Flow-insensitive type qualifier inference on programming languages allowing type casts

Huisong Li[1, 2*]

[1]*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, 100190, China*

[2]*University of Chinese Academy of Sciences, Beijing 100049, China*

Received 1 January 2014, www.tsi.lv

Abstract

Type qualifiers are lightweight specifications of atomic properties that refine the standard types. Flow-insensitive type qualifier inference has been used in the CQual framework to improve the quality of C programs. However, type casts will affect the effectiveness of type qualifier inference, as they can lead to either accepting some flawed programs due to discarding some useful qualifier information, or rejecting some safe programs when the analysis is conservative. In this paper, we first present a language, which allows type casts and formalize its flow-insensitive qualifier inference system. We then show some examples to illustrate how qualifiers are lost because of type casts in CQual and give an idea on solving this problem.

Keywords: Type cast, Type inference, Flow-insensitive type

1 Introduction

Type system is one of the popular formal methods to express general correctness properties of programs [6]. Type systems can be considered as tools for reasoning about programs. For example, they can statically check whether the arguments of primitive arithmetic operations are always integers. However, type system cannot capture arbitrary program behaviour. They can only guarantee that a well-typed program is free from certain kinds of bad program behaviours. For instance, type systems cannot check that the second argument of the division operation is non-zero. Nevertheless, several methods have been investigated to extend the expressivity of type systems: generalized algebraic data types [19], dependent types [20, 21], refinement types [2, 12], type qualifiers [8] and so on, among which, only type qualifiers are lightweight.

Type qualifiers are lightweight specifications for specifying atomic properties that refine standard types. Generally, qualifiers fall into two classes; value qualifiers, such as **tainted** and **untainted**, and reference qualifiers, such as **const** and **nonconst**. Value qualifiers pertain only to the value of an expression and reference qualifiers pertain to the address of an expression [5].

Specifically, **tainted** and **untainted** are qualifiers specified for checking the format-string vulnerabilities of C programs [16]. Usually, data read from the environment which could be controlled by untrusted users, like the network users, should be annotated with **tainted**. Printing functions, like *printf*, require the first argument to be **untainted** as the format specifier. It is safe to interpret **untainted** data as **tainted** data, but not vice versa, which can be represented as **untainted** $\leq$ **tainted**. Type qualifiers **const** and **nonconst** are used in **const** inference [10]. Usually, a **nonconst** cell can be reassigned, but a **const** cell cannot be reassigned, so it is safe to interpret a nonconst cell as a const cell, but not vice versa, which can be represented as **nonconst** $\leq$ **const**. In [10], Foster proposed a framework using type qualifiers to improve the quality of programs. In his framework, programmers are allowed to specify qualifiers and add qualifier annotations to the programs. Type qualifier inference, including flow-insensitive inference and flow-sensitive inference, will determine the remaining qualifiers and check the consistency of the qualifier annotations. In flow-insensitive inference system, a value's qualified type is the same everywhere, but always changes with the assignment in a flow-sensitive inference system. In both systems, however, the underlying types keep unchanged. Based on this framework, CQual was developed for C programs and has been used in a number of different practical applications [1, 11, 14, 16, 18].

Example 1 Example Program fragment in C:

```
const int* x;
int* y;
int a, b;
a = (int)x;          /*(1)*/
b = a;               /*(2)*/
y = (int*)b;         /*(3)*/
*y = 5;              /*(4)*/
```

However, type casts will affect the effectiveness of type qualifier inference, thus leading to missing some

* Corresponding author - E-mail:lihs@ios.ac.cn

flaws of programs. For the flow-insensitive inference, consider the example program fragment in Example 1, which can be accepted by CQual. In the program, *x* is a pointer to **const**. From the lines (1), (2), (3), *x* and *y* have the same value, which means *y* should be inferred as a pointer to **const** and the assignment should not be allowed at line (4). However, the type information that *x* is a pointer to **const** cannot be transferred to *y* by *a* and *b* because their type int cannot "contain" enough qualifier information as int*, which leads to losing some useful information during the qualifier inference process. However, note that at line (1), *x* is assigned to *a* after the type cast, which means the value of *a* is actually of type int*. The value of *b* is also of type int* as *b* is assigned to the value of *a* at line (2). Then, in the qualifier inference process, we can use int* instead of int for *a* and *b*. By doing so, we get that *y* is a pointer to **const**, that is, int* is more suitable for the qualifier inference process than int for *a* and *b*. Thus, for the qualifier inference process, we believe that the most suitable type of each expression may not be the type offered in the program.

The rest of this paper is organized as follows: Section 2 shows our source language and some preliminaries; Section 3 presents a flow-insensitive qualifier inference system for the source language; Section 4 show more examples about losing qualifier information because of type casts in CQual and a simple idea on how to solve this problem. Section 5 presents the related work.

2 Preliminaries

In the paper, we present the basic theory using a simple call-by-value source language. Prior to introducing expressions, we first present the definitions of types and qualified types respectively. A type t is a term generated by the following grammar:

$$t ::= \text{int} \mid \text{ref}(t) \mid t \to t \quad ,$$

where ref is a pointer type constructor and $\to$ is a function type constructor. Given a set of qualifiers Q, a qualified type τ is a term generated by:

$$\tau ::= q_c \sigma \quad q_c \in Q$$
$$\sigma ::= \text{int} \mid \text{ref}(\tau) \mid \tau \to \tau \quad .$$

Definition 1 gives the definition of the expressions of the simple language, where $(t)e$ is a type cast expression that casts the type of expression e to type t and $\text{annot}(e, q_c)$ is a qualifier annotation expression, which specifies q_c as the outermost qualifier of the qualified type of e. Generally, a set of qualifiers Q and their order $\le$ form a partial order set. For simplicity, the partial order: $(Q, \le)=\{\text{untainted} \le \text{tainted}\}$ is used for the examples of the rest paper.

Example 2 Main Example Program:

let $x =$ ref annot$(0, \textbf{tainted})$ in
let $y =$ $(\text{int})x$ in
let $z =$ $(\text{ref}(\text{int}))y$ in
annot$(*z, \textbf{untainted})$

- *Definition 1* Source language with type casts and qualifier annotations:

$e ::=$	v	value
	$\mid e_1 e_2$	application
	$\mid \text{let } x = e_1 \text{ in } e_2$	bind
	$\mid \text{ref } e$	reference
	$\mid *e$	deference
	$\mid (t)e$	type cast
	$\mid e_1 := e_2$	assignment
	$\mid \text{annot}(e, q_c)$	annotation
$v ::=$	n	integer
	$\mid x$	variable
	$\mid \lambda x : t.e$	function

Example 2 shows a program, which is written in our simple language and shows type casts between pointer type ref(int) and int. This program is not safe as **x* and **z* refer to the same object, which means we interpret a **tainted** object as an **untainted** object. Therefore, we expect to find this flaw by the qualifier inference system.

The partial order on type qualifiers can induce the qualified subtype relation $\le$ (In this paper, we use $\le$ for both the order between qualifiers and qualified types.) among qualified types in Definition 2 [10]. $t_1 \rhd t_2$ means it is entirely safe to interpret an object of type τ_1 as an object of type τ_2. In order to ensure that all aliases of the same ref cell contain the same qualifiers, the second rule, $(\text{Ref}_{\le})$, requires that $\tau_1 = \tau_2$ ($\tau_1 \le \tau_2$ and $\tau_2 \le \tau_1$) instead of $\tau_1 \le \tau_2$. Otherwise, there will be problems such as assigning a pointer of type ref(**untainted** int) to pointer of type ref(**tainted** int) where we can write **tainted** data to **untainted** position through the ref(**tainted** int) pointer. In the rule, $\text{strip}_q(\tau_1) \rhd \text{strip}_q(\tau_2)$, functions are contravariant in their domain and covariant in their range. In Definition 3, embed_q is a function for mapping standard types to qualified types with fresh qualifier variables. Qualifier variables (written as *q*) stand for unknown qualifiers. strip_q is a function mapping qualified types to their standard types.

The type casts allowed in our source language are showed in Definition 4. If the cast from t_1 to t_2 is allowed, we say t_1 is the castable type of t_2, denoted as $t_1 \rhd t_2$, meaning that a value of type t_1 can be interpreted

as a value of type t_2. $(\mathrm{IR}_{\triangleright})$ and $(\mathrm{RI}_{\triangleright})$ show all type casts between the pointer type and type int are allowed as an integer can be interpreted as an address and vice versa. $(\mathrm{RR}_{\triangleright})$ shows all type casts between pointer types are allowed as an address of one type can be interpreted as an address of another type. For simplicity, we forbid all the other type casts, like type casts between function types.

Based on the castable type relation, we define the following qualified castable type relation. We first give the rules of the relation $\triangleright^q$ in Definition 5, in which $\tau_1 \triangleright^q \tau_2$ means their "corresponding" qualifiers satisfy the partial order $\leq$ of $(Q, \leq)$. Rules, $(\mathrm{RR}_{\triangleright^q})$ and $(\mathrm{FF}_{\triangleright^q})$, are similar to the rules of the qualified subtype relation. All the other rules only have the outermost qualifiers as the "corresponding" qualifiers. Note that the rules in Definition 5 are general formalizations for value qualifiers, like **tainted** and **untainted**. For the other qualifier kinds, the rules can be adjusted according to the properties of qualifiers and our analysis requirements. If $\mathrm{strip}_q(\tau_1) \triangleright \mathrm{strip}_q(\tau_2)$ (the standard type of τ_1 is the castable type of the standard type of τ_2) and $\tau_1 \triangleright^q \tau_2$, then τ_1 is the qualified castable type of τ_2, meaning that interpreting a value of type τ_1 as a value of type τ_2 is partially safe. The reason is that τ_1 is the qualified castable type of τ_2 and τ_2 is the qualified castable type of τ_3 does not guarantee that τ_1 is the qualified castable type of τ_3.

- *Definition 2* Qualified subtype relation:

$$\frac{q_1 \leq q_2}{q_1 \text{ int} \leq q_2 \text{ int}} \quad (\mathrm{Int}_{\leq})$$

$$\frac{q_1 \leq q_2 \quad \tau_1 = \tau_2}{q_1 \text{ ref}(\tau_1) \leq q_2 \text{ ref}(\tau_2)} \quad (\mathrm{Ref}_{\leq})$$

$$\frac{q_1 \leq q_2 \quad \tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4}{q_1 (\tau_1 \to \tau_2) \leq q_2 (\tau_3 \to \tau_4)} \quad (\mathrm{Fun}_{\leq})$$

- *Definition 3* embed_q and strip_q:

$\mathrm{embed}_q(\mathrm{int}) = q \text{ int} \quad q$ fresh
$\mathrm{embed}_q(\mathrm{ref}(t)) = q \text{ ref}(\mathrm{embed}_q(t)) \quad q$ fresh
$\mathrm{embed}_q(t_1 \to t_2) =$
$q (\mathrm{embed}_q(t_1) \to \mathrm{embed}_q(t_2)) \quad q$ fresh
$\mathrm{strip}_q(q \text{ int}) = \text{ int}$
$\mathrm{strip}_q(q \text{ ref}(\tau)) = \text{ ref}(\mathrm{strip}_q(\tau))$
$\mathrm{strip}_q(q (\tau_1 \to \tau_2)) = \mathrm{strip}_q(\tau_1) \to \mathrm{strip}_q(\tau_2)$

- *Definition 4* Castable-type relation:

$\text{int} \triangleright \text{int} \quad (\mathrm{II}_{\triangleright})$
$\text{int} \triangleright \text{ref}(t) \quad (\mathrm{IR}_{\triangleright})$
$\text{ref}(t) \triangleright \text{int} \quad (\mathrm{RI}_{\triangleright})$
$\text{ref}(t_1) \triangleright \text{ref}(t_2) \quad (\mathrm{RR}_{\triangleright})$

- *Definition 5* Qualified castable-type relation:

$$\frac{q_1 \leq q_2}{q_1 \text{ int} \triangleright^q q_2 \text{ int}} \quad (\mathrm{II}_{\triangleright^q})$$

$$\frac{q_1 \leq q_2}{q_1 \text{ int} \triangleright^q q_2 \text{ ref}(\tau)} \quad (\mathrm{IR}_{\triangleright^q})$$

$$\frac{q_1 \leq q_2}{q_1 \text{ ref}(\tau) \triangleright^q q_2 \text{ int}} \quad (\mathrm{RI}_{\triangleright^q})$$

$$\frac{q_1 \leq q_2}{q_1 \text{ int} \triangleright^q q_2 (\tau_1 \to \tau_2)} \quad (\mathrm{IF}_{\triangleright^q})$$

$$\frac{q_1 \leq q_2}{q_1 (\tau_1 \to \tau_2) \triangleright^q q_2 \text{ int}} \quad (\mathrm{FI}_{\triangleright^q})$$

$$\frac{q_1 \leq q_2}{q_1 \text{ ref}(\tau_1) \triangleright^q q_2 (\tau_2 \to \tau_3)} \quad (\mathrm{RF}_{\triangleright^q})$$

$$\frac{q_1 \leq q_2}{q_1 (\tau_1 \to \tau_2) \triangleright^q q_2 \text{ ref}(\tau_3)} \quad (\mathrm{FR}_{\triangleright^q})$$

$$\frac{q_1 \leq q_2 \quad \tau_1 \triangleright^q \tau_2 \quad \tau_2 \triangleright^q \tau_1}{q_1 \text{ ref}(\tau_1) \triangleright^q q_2 \text{ ref}(\tau_2)} \quad (\mathrm{RR}_{\triangleright^q})$$

$$\frac{q_1 \leq q_2 \quad \tau_3 \triangleright^q \tau_1 \quad \tau_2 \triangleright^q \tau_4}{q_1 (\tau_1 \to \tau_2) \triangleright^q q_2 (\tau_3 \to \tau_4)} \quad (\mathrm{FF}_{\triangleright^q})$$

3 The Qualifier Inference System

For the source language, the standard type checking system is presented in Definition 6. Judgements of the form $\Gamma \vdash e : t$ mean that in the type environment Γ which maps variables to their types, expression *e* has type *t*. (Cast) computes the type *t'* of *e*, checks that *t'* is the castable type of *t*, and then *t* is the type of the **cast** expression. (Annot) ignores the annotation of the qualifier as standard types do not contain qualifiers. The type *t* of e is the type of the **Annot** expression. The discussion of the other rules can be found in [10]. Note that the standard static type checking system cannot guarantee that programs run without runtime errors because of type casts, which is a well-known limitation of type systems.

The qualifier inference system in Definition 7 is used to infer qualifiers automatically under the assumption that the program passes the standard type checking system. For this system, judgements of the form $\Gamma_q \vdash_q e : \tau$ mean that in qualified type environment Γ_q, which maps variables to their qualified types, expression *e* has qualified type τ. Whenever we assign a type to a term constructor, we introduce a fresh qualifier variable to stand for the unknown qualifier on the term that we need to solve for (see (Int_q), (Lam_q) and (Ref_q)). We use embed_q in (Lam_q) and (Cast_q) to map the given standard type to a qualified type with fresh qualifier variables. Rules (App_q) and (Assign_q) generate the qualified subtype constraints of the form $\tau_1 \leq \tau_2$. (Cast_q) generates the qualified castable type constraints of the form $\tau_1 \triangleright^q \tau_2$. (Annot_q) computes the qualified type $q\,\sigma$ of *e* and generates the constraints

$q = q_c$ ($q \le q_c$ and $q_c \le q$) in order to infer that the value of q is q_c.

After we perform the qualifier inference system on the program, we will get a set C_τ containing constraints of the forms $\tau_1 \le \tau_2$, $\tau_1 \rhd^q \tau_2$ and a set C_q containing constraints of the form $q = q_c$. Then, we reduce all constraints of C_τ into C_q using $R(\tau,\ \tau')$ in Definition 8 which are simply the rules of Definition 5 rewritten left-to-right.

Formally,

$$C_q = C_q \cup \bigcup_{\forall \tau_1 \le \tau_2 \in C_\tau} R(\tau_1,\ \tau_2) \cup \bigcup_{\forall \tau_1 \rhd^q \tau_2 \in C_\tau} R(\tau_1,\ \tau_2).$$

The algorithm for solving C_q to get the value of qualifier variables can be found in [10]. Note that as we assume the program is correct with respect to the standard type checking system, then for any $\tau_1 \le \tau_2 \in C_\tau$, $\text{strip}_q(\tau_1) = \text{strip}_q(\tau_2)$ (syntactic equality) and for any $\tau_1 \rhd^q \tau_2 \in C_\tau$, $\text{strip}_q(\tau_1) \rhd \text{strip}_q(\tau_2)$.

Example 3 The inference result of the Example 2:

$$\Gamma_q: \left\{ \begin{array}{l} (x,\ q_2\ \text{ref}\ (q_1\ \text{int})),\ (y,\ q_3\ \text{int}), \\ (z,\ q_5\ \text{ref}\ (q_4\ \text{int})) \end{array} \right\}$$

$$C_\tau: \left\{ \begin{array}{l} q_2\ \text{ref}\ (q_1\ \text{int}) \rhd^q q_3\ \text{int}, \\ q_3\ \text{int} \rhd^q q_5\ \text{ref}\ (q_4\ \text{int}) \end{array} \right\}$$

$$C_q: \{q_1 = \textbf{tainted},\ q_4 = \textbf{untainted}\}$$

Then, reducing C_τ into C_q will get that $C_q = \{q_1 = \textbf{tainted},\ q_4 = \textbf{untainted},\ q_2 \le q_3,\ q_3 \le q_5\}$. Solving C_q will get that the value of q_1 and q_4 are **tainted** and **untainted** respectively. However, as we have discussed in Example 2, this program is not safe by interpretting a **tainted** object as an **untainted** object. But this system does not find the flaw because the qualifier information of q_1 cannot be transferred to q_4. That is, some qualifier information is lost during the inference process.

- *Definition 6* The standard type checking system:

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(Var)}$$

$$\frac{}{\Gamma \vdash \text{n} : \text{int}} \quad \text{(Int)}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash \lambda x : t.e : t \to t'} \quad \text{(Lam)}$$

$$\frac{\Gamma \vdash e_1 : t \to t'\ \ \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : t'} \quad \text{(App)}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash ref\,(e) : ref\,(t)} \quad \text{(Ref)}$$

$$\frac{\Gamma \vdash e : ref\,(t)}{\Gamma \vdash *e : t} \quad \text{(Deref)}$$

$$\frac{\Gamma \vdash e_1 : t\ \ \Gamma[x \mapsto t] \vdash e_2 : t'}{\Gamma \vdash \text{let}\ x = e_1\ \text{in}\ e_2 : t'} \quad \text{(Let)}$$

$$\frac{\Gamma \vdash e_1 : ref\,(t)\ \ \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 := e_2 : t} \quad \text{(Assign)}$$

$$\frac{\Gamma \vdash e : t'\ \ t' \rhd t}{\Gamma \vdash (t)e : t} \quad \text{(Cast)}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash annot(e,\ \text{q}_\text{c}) : t} \quad \text{(Annot)}$$

4 Examples

Based on the inference of **tainted** and **untainted**, we list some examples in Table 1 to specifically show how CQual keep track of qualifiers in the presence type casts. For simplicity, we only show the source qualified types in the first column, the destination qualified types in the second column and the generated constraints for transferring the qualifier information of the source to the destination in the last column. The second line shows casting from q_1 ref(q_2 int) to q_3 int. The constraint $q_1 \le q_3$ transfers the qualifier information of q_1 to q_3. Nevertheless, the qualifier information of q_2 is discarded. The forth line shows casting between pointers. From the qualified castable type relation, $q_2 = q_4$ ($q_2 \le q_4$ and $q_4 \le q_2$) is necessary. The last two lines show casting related to function pointers. We believe the constraints are some conservative, for example in the last line $q_2 = q_7$ will transfer the qualifier information of q_2 to q_7. To conclude, we can get that there are always some qualifier information which may be useful been discarded. On the other side, some constraints on qualifiers are some conservative. These will affect the effectiveness of qualifier inference, thus leading to miss some flawed programs or produce some false positives.

Consider the second line of Table 1 again, if we augment q_3 int to q_3 ref(q_4 int) in which q_4 is a fresh

qualifier variable, we can preserve the qualifier information of q_2 to q_4 instead of discarding it directly. Note that the augmentation is only for the qualifier inference. However, for the type cast in the last line of Table 1, in order to preserve the qualifier information of q_3 and q_4, we have to augment the destination type to contain a function type. The only way is introducing union types. We expect the augmentation result is $q_5\ \mathrm{ref}(q_6\ (\mathrm{ref}(q_7\ \mathrm{int}) \vee (q_8\ \mathrm{int} \to q_9\ \mathrm{int})))$, in which q_8 and q_9 are fresh qualifier variables. Then we can preserve the qualifier information of q_3 and q_4 to q_8 and q_9 respectively. We expect to do this in the future work.

5 Related work and conclusion

Type qualifiers can be seen as a kind of refinement types [2, 12], which do not change the underlying type structure and extend the expressivity of type systems. The type refinement framework in [15] supports a flow-sensitive, sophisticated type system. The theory of type qualifiers proposed in [10] describes a framework for adding type qualifiers to a language and show a flow-insensitive type qualifier inference system. Flow-sensitive type qualifier inference systems were proposed in [9], in which only type qualifiers were modelled flow-sensitively. In [5], a framework for allowing users to explicitly write type rules for their new type qualifiers and to explicitly specify the run-time invariant that the type qualifiers meant to represent was proposed, but it was not flexible enough, only supporting certain kinds of qualifiers. Based on the theory of type qualifiers, CQual [7] was developed for C programs and has had many applications, like const qualifiers inference[10], finding format-string vulnerabilities [16], static analysis of authorization hook placement [11, 18], finding user/kernel bugs [10], and deadlocks in Linux kernels [1]. Later, JQual [13] was developed for adding user-defined type qualifiers to Java.

Program analysis based on type qualifiers is a kind of static analysis techniques for improving software quality. However, as C standard allows arbitrary type casts between pointer types and some other type casts [17], we will lose some useful qualifier information during the qualifier inference process. Our work aims to solve this problem in the flow-insensitive qualifier inference process. Therefore, it can be seen as an improvement of flow-insensitive qualifier inference.

In this paper, we formalized a flow-insensitive qualifier inference system for a source language allowing type casts, and showed the problem of losing qualifier information caused by type casts in CQual and proposed a simple idea to solve this problem.

- *Definition 7* Qualifier inference system:

$$\frac{x \in dom(\Gamma_q)}{\Gamma_q \hbar_q\ x : \Gamma_q(x)} \quad (\mathrm{Var}_q)$$

$$\frac{q\ \text{ fresh}}{\Gamma_q \hbar_q\ \mathrm{n} : q\ \mathrm{int}} \quad (\mathrm{Int}_q)$$

$$\frac{\tau = embed_q(t) \quad \Gamma_q[x \mapsto \tau] \hbar_q\ e : \tau'}{\Gamma_q \hbar_q\ \lambda x : t.e : q\ (\tau \to \tau')} \quad (\mathrm{Lam}_q)$$

$$\frac{\Gamma_q \hbar_q\ e_1 : q\ (\tau_1 \to \tau_2) \quad \Gamma_q \hbar_q\ e_2 : \tau_3 \quad \tau_3 \le \tau_1}{\Gamma_q \hbar_q\ e_1 e_2 : \tau_2} \quad (\mathrm{App}_q)$$

$$\frac{\Gamma_q \hbar_q\ e : \tau \quad q\ \text{fresh}}{\Gamma_q \hbar_q\ ref\,(e) : q\ ref\,(\tau)} \quad (\mathrm{Ref}_q)$$

$$\frac{\Gamma_q \hbar_q\ e : q\ ref\,(\tau)}{\Gamma_q \hbar_q\ *e : \tau} \quad (\mathrm{Deref}_q)$$

$$\frac{\Gamma_q \hbar_q\ e_1 : \tau \quad \Gamma_q[x \mapsto \tau] \hbar\ e_2 : \tau'}{\Gamma_q \hbar_q\ \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : \tau'} \quad (\mathrm{Let}_q)$$

$$\frac{\Gamma_q \hbar_q\ e_1 : q\ ref\,(\tau) \quad \Gamma_q \hbar_q\ e_2 : \tau' \quad \tau' \le \tau}{\Gamma_q \hbar_q\ e_1 := e_2 : \tau'} \quad (\mathrm{Assign}_q)$$

$$\frac{\Gamma_q \hbar_q\ e : \tau \quad \tau' = embed_q(t) \quad \tau \rhd^q \tau'}{\Gamma_q \hbar_q\ (t)e : \tau'} \quad (\mathrm{Cast}_q)$$

$$\frac{\Gamma_q \hbar_q\ e : q\ \sigma \quad q = \mathrm{q_c}}{\Gamma_q \hbar_q\ annot(e, \mathrm{q_c}) : q\ \sigma} \quad (\mathrm{Annot}_q)$$

- *Definition 8* Constraint resolution rules:

$$R(q_1\ \mathrm{int},\ q_2\ \mathrm{int}) = \{q_1 \le q_2\}$$
$$R(q_1\ \mathrm{int},\ q_2(\tau_1 \to \tau_2)) = \{q_1 \le q_2\}$$
$$R(q_1\ \mathrm{int},\ q_2\ \mathrm{ref}\,(\tau)) = \{q_1 \le q_2\}$$
$$R(q_1(\tau_1 \to \tau_2),\ q_2\ \mathrm{int}) = \{q_1 \le q_2\}$$
$$R(q_1\ \mathrm{ref}\,(\tau),\ q_2\ \mathrm{int}) = \{q_1 \le q_2\}$$
$$R(q_1\ \mathrm{ref}\,(\tau_1),\ q_2(\tau_2 \to \tau_3)) = \{q_1 \le q_2\}$$
$$R(q_1\ \mathrm{ref}\,(\tau_1),\ q_2\ \mathrm{ref}\,(\tau_2)) = \{q_1 \le q_2\} \cup R(\tau_1, \tau_2) \cup R(\tau_2, \tau_1)$$
$$R(q_1(\tau_1 \to \tau_2),\ q_2\ \mathrm{ref}\,(\tau_3)) = \{q_1 \le q_2\}$$
$$R(q_1(\tau_1 \to \tau_2),\ q_2(\tau_3 \to \tau_4)) = \{q_1 \le q_2\} \cup R(\tau_3, \tau_1) \cup R(\tau_2, \tau_4)$$

References

[1] Aiken A, Foster J S, Kodumal J, Terauchi T 2003 *ACM SIGPLAN Notices* **38**(5) 129–40
[2] Bierman G M, Gordon A D, Hriţcu C, Langworthy D 2010 *ACM SIGPLAN Notices* **45**(9) 105–16
[3] Castagna G, Xu Zhiwu. 2011 *ACM SIGPLAN Notices* **46**(9) 94-106
[4] Chandra S, Reps T 1999 *ACM SIGSOFT Software Engineering Notes* **24**(5) 66-75
[5] Chin B, Markstrum S, Millstein T 2005 *ACM SIGPLAN Notices* **40**(6) 85-95
[6] Pierce B C 2002 *Types and programming languages* MIT Press
[7] Foster J S 2001 CQual User's Guide. Berkeley: University of California
[8] Foster J S, Fähndrich M, Aiken A 1999 *ACM SIGPLAN Notices* **34**(5) 192-203
[9] Foster J S, Terauchi T, Aiken A 2002 *ACM SIGPLAN Notices* **37**(5) 1-12
[10] Foster J S 2002 *Type qualifiers: lightweight specifications to improve software quality* Berkeley: University of California

[11] Fraser T, Petroni Jr N L, Arbaugh W A 2006 Applying flow-sensitive CQUAL to verify MINIX authorization check placement *Proceedings of the workshop on Programming languages and analysis for security* ACM 3-6
[12] Freeman T, Pfenning F 1991 *ACM SIGPLAN Notices* **26**(**6**) 268–77
[13] Greenfieldboyce D, Foster J S 2007 *ACM SIGPLAN Notices* **42**(10) 321-36
[14] Johnson R, Wagner D 2004 Finding User/Kernel Bugs With Type Inference *Proceedings of the 13th Usenix Security Symposium, San Diego*, CA
[15] Mandelbaum Y, Walker D, Harper R 2003 *ACM SIGPLAN Notices* **38**(9) 213-25
[16] Shankar U, Talwar K, Foster J S, et al 2001 Detecting Format String Vulnerabilities with Type Qualifiers *USENIX Security Symposium* 201-20
[17] Siff M, Chandra S, Ball T, et al. 1999 Coping with type casts in C. *Software Engineering—ESEC/FSE'99*. Springer Berlin Heidelberg: 180-98
[18] Zhang Xiaolan, Edwards A, Jaeger T 2002 Using CQUAL for Static Analysis of Authorization Hook Placement. *USENIX Security Symposium* 33-48
[19] Xi Hongwei, Chiyan Chen, Gang Chen 2003 *ACM SIGPLAN Notices* **38**(1) 224-35
[20] Augustsson L 1999 Cayenne - a language with dependent types *Advanced Functional Programming* Berlin-Heidelberg: Springer 240-267
[21] Xi Hongwei, Pfenning F 1999 Dependent types in practical programming *POPL '99 Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* 214-227

Author

Huisong Li

University studies: master student in University of Chinese Academy of Sciences
Scientific interest: research interests are programming language and static analysis