Inference of User-Defined Type Qualifiers and Qualifier Rules

Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg

University of California, Los Angeles
{naerbnic, smarkstr, todd, palsberg}@cs.ucla.edu

Abstract. In previous work, we described a new approach to supporting user-defined *type qualifiers*, which augment existing types to specify and check additional properties of interest. For each qualifier, users define a set of rules that are enforced during static typechecking of programs. Separately, these rules are automatically validated with respect to a user-defined predicate that formalizes the qualifier's intended run-time invariant. We instantiated this approach as a framework for user-defined type qualifiers in C programs, called CLARITY.

In this paper, we extend our earlier approach by resolving two usability issues. First, we show how to perform qualifier inference in the presence of user-defined rules by generating and solving a system of conditional set constraints, thereby relieving users of the burden of explicitly annotating programs. Second, we show how to automatically infer rules that respect a given user-defined invariant, thereby relieving qualifier designers of the burden of manually producing such rules. We have formalized both qualifier and rule inference and proven their correctness. We have also extended CLARITY to support qualifier and rule inference, and we illustrate their utility in practice through experiments with several type qualifiers and open-source C programs.

1 Introduction

Type systems are a natural and powerful discipline for specifying and statically checking properties of programs. However, language designers cannot anticipate all of the properties that programmers will wish to specify, nor can they anticipate all of the practical ways in which such properties can be statically checked. Therefore, it is desirable to allow programmers to refine existing types in order to specify and check additional program properties. A practical form of refinement can be achieved through user-defined *type qualifiers* [6, 7].

In previous work [2], we described a new approach to user-defined type qualifiers that is more expressive and provides stronger guarantees than prior approaches. Users provide a set of *qualifier rules* in a stylized language. These rules declaratively define a qualifier's associated programming discipline and are automatically enforced during static typechecking of programs. Users may also provide a predicate that formalizes a qualifier's intended run-time invariant. This invariant is used to automatically validate the correctness of the provided qualifier rules. We instantiated this approach as a framework for user-defined type qualifiers in C programs, called CLARITY, and illustrated its utility for a variety of qualifiers, including `pos` and `neg` for integers, `nonnull` for pointers, and `tainted` and `untainted` for format strings.

P. Sestoft (Ed.): ESOP 2006, LNCS 3924, pp. 264–278, 2006.

```
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
```

Fig. 1. A user-defined type qualifiers for nonzero integers.

In this paper, we extend our earlier approach by resolving two usability issues. First, we show how to perform qualifier inference, relieving programmers of the burden of explicitly annotating their programs. We describe an inference algorithm that is parameterized by a set of user-defined qualifier rules. The algorithm generates and solves a system of *conditional set constraints*. We have formalized constraint generation and proven that the constraint system is equivalent to the associated qualifier type system. We have also extended CLARITY to support qualifier inference and have used it to infer qualifiers on several open-source C programs.

Second, we show how to automatically infer rules that respect a given user-defined invariant, relieving qualifier designers of the burden of manually producing such rules. We define a partial order of candidate rules that formalizes the situation when one rule subsumes another. We then describe an algorithm for walking this partial order to generate all valid rules that are not subsumed by any other valid rule. We have implemented rule inference in CLARITY and used it to automatically generate all of our manually produced qualifier rules as well as some valid rules we had not thought of.

The next section reviews our approach to user-defined type qualifiers in the context of CLARITY. Sections 3 and 4 describe qualifier inference and rule inference, respectively. Section 5 discusses our experiments with qualifier and rule inference in CLARITY. Section 6 compares with related work, and section 7 concludes.

2 An Overview of CLARITY

2.1 Qualifier Rules and Qualifier Checking

Figure 1 illustrates the definition of a simple user-defined qualifier for nonzero integers in CLARITY.[1] The first line defines the qualifier `nonzero` to be applicable to all expressions of type `int`. The `case` and `restrict` blocks provide the user-defined typing

[1] This paper focuses on CLARITY's "value" qualifiers; its "reference" qualifiers are not considered [2].

discipline associated with the new qualifier. Each clause in the `case` block represents a qualifier rule, consisting of some *metavariable* declarations for use in the rest of the rule, a *pattern*, and a *condition*. The first `case` clause in Figure 1 indicates that an integer constant can be given the qualifier `nonzero` if the constant is not equal to zero. The second clause indicates that an expression can be given the qualifier `nonzero` if it can be given another user-defined qualifier `pos`, whose definition is not shown. The third clause indicates that the product of two `nonzero` expressions can also be considered `nonzero`.

A `restrict` clause has the same syntax as a `case` clause, but the semantics is different. Namely, a `restrict` clause indicates that whenever the pattern is matched by some program expression, then the condition should also be satisfied by that expression. Therefore, the `restrict` clause in Figure 1 indicates that all denominator expressions in a division must have the qualifier `nonzero`.

CLARITY includes a *qualifier checker* that statically enforces user-defined qualifier rules on programs. As a simple example, consider the statement

```
nonzero int prod = a*b;
```

where `a` and `b` are each declared to have the type `pos int`. Since `prod` is declared to have the qualifier `nonzero`, the qualifier checker must ensure that `a*b` can be given this qualifier. By the third `case` clause for `nonzero` in Figure 1, the check succeeds if it can be shown that `a` and `b` each recursively has qualifier `nonzero`. Each of these recursive checks succeeds by the second `case` clause for `nonzero`, since `a` and `b` are each declared to have qualifier `pos`.

In general, each program expression can be associated with a set of qualifiers. Qualifier checking employs a natural notion of subtyping in the presence of user-defined qualifiers: a type $Q_1\tau$ subtypes another type $Q_2\tau$, where Q_1 and Q_2 are sets of qualifiers and τ is an unqualified type, if $Q_1 \supseteq Q_2$. We have formalized this notion of subtyping and proven that it is sound for all user-defined qualifiers expressible in our rule language [2]. There is no direct notion of subtyping between qualifiers, but this can be encoded in the rules. For example, the second `case` clause for `nonzero` in Figure 1 has the effect of making `pos int` a subtype of `nonzero int`.

2.2 Qualifier Invariants and Qualifier Validation

In addition to the qualifier rules, CLARITY allows users to provide a predicate that formalizes a qualifier's intended run-time invariant. For example, the `invariant` clause for `nonzero` in Figure 1 indicates that the value of an expression qualified with `nonzero` should not equal zero, in all run-time execution states. The invariant makes use of a `value` predicate that our framework provides. Given a qualifier's invariant, CLARITY's *qualifier validator* component ensures that the qualifier's rules are correct, in the sense that they respect this invariant. Qualifier validation happens once, independent of any particular program that uses the qualifier. For each `case` clause, the qualifier validator generates one proof obligation to be discharged.[2] Our implementation discharges obligations with the Simplify automatic theorem prover [5].

[2] We do not validate `restrict` rules, whose correctness depends on a user-specific notion of run-time error.

Each proof obligation requires that a rule's pattern and condition are sufficient to ensure the qualifier's invariant at run time. For example, the qualifier validator generates the following obligation for the first `case` clause for `nonzero` in figure 1: if an expression `E` is an integer constant other than zero, then the value of `E` in an arbitrary execution state is not equal to zero. For the third `case` clause, the qualifier validator generates the following obligation: if an expression `E` has the form `E1 * E2` and both `E1` and `E2` satisfy `nonzero`'s invariant in an arbitrary execution state, then `E` also satisfies `nonzero`'s invariant in that state. These obligations are easily discharged by Simplify.[3] On the other hand, if the pattern in the third `case` clause were erroneously specified as `E1 + E2`, the qualifier validator would catch the error, since it is not possible to prove that the sum of two nonzero integers is always nonzero.

CLARITY's qualifier validator is currently limited by the capabilities of Simplify, which includes decision procedures for propositional logic, linear arithmetic, and equality with uninterpreted functions, and additionally includes heuristics for handling first-order quantification. Simplify works well for many kinds of properties, for example arithmetic invariants and simple invariants about pointers such as nonnullness. Simplify is not tailored for reasoning about other useful kinds of invariants, for example shape invariants on data structures. However, our approach to qualifier validation could easily be adapted for use with other decision procedures and theorem provers, including tools requiring some user interaction.

3 Qualifier Inference

The original CLARITY system supports qualifier *checking*: all variables must be explicitly annotated with their qualifiers. In this section, we show how to support qualifier *inference* in the presence of user-defined qualifier rules. We formalize qualifier inference for a simply-typed lambda calculus with references and user-defined qualifiers, as defined by the following grammar:

$$e ::= c \mid e_1 + e_2 \mid x \mid \lambda x : \tau.e \mid e_1\ e_2 \mid \texttt{ref}\ e \mid e_1 := e_2 \mid !e \mid \texttt{assert}(e, q)$$
$$\tau ::= \texttt{int} \mid \tau_1 \to \tau_2 \mid \texttt{ref}\ \tau$$

Let Q be the set $\{q_1, \ldots, q_n\}$ of user-defined qualifiers in a program. Sets of qualifiers from Q form a natural lattice, with partial order $\supseteq$, least-upper-bound function $\cap$, and greatest-lower-bound function $\cup$. We denote elements of this lattice by metavariable l; qualified types are ranged over by metavariable ρ and are defined as follows:

$$\rho ::= l\ \phi \qquad \phi ::= \texttt{int} \mid \rho_1 \to \rho_2 \mid \texttt{ref}\ \rho$$

We present both a type system and a constraint system for qualifier inference and prove their equivalence, and we describe an algorithm for solving the generated constraints. We assume the bound variables in expressions are annotated with unqualified types τ. It is possible to combine qualifier inference with type inference, but separating them simplifies the presentation.

[3] Our qualifier validator currently does not properly model overflow.

3.1 Formal Qualifier Rules

We formalize the `case` rules as defining two kinds of relations. First, some `case` clauses have the effect of declaring a specificity relation between qualifiers. We formalize these rules as defining axioms for a relation of the form $q_1 \triangleright q_2$. For example, the second `case` clause in Figure 1 would be represented by the axiom $\texttt{pos} \triangleright \texttt{nonzero}$. We use $\triangleright^*$ to denote the reflexive, transitive closure of the user-defined $\triangleright$ relation, and we require $\triangleright^*$ to be a partial order.

The other kind of `case` clause uses a pattern to match on a constructor (e.g., +), and the clause determines the qualifier of the entire expression based on the qualifiers of the immediate subexpressions. We formalize these rules as defining relations of the form R^q_p, where q is a qualifier and p represents one of the constructors in our formal language, ranging over integer constants and the symbols +, λ, and `ref`. The arity of each relation R^q_p is the number of immediate subexpressions of the constructor represented by p, and the domain of each argument to the relation is Q. Each `case` clause is formalized through axioms for these relations. For example, the third `case` clause in Figure 1 would be represented by the axiom $R^{\texttt{nonzero}}_*(\texttt{nonzero}, \texttt{nonzero})$ (if our formal language contained multiplication). The first `case` clause in that figure would be formalized through the (conceptually infinite) set of axioms $R^{\texttt{nonzero}}_1()$, $R^{\texttt{nonzero}}_2()$, etc. For simplicity of presentation, we assume that each subexpression is required to satisfy only a single qualifier. In fact, our implementation allows each subexpression to be constrained to satisfy a set of qualifiers, and it would be straightforward to update our formalism to support this ability.

Finally, we formalize the `restrict` rules with an expression of the form $\texttt{assert}(e, q)$, which requires the type system to ensure that the top-level qualifier on expression e's type includes qualifier q. For example, the `restrict` rule in Figure 1 is modeled by replacing each denominator expression e in a program with $\texttt{assert}(e, \texttt{nonzero})$. The `assert` expression can also be used to model explicit qualifier annotations in programs.

3.2 The Type System

We assume we are given an expression e along with a set A of axioms representing the user-defined qualifier rules, as described above. The qualifier type system is presented in Figure 3, and the axioms in A are implicitly considered to augment this formal system. As usual, metavariable Γ ranges over type environments, which map variables to qualified types. The rule for $\texttt{assert}(e, q)$ infers a qualified type for e and then checks that q is in the top-level qualifier of this type. The *strip* function used in the rule for lambdas removes all qualifiers from a qualified type ρ, producing an unqualified type τ.

The main novelty in the type system is the consultation of the axioms in A to produce the top-level qualifiers for constructor expressions. For example, consider the first rule in Figure 3, which infers the qualifiers for an integer constant c using a set comprehension notation. The resulting set l includes all qualifiers q' such that the $R^{q'}_c()$ relation holds (according to the axioms in A), as well as all qualifiers q that are "less specific" than such a q' as defined by the $\triangleright^*$ relation. In this way, the rule finds all possible qualifiers that can be proven to hold given the user-defined `case` clauses. The subsumption

$$\frac{l_1 \supseteq l_2}{l_1\,\texttt{int} \le l_2\,\texttt{int}} \qquad \frac{l_1 \supseteq l_2 \qquad \rho \le \rho' \qquad \rho' \le \rho}{l_1\ \texttt{ref}\ \rho \le l_2\ \texttt{ref}\ \rho'} \qquad \frac{l_1 \supseteq l_2 \qquad \rho_2 \le \rho_1 \qquad \rho_1' \le \rho_2'}{l_1(\rho_1 \to \rho_1') \le l_2(\rho_2 \to \rho_2')}$$

Fig. 2. Formal subtyping rules for qualified types

$$\frac{l = \{q \mid R_c^{q'}() \wedge q' \rhd^* q\}}{\Gamma \vdash c : l\ \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : l_1\ \texttt{int} \qquad \Gamma \vdash e_2 : l_2\ \texttt{int} \qquad l = \{q \mid R_+^{q'}(q_1,q_2) \wedge q_1 \in l_1 \wedge q_2 \in l_2 \wedge q' \rhd^* q\}}{\Gamma \vdash e_1 + e_2 : l\ \texttt{int}} \qquad \frac{\Gamma(x) = \rho}{\Gamma \vdash x : \rho}$$

$$\frac{\mathit{strip}(\rho_1) = \tau_1 \qquad \Gamma, x : \rho_1 \vdash e : \rho_2 \qquad \rho_2 = l_2\,\phi_2 \qquad l = \{q \mid R_\lambda^{q'}(q_2) \wedge q_2 \in l_2 \wedge q' \rhd^* q\}}{\Gamma \vdash \lambda x : \tau_1.e : l(\rho_1 \to \rho_2)} \qquad \frac{\Gamma \vdash e_1 : l(\rho_2 \to \rho) \qquad \Gamma \vdash e_2 : \rho_2}{\Gamma \vdash e_1\ e_2 : \rho}$$

$$\frac{\Gamma \vdash e : \rho \qquad \rho = l_0\,\phi_0 \qquad l = \{q \mid R_{\texttt{ref}}^{q'}(q_0) \wedge q_0 \in l_0 \wedge q' \rhd^* q\}}{\Gamma \vdash \texttt{ref}\ e : l\ \texttt{ref}\ \rho} \qquad \frac{\Gamma \vdash e_1 : l\ \texttt{ref}\ \rho \qquad \Gamma \vdash e_2 : \rho}{\Gamma \vdash e_1 := e_2 : \rho}$$

$$\frac{\Gamma \vdash e : l\ \texttt{ref}\ \rho}{\Gamma \vdash\, !e : \rho} \qquad \frac{\Gamma \vdash e : \rho \qquad \rho = l\,\phi \qquad q \in l}{\Gamma \vdash \texttt{assert}(e,q) : \rho} \qquad \frac{\Gamma \vdash e : \rho' \qquad \rho' \le \rho}{\Gamma \vdash e : \rho}$$

Fig. 3. Formal qualifier inference rules

rule at the end of the figure can then be used to forget some of these qualifiers, via the subtyping rules in Figure 2. The inference of top-level qualifiers is similar for the other constructors, except that consultation of the R relation makes use of the top-level qualifiers inferred for the immediate subexpressions.

3.3 The Constraint System

In this section we describe a constraint-based algorithm for qualifier inference. The key novelty is the use of a specialized form of *conditional constraints* to represent the effects of user-defined qualifier rules. The metavariable α represents *qualifier variables*, and we generate constraints of the following forms:

$$\alpha \supseteq \alpha \qquad q \in \alpha \qquad q \in \alpha \Rightarrow \bigvee(\bigwedge q \in \alpha)$$

Given a set C of constraints, let S be a mapping from the qualifier variables in C to sets of qualifiers. We say that S is a *solution* to C if S satisfies all constraints in C. We say that S is the *least solution* to C if for all solutions S' and qualifier variables α in the domain of S and S', $S(\alpha) \supseteq S'(\alpha)$. It is easy to show that if a set of constraints C in the above form has a solution, then it has a unique least solution.

Constraint Generation. We formalize constraint generation by a judgment of the form $\kappa \vdash e : \delta \mid C$. Here C is a set of constraints in the above form, and the metavariable δ represents qualified types whose qualifiers are all qualifier variables:

$$\delta ::= \alpha\,\varphi \qquad \varphi ::= \texttt{int} \mid \delta_1 \to \delta_2 \mid \texttt{ref}\ \delta$$

$$\alpha_1\,\texttt{int} \sqsubseteq \alpha_2\,\texttt{int} \equiv \{\alpha_1 \supseteq \alpha_2\}$$
$$\alpha_1\,\texttt{ref}\ \delta_1 \sqsubseteq \alpha_2\,\texttt{ref}\ \delta_2 \equiv \{\alpha_1 \supseteq \alpha_2\} \cup \delta_1 \sqsubseteq \delta_2 \cup \delta_2 \sqsubseteq \delta_1$$
$$\alpha_1(\delta_1 \to \delta_1') \sqsubseteq \alpha_2(\delta_2 \to \delta_2') \equiv \{\alpha_1 \supseteq \alpha_2\} \cup \delta_2 \sqsubseteq \delta_1 \cup \delta_1' \sqsubseteq \delta_2'$$

Fig. 4. Converting type constraints into set constraints

$$\frac{\alpha' \text{ fresh} \qquad \delta' = \alpha'\ \texttt{int} \qquad \delta = \mathit{refresh}(\delta')}{\kappa \vdash c : \delta \mid \delta' \sqsubseteq \delta \cup \{C_c^q(\alpha') \mid q \in Q\}}$$

$$\frac{\kappa \vdash e_1 : \alpha_1\ \texttt{int} \mid C_1 \qquad \kappa \vdash e_2 : \alpha_2\ \texttt{int} \mid C_2 \qquad \alpha' \text{ fresh} \qquad \delta' = \alpha'\ \texttt{int} \qquad \delta = \mathit{refresh}(\delta')}{\begin{array}{c}\kappa \vdash e_1 + e_2 : \delta \mid \\ C_1 \cup C_2 \cup \delta' \sqsubseteq \delta \cup \{C_+^q(\alpha_1, \alpha_2, \alpha') \mid q \in Q\}\end{array}}$$

$$\frac{\kappa(x) = \delta' \qquad \delta = \mathit{refresh}(\delta')}{\kappa \vdash x : \delta \mid \delta' \sqsubseteq \delta}$$

$$\frac{\kappa, x : \delta_1 \vdash e : \delta_2 \mid C \qquad \delta_1 = \mathit{embed}(\tau_1) \qquad \delta_2 = \alpha_2\ \varphi_2 \qquad \alpha' \text{ fresh} \qquad \delta' = \alpha'(\delta_1 \to \delta_2) \qquad \delta = \mathit{refresh}(\delta')}{\kappa \vdash \lambda x : \tau_1 . e : \delta \mid C \cup \delta' \sqsubseteq \delta \cup \{C_\lambda^q(\alpha_2, \alpha') \mid q \in Q\}}$$

$$\frac{\kappa \vdash e_1 : \alpha(\delta_2 \to \delta') \mid C_1 \qquad \kappa \vdash e_2 : \delta_2' \mid C_2 \qquad \delta = \mathit{refresh}(\delta')}{\begin{array}{c}\kappa \vdash e_1\ e_2 : \delta \mid \\ C_1 \cup C_2 \cup \delta_2' \sqsubseteq \delta_2 \cup \delta' \sqsubseteq \delta\end{array}}$$

$$\frac{\kappa \vdash e : \delta_0 \mid C \qquad \delta_0 = \alpha_0\ \varphi_0 \qquad \alpha' \text{ fresh} \qquad \delta' = \alpha'\ \texttt{ref}\ \delta_0 \qquad \delta = \mathit{refresh}(\delta')}{\kappa \vdash \texttt{ref}\ e : \delta \mid C \cup \delta' \sqsubseteq \delta \cup \{C_{\texttt{ref}}^q(\alpha_0, \alpha') \mid q \in Q\}}$$

$$\frac{\kappa \vdash e_1 : \alpha\ \texttt{ref}\ \delta' \mid C_1 \qquad \kappa \vdash e_2 : \delta'' \mid C_2 \qquad \delta = \mathit{refresh}(\delta')}{\begin{array}{c}\kappa \vdash e_1 := e_2 : \delta \mid \\ C_1 \cup C_2 \cup \delta'' \sqsubseteq \delta' \cup \delta' \sqsubseteq \delta\end{array}}$$

$$\frac{\kappa \vdash e : \alpha\ \texttt{ref}\ \delta' \mid C \qquad \delta = \mathit{refresh}(\delta')}{\kappa \vdash\, !e : \delta \mid C \cup \delta' \sqsubseteq \delta}$$

$$\frac{\kappa \vdash e : \delta' \mid C \qquad \delta' = \alpha\ \phi \qquad \delta = \mathit{refresh}(\delta')}{\begin{array}{c}\kappa \vdash \texttt{assert}(e, q) : \delta \mid \\ C \cup \{q \in \alpha\} \cup \delta' \sqsubseteq \delta\end{array}}$$

Fig. 5. Formal constraint generation rules for qualifier inference

The metavariable κ denotes type environments that map program variables to qualified types of the form δ.

The inference rules defining this judgment are shown in Figure 5. The *embed* function adds fresh qualifier variables to an unqualified type τ in order to turn it into a qualified type δ, and $\mathit{refresh}(\delta)$ is defined as $\mathit{embed}(\mathit{strip}(\delta))$. To keep the constraint generation purely syntax-directed, subsumption is "built in" to each rule: the *refresh* function is used to create a fresh qualified type δ, which is constrained by a subtype constraint of the form $\delta' \sqsubseteq \delta$. Subtype constraints are also generated for applications and assignments, as usual. We treat a subtype constraint as a shorthand for a set of qualifier-variable constraints, as shown in Figure 4.

Each rule for an expression with top-level constructor p produces one conditional constraint per qualifier q in Q, denoted C_p^q. Informally, the constraint C_p^q *inverts* the user-defined qualifier rules, indicating all the possible ways to prove that an expression with constructor p can be given qualifier q according to the axioms in A. For example, both the second and third `case` clauses in Figure 1 can be used to prove that a product `a*b` has the qualifier `nonzero`, so our implementation of constraint generation in CLARITY produces the following conditional constraint:

$$\texttt{nonzero} \in \alpha_{\texttt{a*b}} \Rightarrow ((\texttt{nonzero} \in \alpha_{\texttt{a}} \wedge \texttt{nonzero} \in \alpha_{\texttt{b}}) \vee (\texttt{pos} \in \alpha_{\texttt{a*b}}))$$

More formally, let $zip(R_p^q(q_1,\ldots,q_m),\alpha_1,\ldots,\alpha_m)$ denote the constraint $q_1 \in \alpha_1 \wedge \ldots \wedge q_m \in \alpha_m$. Let $\{a_1,\ldots,a_u\}$ be all the axioms in A for the relation R_p^q, and let $\{q_1,\ldots,q_v\} = \{q' \in Q \mid q' \triangleright q\}$. Then $C_p^q(\alpha_1,\ldots,\alpha_m,\alpha')$ is the following conditional constraint:

$$q \in \alpha' \Rightarrow (\bigvee_{1\le i\le u} zip(a_i,\alpha_1,\ldots,\alpha_m) \vee \bigvee_{1\le i\le v} q_i \in \alpha')$$

We have proven the equivalence of our constraint system with the type system presented in the previous subsection; details are in our companion technical report [3].

Theorem: $\emptyset \vdash e : \rho$ if and only if $\emptyset \vdash e : \delta \mid C$ and there exists a solution S to C such that $S(\delta) = \rho$.

Constraint Solving. We solve the constraints by a graph-based propagation algorithm, which either determines that the constraints are unsatisfiable or produces the unique least solution. Figure 6 shows a portion of the constraint graph generated for the statement `int prod = a*b;`. On the left side, the graph includes one node for each qualifier variable, which is labeled with the corresponding program expression. Each node contains a bit string of length $|Q|$ (not shown in the figure), representing the qualifiers that may be given to the associated expression. All bits are initialized to true, indicating that all expressions may be given all qualifiers. If bit i for node α ever becomes false during constraint solving, this indicates that α cannot include the ith qualifier in any solution.

Because our algorithm propagates the *inability* for an expression to have a qualifier, the direction of flow is opposite what one might expect. For each generated constraint of the form $\alpha_1 \supseteq \alpha_2$, the graph includes an edge from α_1 to α_2. For each conditional constraint, the graph contains a representation of its *contrapositive*. For example, the right side of Figure 6 shows an *and-or* tree that represents the following constraint:

$$((\texttt{nonzero} \notin \alpha_{\texttt{a}} \vee \texttt{nonzero} \notin \alpha_{\texttt{b}}) \wedge (pos \notin \alpha_{\texttt{a*b}})) \Rightarrow \texttt{nonzero} \notin \alpha_{\texttt{a*b}}$$

The tree's root has an outgoing edge to the `nonzero` bit of the node `a*b`, and the leaves similarly have incoming `nonzero`-bit edges. In the figure, edges to and from individual

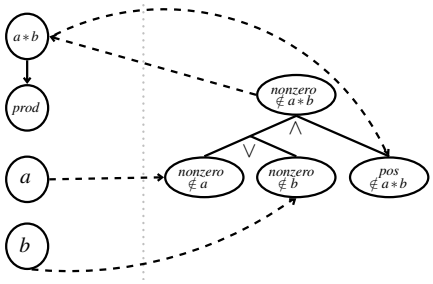

Fig. 6. An example constraint graph

bits are dotted. The root of each and-or tree maintains a counter of the number of subtrees it is waiting for before it can "fire." Our example tree has a counter value of 2.

To solve the constraints, we visit the root of each and-or tree once. If its counter is greater than 0, we do nothing. Otherwise, the outgoing edge from its root is traversed, which falsifies the associated bit and propagates this falsehood to its successors recursively until quiescence. For example, if the and-or tree in Figure 6 ever fires, that will falsify the `nonzero` bit of `a*b`, which in turn will falsify the `nonzero` bit of `prod`.

After the propagation phase is complete, we employ the constraints of the form $q \in \alpha$ to check for satisfiability. For each such constraint, if the bit corresponding to qualifier q in node α is false, then we have a contradiction and the constraints are unsatisfiable. Otherwise, the least solution is formed by mapping each qualifier variable α to the set of all qualifiers whose associated bit in node α is true.

Complexity Analysis. Let n be the size of a program, m be the size of the axioms in A, and q be the number of user-defined qualifiers. There are $O(n)$ qualifier variables, $O(n^2)$ constraints of the form $\alpha \supseteq \alpha$, $O(qn)$ constraints of the form $q \in \alpha$, and $O(qn)$ conditional constraints generated, each with size $O(m)$. Therefore, the constraint graph has $O(n^2)$ edges between qualifier-variable nodes, each of which can be propagated across q times. There are $O(qnm)$ edges in total for the and-or trees, and there are $O(qnm)$ edges between the qualifier-variable nodes and the and-or trees, each of which can be propagated across once. Therefore, the total number of propagations, and hence the total time complexity, is $O(qn(n+m))$.

4 Rule Inference

Writing qualifier rules can be tedious and error prone. The qualifier validator that is part of our framework reduces errors by checking that each user-defined rule respects its associated qualifier's invariant. However, other errors are possible. For example, a user-defined rule may be correct but be overly specific, and there may be useful rules that are completely omitted. To reduce the burden on qualifier designers and to reduce these kinds of errors, we have created a technique for automatically *inferring* correct `case` rules from a qualifier's invariant.

A naive approach to rule inference is to generate each candidate rule and use the qualifier validator to remove all candidates that do not respect the intended invariant. However, since qualifier validation is relatively expensive, requiring usage of decision procedures, and since there are an exponential number of candidates in the number of qualifiers, it is desirable to minimize the number of candidates that need to be explicitly considered.[4] To efficiently search the space of candidate rules, we define a partial order $\preceq$ that formalizes the situation when one candidate subsumes another.

The most precise partial ordering on `case` clauses is logical implication. For example, the third `case` clause in Figure 1 corresponds to the following formula, obtained by replacing qualifiers with their invariants:

[4] Conceptually, there are an infinite number of candidates, due to constants. We handle constants through a simple heuristic that works well in practice. For each qualifier, we only consider a single candidate rule (possibly) containing constants, which is derived from the qualifier's invariant by replacing all references to `value(E)` with a metavariable ranging over constants.

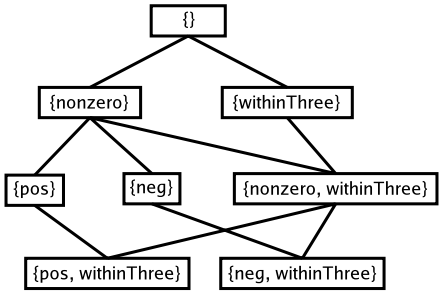

Fig. 7. An example of $\preceq_S$ for four qualifiers

$$\texttt{value}(\texttt{E1}) \neq 0 \wedge \texttt{value}(\texttt{E2}) \neq 0 \Rightarrow \texttt{value}(\texttt{E1} * \texttt{E2}) \neq 0$$

The above clause subsumes a clause that requires both `E1` and `E2` to be `pos` instead of `nonzero`, since the above formula logically implies the formula associated with the new clause. Unfortunately, precisely computing this partial order requires an exponential number of calls to decision procedures to reason about logical implication, which is exactly what we are trying to avoid.

Instead, our approach is to use logical implication to define a partial ordering on individual qualifiers, but to then lift this partial ordering to `case` clauses in a purely syntactic way. Therefore, we need only make a quadratic number of calls to the decision procedures in order to compute the partial order. This approximation of the "true" partial ordering is still guaranteed to completely exhaust the space of candidates, but it is now possible to produce qualifier rules that are redundant. As we show in Section 5, however, our approach works well in practice. The rest of this section formalizes our partial order and describes the rule inference algorithm; more details are in our companion technical report [3].

The Partial Order. We assume that every qualifier $q \in Q$ has an invariant, which is a unary predicate that we also denote q. We also assume that no two qualifiers have logically equivalent invariants. Then we define a partial order $\preceq_Q$ on qualifiers as follows:

$$q_1 \preceq_Q q_2 \stackrel{\Delta}{=} \forall x.q_1(x) \Rightarrow q_2(x)$$

This partial order is computed by $|Q|^2$ queries to Simplify. We similarly use $|Q|^2$ Simplify queries to compute mutual exclusivity of pairs of qualifiers:

$$q_1 \perp_Q q_2 \stackrel{\Delta}{=} \forall x.\neg(q_1(x) \wedge q_2(x))$$

Let $S = \mathcal{P}(Q)$. We lift $\preceq_Q$ to sets of qualifiers (or *qualsets*) s_1 and s_2 in S as follows:

$$s_1 \preceq_S s_2 \stackrel{\Delta}{=} \forall q_2 \in s_2.\exists q_1 \in s_1.q_1 \preceq_Q q_2$$

When considering qualsets for use in a candidate `case` clause, we restrict our attention to qualsets that are *canonical*. We define a set $s \in S$ as canonical if the following condition holds:

$$\forall q_1, q_2 \in s. \neg(q_1 \perp_Q q_2) \wedge (q_1 \preceq_Q q_2 \Rightarrow q_1 = q_2)$$

It is easy to prove that $\preceq_S$ is a partial order on canonical qualsets. An example of the $\preceq_S$ partial order over canonical qualsets that may include any of the four qualifiers `nonzero`, `pos`, `neg`, and `withinThree` (whose invariant requires the value to be ≥ -3 and ≤ 3) is shown in Figure 7. We lift $\preceq_S$ to tuples of canonical qualsets in the obvious way:

$$(s_1, \ldots, s_k) \preceq_T (t_1, \ldots, t_k) \stackrel{\Delta}{=} \forall i \in \{1, \ldots, k\}. s_i \preceq_S t_i$$

Finally, we can describe the partial order on candidate `case` clauses. A candidate c can be considered to be a triple containing the constructor p used as the pattern; a tuple of qualsets $(s_1, \ldots, s_k)$, one per subexpression of p, representing the clause's condition; and the qualifier q that the clause is defined for. We define the partial ordering on `case` clauses c_1 and c_2 as follows:

$$\begin{aligned}(p_1, (s_1, \ldots, s_k), q_1) \preceq (p_2, (t_1, \ldots, t_j), q_2) \stackrel{\Delta}{=} \\ p_1 = p_2 \wedge k = j \wedge (t_1, \ldots, t_k) \preceq_T (s_1, \ldots, s_k) \wedge q_1 \preceq_Q q_2\end{aligned}$$

We have proven that if $c_1 \preceq c_2$ then in fact c_1 logically implies c_2 [3].

The Algorithm. Consider generating all valid `case` rules for a single qualifier q. Further, fix a particular constructor p to use in the rule's pattern, and assume that this constructor has exactly one subexpression. Let W be a worklist of pairs of the form (s, l) where s is a qualset and l is a list of qualifiers. Initialize the set W to $\{(\emptyset, [q_1, q_2, q_3, \ldots])\}$, where $[q_1, q_2, q_3, \ldots]$ is an ordered list of all the qualifiers in reverse topological order according to $\preceq_Q$. Using reverse topological order ensures we will generate qualsets for use in a `case` rule from most-general to most-specific, which is necessary given the contravariance in the definition of $\preceq$. We similarly maintain W in sorted order according to a reverse topological sort of the first component of each pair. We also maintain a set T of valid `case` rules, initialized to $\emptyset$.

1. If W is empty, we are done and T contains all the valid non-redundant rules. Otherwise, remove the first pair (s, l) in W.
2. If there is some candidate $(p', s', q') \in T$ such that $(p', s', q') \preceq (p, s, q)$ then s is redundant, so we drop the pair (s, l) and return to the previous step. Otherwise, we continue to the next step.
3. We run our framework's qualifier validator on (p, s, q). If it passes, we add (p, s, q) to T. If not, then we need to check less-specific candidates. For each $q \in l$, we add the pair $(s \cup \{q\}, l')$ to W, where l' is the suffix of l after q. These pairs are placed appropriately in W to maintain its sortedness, as described earlier.

In the case when the constructor p has $k > 1$ subexpressions, we need to enumerate k-ary multisets. To do so, the worklist W now contains k-tuples of pairs of the form (s, l). When adding new elements to W, we apply the procedure described in Step 3 above to each component of the k-tuple individually, keeping all other components unchanged. The only subtlety is that we want to avoid generating redundant tuples. For example, if $q_1 \preceq_Q q_2$, then the tuple $(\{q_2\}, \{q_2\})$ could be a successor of both $(\{q_1\}, \{q_2\})$ and

Table 1. Qualifier inference results

qualifier sets			nonnull			nonnull/pos/neg/nz		
program	kloc	vars	cons	gen (s)	solv (s)	cons	gen (s)	solv (s)
identd-1.0	0.19	624	1381	0.09	0.01	2757	0.15	0.01
mingetty-0.9.4	0.21	488	646	0.04	0.01	1204	0.06	0.01
bftpd-1.0.11	2.15	1773	3768	0.39	0.05	6426	0.58	0.08
bc-1.04	4.75	4769	14913	1.21	0.13	27837	5.78	0.18
grep-2.5	10.43	4914	15719	0.75	0.55	28343	7.84	0.71
snort-2.06	52.11	29013	99957	36.39	46.81	176852	290.24	58.07

$(\{q_2\},\{q_1\})$. To avoid this duplication, we only augment a component of a k-tuple when generating new candidates for W in Step 3 if it is either the component that was last augmented along this path of the search, or it is to the right of that component. This rule ensures that once a component is augmented, the search cannot "go back" and modify components to its left. In our example, $(\{q_2\},\{q_2\})$ would not be generated from $(\{q_1\},\{q_2\})$ in Step 3, because the last component to have been augmented must have been the second one (since all components begin with the empty set).

Finally we describe the full algorithm for candidate generation. We enumerate each qualifier q in topological order according to $\preceq_Q$. For each such qualifier, we enumerate each constructor p in any order and use the procedure described above to generate all the valid non-redundant rules of the form $(p,(s_1,\ldots,s_k),q)$. The set T is initialized to $\emptyset$ at the beginning of this algorithm and is augmented throughout the entire process. In this way, candidates shown to be valid for some qualifier q can be used to find a later candidate for a target q' to be redundant. For example, a rule allowing the sum of two `pos` expressions to be considered `pos` will be found to subsume a rule allowing the sum of two `pos` expressions to be considered `nonzero`. When this algorithm completes, the set T will contain all valid rules such that none is subsumed by any other valid rule according to $\preceq$. Finally, we augment T with rules that reflect the specificity relation among qualifiers, such as the second `case` rule in Figure 1. These rules are derived directly from the computed $\preceq_Q$ relation.

5 Experiments

5.1 Qualifier Inference

We implemented qualifier inference in CLARITY and ran it on six open-source C programs, ranging from a few hundred to over 50,000 lines of code, as shown in Table 1. Each test case was run through the inferencer twice. The first time, the inferencer was given a definition only for a version of `nonnull`, with a `case` clause indicating that an expression of the form `&E` can be considered `nonnull` and a `restrict` clause requiring dereferences to be to `nonnull` expressions. The second time, the inferencer was additionally given versions of the qualifiers `pos`, `neg`, and `nonzero` for integers, each with 5 case rules similar to those in Figure 1. For each run, the table records the number

of constraints produced as well as the time in seconds for constraint generation and constraint solving.

Several pointer dereferences fail to satisfy the `restrict` clause for `nonnull`, causing qualifier inference to signal inconsistencies. We analyzed each of the signaled errors for `bc` and inserted casts to `nonnull` where appropriate to allow inference to succeed. In total, we found no real errors and inserted 107 casts. Of these, 98 were necessary due to a lack of flow-sensitivity in our type system. We plan to explore the incorporation of targeted forms of flow-sensitivity to handle commonly arising situations. Despite this limitation, the qualifier rules were often powerful enough to deduce interesting invariants. For example, on `bc`, 37% (163/446) of the integer lvalues were able to be given the `nonzero` qualifier and 5% (24/446) the `pos` qualifier. For `snort`, 8% (561/7103) of its integer lvalues were able to be given the `nonzero` qualifier, and 7% (317/4571) of its pointer lvalues were able to be given the `nonnull` qualifier (without casts).

5.2 Rule Inference

We implemented rule inference in the context of CLARITY and performed two experiments. First, we inferred rules for `pos`, `neg`, and `nonzero`, given only their invariants. In the second experiment, we additionally inferred rules for `withinThree`. For the first experiment, our rule inference algorithm automatically generated all of the `case` rules we had originally hand-written for the three qualifiers. In addition, rule inference generated several valid rules that we had not written. For example, one new rule allows the negation of a `nonzero` expression to also be `nonzero`. The second experiment produced no new rules for `nonzero`, `pos`, and `neg`, indicating their orthogonality to the `withinThree` qualifier. However, it did generate several nontrivial rules for `withinThree` that we had not foreseen. For example, one rule allows a sum to be considered `withinThree` if one operand is `withinThree` and `pos` while the other operand is `withinThree` and `neg`. In both experiments, no redundant rules were generated.

The first experiment required 18 queries to the decision procedures in order to compute the $\preceq_Q$ and $\perp_Q$ relations, for use in the overall $\preceq$ partial order, and 142 queries to validate candidate rules. In contrast, the naive generate-and-test algorithm would require 600 queries. The second experiment required 32 queries to compute $\preceq_Q$ and $\perp_Q$ as well as 715 queries for candidate validation, while the naive algorithm would require 3136 queries. The first experiment completed in under six minutes, and the second experiment in under 26 minutes. The running times are quite reasonable, considering that rule inference need only be performed once for a given set of qualifiers, independent of the number of programs that employ these qualifiers.

6 Related Work

Our framework is most closely related to the CQUAL system, which also allows users to define new type qualifiers for C programs [6]. The main novelty in our approach is the incorporation of user-defined qualifier rules, which are not supported in CQUAL. Our qualifier inference algorithm extends the technique used for inference in CQUAL [6] to handle such user-defined rules via a form of conditional constraints. Our notion of

rule inference has no analogue in CQUAL. CQUAL includes a form of qualifier polymorphism, and follow-on work extended CQUAL's type system to be flow sensitive [7], while CLARITY currently lacks both of these features.

Work on *refinement types* [8] allows programmers to create subtypes of ML datatype definitions. Intersection types allow a function to have multiple type signatures with varying refinements, playing a role analogous to our `case` rules. A refinement inference algorithm is provided for a functional subset of ML. Later work [4] considered the interaction of intersection types with computational effects, and recent work extends these ideas to a flow-sensitive setting [10]. These two systems are more powerful than our type qualifiers, but they do not support full type inference.

HM(X) [11] is a Hindley-Milner-style type inference system that is parameterized by the form of constraints. Our situation is dual to that one: while HM(X) has a fixed type system that is parameterized by a constraint system, qualifier inference in our framework uses a fixed form of constraints but is parameterized by the qualifier rules.

Rule inference is related to work on *predicate abstraction* [9, 1] and on finding the *best transformer* [12, 13]. These algorithms use decision procedures to precisely abstract a program with respect to a set of predicates. Rule inference is similar, as it produces an abstraction automatically from the user-defined invariants. However, this abstraction is produced *once*, independent of any particular program.

7 Conclusions

We have described two forms of inference that reduce the burden on users of our approach to user-defined type qualifiers. *Qualifier inference* employs user-defined rules to infer qualifiers on programs, obviating the need for manual program annotations. We described an algorithm for qualifier inference based on generating and solving a system of conditional set constraints. *Rule inference* employs decision procedures to automatically produce qualifier rules that respect a qualifier's user-defined invariant, reducing the burden on qualifier designers. We described a partial order on candidate qualifier rules that allows us to search the space of candidates efficiently without losing completeness. We have implemented both qualifier and rule inference in the CLARITY system for C, and our experimental results illustrate their utility in practice.

Acknowledgments

This research was supported in part by NSF ITR award #0427202 and by a generous gift from Microsoft Research. Thanks to Craig Chambers, Vass Litvinov, Scott Smith, and Frank Tip for discussions that led to a simplification of the presentation of the constraint system. Thanks to Rupak Majumdar for useful feedback on the paper.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213. ACM Press, 2001.

2. B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–95, New York, NY, USA, 2005. ACM Press.
3. B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. Technical Report CSD-TR-050041, UCLA Computer Science Department, October 2005.
4. R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP '00: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 198–208. ACM Press, 2000.
5. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
6. 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.
7. 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.
8. T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 268–277, New York, NY, USA, 1991. ACM Press.
9. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
10. 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.
11. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.
12. T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 252–266, 2004.
13. G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 530–545, 2004.