules are standard. As usual, care must be taken in the presence of pointers [19]. 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;
```

Section 5 contains the formal definition of our subtype relation.

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 its 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 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, without knowledge of the intended 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 automatically proves, independent of any particular program to be typechecked, that the qualifier's type rules establish this invariant.

For example, consider the definition of `pos` in figure 1. Line 9 uses the `invariant` clause to provide the qualifier's associated run-time invariant. The invariant is a predicate that is implicitly defined in the context of an arbitrary run-time execution state. Let us denote this execution state by ρ. The `value` predicate is provided by our framework and represents the value of a given expression in ρ. 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.

```
value qualifier untainted(T Expr E)

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

Figure 4. Specifying a taintedness analysis in our framework.

Given this invariant, our soundness checker generates one proof obligation for each `case` clause and automatically discharges these obligations via an off-the-shelf automatic theorem prover. Each clause's 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 ρ, then the qualifier's invariant also holds in ρ. 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 ρ is greater than zero. This obligation is easily proven, given the evaluation semantics of integer constants.

Now consider the second `case` clause for `pos`. 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 ρ, then `E` also satisfies `pos`'s invariant in ρ. This obligation is easily proven by the semantics of multiplication. On the other hand, if the pattern in that clause 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` clauses do not affect whether or not an expression of qualified type satisfies its qualifier's invariant, so `restrict` clauses are ignored by the soundness checker.

The limitations of our language for writing type rules, and of static typechecking in general, will sometimes require programmers to insert casts in order for qualifier checking to succeed. To retain soundness in this case, our extensible typechecker instruments programs with a run-time check for each cast to a value-qualified type. Each run-time check tests whether the expression being cast satisfies the cast-to qualifier's invariant. In our current implementation, a fatal error is signaled if the test fails. For example, consider the cast in the last statement of `lcm` in figure 2. At run time, a check ensures that the value of `(prod / d)` is in fact greater than zero.

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 sources. For soundness, the only requirement is that `tainted` data never flows where `untainted` data is expected. Taintedness qualifiers can help to statically detect format-string vulnerabilities in calls to `printf` and related procedures [36]. As another example, flow qualifiers `user` and `kernel` can be used to statically ensure that user pointers are never dereferenced in kernel space [24].

Figure 4 specifies the taintedness analysis in our framework. The `untainted` qualifier can qualify any type `T`. Flow qualifiers like `untainted` are a degenerate form of value qualifier in our framework. Since the `untainted` qualifier has no `case` block, the only way to introduce an expression of type `T untainted` is with a cast, to explicitly mark the expression as being trustworthy. The qualifier also lacks an explicit run-time invariant. Proper value flow is guaranteed "for free" because `T untainted` is a subtype of `T` but not vice versa. Therefore, untainted data can flow where arbitrary data is expected, but not vice versa.

```
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.

As mentioned earlier, the `untainted` qualifier can help find errors in calls to `printf` and related procedures. The first argument to `printf` is a *format string* that specifies the number and types of the remaining arguments. C does not verify, either at compile time or run time, that `printf` is always called with the appropriate number and types of arguments, as directed by the format-string argument. To ensure some measure of reliability for format strings, the programmer can use a type signature for `printf` requiring the first formal parameter to have type `char* untainted`. Therefore, only strings that are known to be trustworthy can be used as format-string arguments [36].

For example, suppose `buf` is some arbitrary (and untrusted) buffer of type `char*`. 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.

It may be useful to explicitly annotate some expressions as being possibly `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 the code snippet shown above.

2.2 Reference Qualifiers

Figure 5 defines a reference qualifier `unique`, which intuitively specifies that an l-value either has the value `NULL` or is the only reference to some memory location. Reference qualifiers pertain in part to an expression's address, so we require each reference qualifier to be applicable only to l-values or variables, rather than to arbitrary expressions.

2.2.1 Type Rules

The `assign` type rules for reference qualifiers are analogous to the `case` type rules for value qualifiers. 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 `assign` clauses for `unique` in figure 5 specify that a `unique` l-value may be assigned either the value `NULL` or the result of mem-

```
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.

```
ref qualifier unaliased(T Var X)
  ondecl
  disallow &X
  invariant forall T** P: *P != location(X)
```

Figure 7. A type qualifier for unaliased variables.

ory 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` (the cast to `int*` in the assignment to `array` is ignored for the purposes of pattern matching). The `assign` block for `unique` pertains only to assignments to the `unique` l-value itself. Assignments to *dereferences* of a `unique` l-value, such as the assignment to `array[i]` in figure 6, are unrestricted.

Because reference qualifiers pertain in part to 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 (and 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 memory location, violating `p`'s uniqueness.

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

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

Another example of a reference qualifier is shown in figure 7, representing unaliased variables. The `ondecl` keyword specifies that any variable can be given the qualifier `unaliased` at the point of its declaration. The keyword indicates that `unaliased` is a property only of a variable's address; the variable's contents are irrelevant. Therefore, an `assign` block is unnecessary for `unaliased`: implicitly, an `unaliased` variable is allowed to hold any type-correct value. The `disallow` clause for `unaliased` ensures that an unaliased variable cannot have its address taken.

Our current framework for reference qualifiers is just a first step, and we are actively exploring extensions to make it more expressive. For example, intuitively we can assign a `unique` l-value any expression that is `fresh`, meaning that the expression evaluates to a previously unreferenced memory location. In turn, a `unique` local variable returned from a procedure may be considered `fresh`. We cannot currently express this rule in our framework because patterns cannot mention procedure calls or returns. As another example, we are 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. The built-in `location` function returns the address of an l-value in a given execution state ρ, and the built-in predicate `isHeapLoc` indicates that a memory location is on the heap (i.e., it was dynamically allocated) rather than the stack. We allow the invariants for reference qualifiers to universally quantify 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. The proof obligation for an `assign` clause 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 requires an automatic theorem prover to show 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 of the `new` construct. The proof obligation for `ondecl` is similar to that for an `assign` clause, requiring the qualifier's invariant to hold for a variable upon declaration.

The proof obligations for `assign` and `ondecl` ensure that a reference qualifier's invariant is properly *established*. Our soundness checker must also show that the invariant is 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.

Unlike for value qualifiers, our extensible typechecker does not instrument programs with run-time checks for casts involving reference qualifiers. Such casts remain unchecked, as with traditional casts in C. As our examples illustrate, the invariants for reference qualifiers typically require universal quantification over all memory locations, making the associated run-time checks difficult to implement correctly and efficiently.

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. The extensible typechecker also uses value qualifiers' declared invariants to instrument the program with run-time checks for casts involving value qualifiers, as mentioned in section 2.1.3.

Our extensible typechecker is implemented as a module in CIL [32], a front end for C written in OCaml [35]. CIL parses C code into an abstract syntax tree (AST) format and provides a framework for performing passes over this AST. After qualifier checking, the AST is output as C code 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 program entities). CIL supports `gcc` 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` used 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, applying user-defined type rules to applicable program fragments. Any type errors found during qualifier checking are provided to the programmer as warnings, but compilation is 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, Expr("E1"), Expr("E2"))
```

The clause's predicate is similarly represented as follows:

```
And(Qual("pos", Expr("E1")),
    Qual("pos", Expr("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 `pos`. 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 C 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, for example arising from pointer arithmetic for array indexing.

Another 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 in library code as directed by 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 creating our own versions of these macros, we have little recourse.

Our qualifier checker can also be unsound because it, like C, allows variables to be used before being initialized. Finally, our checker is unsound in the presence of arithmetic overflow.

Because of these unsoundnesses, our extensible typechecker for C can be used to statically detect potential errors but cannot guarantee the absence of errors of a particular kind. However, the ideas underlying our framework are not specific to C and could be applied to other languages. For example, a version of our extensible typechecker for a typesafe language like Java [2, 22] would remove most of these sources of unsoundness.

4. Automated Soundness Checking

Our soundness checker takes a qualifier definition, generates the necessary proof obligations for each user-defined type rule, and automatically discharges these obligations using Simplify [14], a Nelson-Oppen-style automatic theorem prover [34]. Simplify contains decision procedures for several decidable theories, including linear arithmetic and equality for uninterpreted function symbols. 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 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 `unaliased` 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 $\rho = (\pi, \iota, \varepsilon, \sigma)$, where π is a program, ι is

an index pointing to the statement about to be executed, ε is the environment, which maps variable names to memory locations, and σ is the store, which maps locations to values.

We define several function symbols for constructing and manipulating execution states. The *state* function symbol takes a program, index, environment, and store, and it constructs an execution state. The function symbols *getStmt*, *getEnv*, and *getStore* take a state and respectively return the statement about to be executed, the environment, and the store. Environments and stores are represented as *maps*. Simplify has built-in function symbols that represent operations on maps. For example, the built-in *select* function symbol takes a map and a key and returns the key's associated value. Finally, we represent C program expressions and statements using additional function symbols. For instance, the statement $*x := \&y$ is encoded as $assign(deref(var(x)), addr(var(y)))$.

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[2]:

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

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. We do, however, explicitly model memory allocation, via a *new* function symbol.

4.2 Proof Obligations

To produce a qualifier's proof obligations, first we define a predicate 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:

$$\forall \rho, e_1, e_2.(pos(\rho, e1) \wedge pos(\rho, e2)) \Rightarrow pos(\rho, multExpr(e1, e2))$$

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 for value qualifiers and have proven that the obligations generated by our soundness checker are sufficient to ensure type soundness for such qualifiers. This section overviews these formal details; the complete formalization for value qualifiers is available in a companion technical report [7].

Stmts	s	::=	$e \mid s_1\ s_2 \mid \texttt{let}\ x = s_1\ \texttt{in}\ s_2 \mid \texttt{ref}\ s \mid s_1 := s_2$
Exprs	e	::=	$c \mid () \mid x \mid \lambda x.s \mid !e$
Consts	c	::=	integer constants
Vars	x	::=	variable names
Types	τ	::=	$\texttt{unit} \mid \texttt{int} \mid \tau_1 \to \tau_2 \mid \texttt{ref}\ \tau \mid \tau\ q$
Qualifiers	q	::=	user-defined value qualifiers

Figure 8. The syntax of the formalized language.

$$\frac{}{\tau\ q \leq \tau}\ \textsc{SubValQual} \qquad \frac{}{\tau\ q_1\ q_2 \leq \tau\ q_2\ q_1}\ \textsc{SubQualReorder}$$

$$\frac{}{\tau \leq \tau}\ \textsc{SubRef} \qquad \frac{\tau \leq \tau'' \qquad \tau'' \leq \tau'}{\tau \leq \tau'}\ \textsc{SubTrans}$$

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}\ \textsc{SubFun}$$

Figure 9. Formal rules for subtyping.

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e_1 : \tau_1\ q_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n\ q_n \qquad \text{each } e_i \text{ is a subexpression of } e}{\Gamma \vdash e : \tau\ q}\ \textsc{T-QualCase}$$

Figure 10. A formal template for user-defined `case` clauses.

5.1 Syntax and Semantics

We formalize our framework in the context of a simply-typed lambda calculus augmented with ML-style references [29] and user-defined type qualifiers. The syntax for this language is shown in figure 8. It is useful to distinguish statements, which are potentially side-effecting, from expressions, which are side-effect-free, but this separation causes no loss of expressiveness.

The static semantics is defined, as usual, by a judgment of the form $\Gamma \vdash s : \tau$, where Γ is a type environment mapping variable names to types. The inference rules defining this judgment include all the standard rules for the simply-typed lambda calculus with references. We augment these rules with the standard subsumption rule and with an associated subtyping relation, which is shown in figure 9. Rule SUBVALQUAL allows a value-qualified type to be a subtype of the associated unqualified type. Rule SUBQUALREORDER formalizes the fact that the order of qualifiers on a type is irrelevant. The other rules are standard. There is no rule for subtyping underneath `ref` types, so the type $\texttt{ref}\ \tau$ is only a subtype of itself.

To complete the static semantics, figure 10 formalizes the user-defined `case` clauses in a value qualifier's definition. The rule T-QUALCASE is actually a rule *template*: each qualifier is assumed to have zero or more rules of this form. The template allows an expression to be given a qualified type if the expression has the associated unqualified type and some of the expression's subexpressions have particular qualified types. For example, the second `case` clause in the definition of `pos` in figure 1 would be formalized by the following rule, which matches the template:

[2] Throughout this section, we elide portions of axioms and proof obligations that involve "typing predicates," which are used to restrict the domains of function symbols.

$$\frac{}{\Gamma;\texttt{int} \vdash < \sigma, c >}\ \text{Q-INT} \qquad \frac{}{\Gamma;\texttt{unit} \vdash < \sigma, () >}\ \text{Q-UNIT}$$

$$\frac{\Gamma \vdash \lambda x.s : \tau_1 \to \tau_2}{\Gamma;\tau_1 \to \tau_2 \vdash < \sigma, \lambda x.s >}\ \text{Q-FUN}$$

$$\frac{\Gamma \vdash l : \texttt{ref}\ \tau \qquad \Gamma;\tau \vdash < \sigma, \sigma(l) > \qquad l \in \text{dom}(\sigma)}{\Gamma;\texttt{ref}\ \tau \vdash < \sigma, l >}\ \text{Q-REF}$$

$$\frac{[[q]](v) \qquad \Gamma;\tau \vdash < \sigma, v >}{\Gamma;\tau\ q \vdash < \sigma, v >}\ \text{Q-QUAL}$$

Figure 11. Semantic conformance of a value to a type.

$$\frac{\Gamma \vdash e_1 * e_2 : \texttt{int}\ \overline{q} \qquad \Gamma \vdash e_1 : \texttt{int pos} \qquad \Gamma \vdash e_2 : \texttt{int pos}}{\Gamma \vdash e_1 * e_2 : \texttt{int}\ \overline{q}\ \texttt{pos}}$$

The `restrict` rules of a value qualifier do not affect soundness, so we have not included these rules in our formalization.

The dynamic semantics of our language is completely standard and is formalized via a big-step operational semantics. Programs are evaluated to values defined by the following grammar:

Values	v	::=	$c \mid () \mid \lambda x.s \mid l$
Locations	l	::=	memory locations

A store σ maps memory locations to values. The evaluation relation has the form $< \sigma, s > \to < \sigma', v >$, indicating that evaluating statement s in store σ produces value v and a new store σ'. Since expressions are side-effect-free, their evaluation is formalized by a relation of the form $< \sigma, e > \to v$.

5.2 Invariants and Proof Obligations

We model a value qualifier q's associated invariant as a unary predicate on values, denoted $[[q]]$. For example, the `invariant` clause for `pos` in figure 1 would be formalized as the predicate $[[\texttt{pos}]](v) \equiv v > 0$. We overload the $[[q]]$ notation to lift these predicates from values to arbitrary expressions:

$$[[q]](\sigma, e, v) \equiv (< \sigma, e > \to v \Rightarrow [[q]](v))$$

Finally, we formalize the proof obligation generated by our soundness checker for each user-defined type rule:

DEFINITION 5.1. A type rule matching the template T-QUALCASE from figure 10 is *locally sound* if the following proof obligation is true:

$$\forall \sigma, v_1, \ldots, v_n, v.$$
$$([[q_1]](\sigma, e_1, v_1) \wedge \ldots \wedge [[q_n]](\sigma, e_n, v_n)) \Rightarrow [[q]](\sigma, e, v)$$

5.3 Type Soundness

We have proven the soundness of our extensible type system for value qualifiers. Intuitively, soundness means that if the proof obligations generated by our soundness checker are all true, then any well-typed program fragment will satisfy its qualifiers' invariants at run time.

We formalize this notion of type soundness via a few auxiliary definitions. The relation $\Gamma;\tau \vdash < \sigma, v >$, defined in figure 11, represents *semantic conformance* of a value to a type. Intuitively, $\Gamma;\tau \vdash < \sigma, v >$ holds if $\Gamma \vdash v : \tau$ and v additionally satisfies all of the associated invariants for qualifiers in τ. The first three rules in figure 11 are the standard typechecking rules for integers, unit, and functions, respectively. Rule Q-QUAL checks that a value of qualified type satisfies the qualifier's invariant. Rule Q-REF checks that a location l is well-typed and recursively checks semantic conformance of the value that l points to in the given store. As others have done [19], for purposes of the static semantics we treat locations as variables.

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

Figure 12. The `nonnull` value qualifier.

Next we lift this notion of semantic conformance to a relation between a store and a type environment:

DEFINITION 5.2. We say that $\Gamma \sim \sigma$ if both of the following conditions hold:

1. $\text{dom}(\Gamma) = \text{dom}(\sigma)$
2. $\forall l \in \text{dom}(\Gamma).(\Gamma;\Gamma(l) \vdash < \sigma, l >)$

In other words, $\Gamma \sim \sigma$ if every memory location is well typed and satisfies its qualifiers' invariants.

Finally we can state our type soundness theorem, which is a variant of the standard type preservation theorem [39]:

THEOREM 5.1. If $\Gamma \sim \sigma$ and $\Gamma \vdash s : \tau$ and $< \sigma, s > \to < \sigma', v >$ and all user-defined type rules are locally sound, then there exists some $\Gamma' \supseteq \Gamma$ such that $\Gamma' \sim \sigma'$ and $\Gamma';\tau \vdash < \sigma', v >$.

The proof of this theorem can be found in our technical report [7].

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 in CIL is under one second.

6.1 Null Dereferences

Figure 12 shows the definition of a `nonnull` value qualifier, which is automatically proven sound by our soundness checker. The sole `case` clause indicates that the address of an l-value can be considered `nonnull`. The `restrict` clause requires all dereferences in a program to be to `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.

We applied `nonnull` annotations to variables in an iterative fashion. Running our extensible typechecker on the unannotated files produced an error message for each dereference, due to the `nonnull` 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

Table 1. Results from the `nonnull` experiment.

program:	`grep`
files:	`dfa.c`, `dfa.h`
lines:	2287
dereferences:	1072
annotations:	114
casts:	59
errors:	0

```
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->calloc, dfa->cindex);
  ++dfa->cindex;
  copyset(s, dfa->charclasses[i]);
  return i;
}
```

Figure 13. A use of `unique` in `grep`.

such imprecision is due to the flow-insensitivity of our type system. An 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 [20].

A related source of imprecision occurs when access to a `NULL`-terminated array is guarded by a test that the index is less than the value of a variable holding the array's length. Statically deducing the invariant between the array and that variable may be difficult. One possibility would be to piggyback our qualifier checker on top of CCured [33], which (among other things) can sometimes statically deduce array bounds.

Table 1 summarizes the results of our experiment. In order for the `restrict` clause in `nonnull` to succeed on all 1072 dereferences, we had to provide 114 `nonnull` annotations and 59 `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 data structure cannot be updated unexpectedly through other pointers.

We annotated several global variables in `dfa.c` with the `unique` qualifier, using the definition of `unique` 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. Our annotated declaration of `dfa` and a representative procedure manipulating `dfa` are shown in figure 13.

Other global variables were not able to be proven `unique` using our qualifier, because the variables are passed as arguments to procedures. This idiom violates the `disallow` clause for `unique` in figure 5. Indeed, this idiom is a violation 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.

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

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.* [36], 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 for `printf`. Re-running the qualifier checker then revealed an exploitable error that had been previously identified [3, 36]. 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 appropriately signals an error since the field has not been declared `untainted`.

The other two test programs were verified to have no format-string 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 untaintedness of all format-string arguments.

7. Related Work

Our framework is inspired by the CQUAL system of Foster *et al.* [19]. As mentioned in the introduction, our language for user-defined type rules is novel; a limited 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 supports explicit subtyping relationships among qualifiers, which our framework lacks. 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 and qualifier polymorphism, neither of which our framework supports. Follow-on work extended CQUAL's type system to be flow-sensitive [20], while our type system is flow-insensitive.

The type-refinement framework of Mandelbaum *et al.* [27] supports a sophisticated type system that is flow sensitive and precise in the face of computational effects, allowing temporal protocols similar to those expressible in the Vault language [12] to be statically checked. Both the framework of Mandelbaum *et al.* and Vault also support a form of polymorphism for refinements. However, those systems lack a language for user-defined type rules and lack automated soundness checking with respect to run-time invariants.

Fugue [13] is an adaptation and extension of Vault's type system to perform static checking of temporal protocols for C# [6]. Among other innovations, Fugue allows a class's typestates, which are the analogues of our qualifiers, to be given an interpretation as a predicate over the class's fields. Such predicates are used during static typechecking to ensure that each method in the class properly implements its declared specification. In this way, the Fugue typechecker directly ensures that typestates respect their intended invariants. These invariants appear to be similar in expressiveness to ours, except that Fugue's predicate language does not support quantification. Further, Fugue's type system relies heavily on built-in annotations and associated type rules for nonnull references as well as for several kinds of aliasing relationships among objects, while these kinds of qualifiers are user-defined in our framework.

Some type systems, including the calculus of constructions [10], Nuprl [9], and type systems [37, 11] for Proof-Carrying Code (PCC) [31] and Typed Assembly Language [30], use a form of dependent types [28] 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 integer qualifiers like `pos` and `nonzero`. DML's types can also express arithmetic invariants that relate multiple program expressions, which are not supported in our framework.

The technical details of our approach build on our previous work on the Cobalt and Rhodium languages [25, 26]. These languages allow users to implement dataflow analyses that can be automatically proven sound by discharging proof obligations with an automatic theorem prover. Our `case` and `assign` rules are similar to Rhodium's syntax for flow functions; our `restrict` and `disallow` rules have no analogue in Cobalt or Rhodium. 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 on programs, and a soundness checker validates the correctness of these rules once, for all possible programs. 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 open-source 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 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 a form of flow sensitivity.

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 over execution traces. 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 [26]. Also, more work is needed to find the right primitives that allow a wide variety of practical reference qualifiers to be specified and proven sound automatically.

Finally, it is possible that our approach can be used beyond type qualifiers, for example to support program checking with semantic guarantees for higher-level programming idioms such as design patterns [21].

9. Acknowledgments

This research was supported in part by NSF ITR award #0427202 and by a generous gift from Microsoft Research. Thanks to Craig Chambers, Jeff Foster, Sorin Lerner, Jens Palsberg, and Ben Titzer for helpful comments on the paper.

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] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[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] C# Language Specifi cation, Second Edition. ECMA International, Standard ECMA-334, Dec. 2002.

[7] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. Technical Report CSD-TR-40045, UCLA Computer Science Department, November 2004.

[8] 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.

[9] 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.

[10] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[11] 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.

[12] 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.

[13] R. DeLine and M. Fahndrich. Typestates for objects. In *Proceedings of the 2004 European Conference on Object-Oriented Programming*, LNCS 3086, Oslo, Norway, June 2004. Springer-Verlag.

[14] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[15] 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 programing, systems, languages, and applications*, pages 302–312. ACM Press, 2003.

[16] 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.

[17] 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.

[18] 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.

[19] J. S. Foster, M. Fähndrich, 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.

[20] 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.

[21] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[22] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[30] 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.

[31] 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.

[32] 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: 11'th International Conference on Compiler Construction*. Springer-Verlag, Apr. 2002.

[33] 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.

[34] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[35] 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.

[36] 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.

[37] 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.

[38] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-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.

[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.