Qualifier Type Inference

We present the full qualifier inference system in this section. Our system extends the flow-sensitive analysis of Foster *et al.* [1]. In particular, we consider pair types (and more generally records) and present their corresponding type inference rules. Providing separate qualifiers for the elements of pairs is important in our problem domain, as records (C `structs`) are used extensively in the Linux kernel. More importantly, pointers to records are often passed between functions and whether a field of a record is or is not initialized is independent of the other fields of the record. We present a type qualifier inference system to infer a qualifier (either $init$ or $uninit$) for each expression of the program.

A. Syntax

Our qualifier inference is performed after alias analysis. The alias analysis results are used to decorate aliased references with the same abstract locations ρ. This can be the line number of an object allocation statement. In the input programs, reference creation expressions are decorated with abstract locations and functions are decorated with effects (*i.e.,* the set of abstract locations that they access). The abstract syntax is defined as follows:

$$
\begin{aligned}
e &:= x \mid n \mid \lambda^L x{:}t.\ e \mid e_1\ e_2 \mid \mathsf{ref}^\rho\, e \mid\ !e \\
&\quad \mid e_1 := e_2 \mid \langle e_1, e_2\rangle \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
&\quad \mid \mathsf{fst}(e_1) := e_2 \mid \mathsf{snd}(e_1) := e_2 \mid \\
&\quad \mid \mathsf{assert}(e, Q) \mid \mathsf{check}(e, Q) \\
t &:= \alpha \mid int \mid ref(\rho) \mid t \to^L t' \mid \langle t_1, t_2\rangle \\
L &:= \{\rho, .., \rho\}
\end{aligned}
$$

An expression e can be a variable x, a constant integer n, a function $\lambda^L x{:}t.\ e$ with argument x of type t, effect set L and body e. The effect set L is the set of abstract locations ρ that the function accesses. A type t is either a type variable α, an integer type int, a reference $ref(\rho)$ (to the abstract location ρ), a function type $t \to^L t'$ (that is decorated with its effects L) or a pair type $\langle t_1, t_2\rangle$. The analysis will involve a store C that maps abstract locations ρ to types. The expression $e_1\ e_2$ is the application of function e_1 to argument e_2. The reference creation expression $\mathsf{ref}^\rho\, e$ (decorated with the abstract location ρ) allocates memory with the value e. The expression $!e$ dereferences the reference e. The expression $e_1 := e_2$ assigns the value of e_2 to the location e_1 points to. The expression $\langle e_1, e_2\rangle$ is the pair of e_1 and e_2. The expressions $\mathsf{fst}(e)$ and $\mathsf{snd}(e)$ are the first and second elements of the pair e respectively. The expressions $\mathsf{fst}(e_1) := e_2$ and $\mathsf{snd}(e_1) := e_2$ assign the value of e_2 to the first element and second elements of the location e_1 points to respectively.

We use *explicit* qualifiers to both annotate and check the initialization status of expressions. The expression $\mathsf{assert}(e, Q)$ annotates the expression e with the qualifier Q. The expression $\mathsf{check}(e, Q)$ requires the top-level qualifier of e to be at most Q. We automatically insert the check expressions through a simple program transformation. Specifically, we consider two types of *use* as security critical: pointer dereferences and conditional branches. To detect UBI, we insert a $\mathsf{check}(e, init)$ statement before every statement where e is dereferenced or is used as the predicate of a conditional branch.

B. Types and Type Stores

We now define the qualified types.

$$
\begin{aligned}
\tau &:= Q\ \sigma \\
Q &:= \kappa \mid init \mid uninit \\
\sigma &:= int \mid ref(\rho) \mid (C, \tau) \to (C', \tau') \mid \langle \tau_1, \tau_2\rangle \\
C &:= \epsilon \mid Alloc(C, \rho) \mid Assign(C, \rho{:}\tau) \\
&\quad \mid Merge(C, C', L) \mid Filter(C, L) \\
\eta &:= 0 \mid 1 \mid \omega
\end{aligned}
$$

The qualified types τ can have qualifiers at different levels. Q can be a qualifier variable κ or a constant qualifier $init$ or $uninit$. The flow-sensitive analysis associates a ground store C to each program point that is a vector that associates abstract locations to qualified types. Thus, function types are now extended to $(C, \tau) \to (C', \tau')$ where C is the store that the function is invoked in and C' is the store when the function returns.

Each location in a store C also has an associated linearity η that can take three values: 0 for unallocated locations, 1 for linear locations, and ω for non-linear locations. An abstract location is *linear* if the type system can prove that it corresponds to a single concrete location in every execution. An update that changes the qualifier of a location is called a strong update; otherwise, it is called a weak update. Strong updates can be applied to only linear locations. The three linearities form a lattice $0 < 1 < \omega$. Addition on linearities is as follows: $0 + x = x$, $1 + 1 = \omega$, and $\omega + x = \omega$. The type inference system tracks the linearity of locations to allow strong updates for only the linear locations.

Since a store C maps from each abstract location ρ_i to a type τ_i and a linearity η_i, we write $C(\rho)$ as the type of ρ in C and $C_{lin}(\rho)$ as the linearity of ρ in C. Store variables are denoted as ϵ. We use the following store constructors to represent the store after an expression as a function of the store before it. $Alloc(C, \rho)$ returns the same store as C except for the location ρ. Allocating ρ does not affect the types in the store; however, as ρ is allocated once more, the linearity of ρ is increased by one. $Merge(C, C', L)$ returns the combination of stores C and C'; for a location ρ, if $\rho \in L$, then its type and linearity are taken from C, otherwise from C'. $Filter(C, L)$

$$
\begin{array}{ll}
Alloc(C,\rho')(\rho) & = C(\rho) \\
Alloc(C,\rho')_{lin}(\rho) & = \begin{cases} 1 + C_{lin}(\rho) & \text{if } \rho = \rho' \\ C(\rho) & \text{otherwise} \end{cases} \\
Merge(C,C',L)(\rho) & = \begin{cases} C(\rho) & \text{if } \rho \in L \\ C(\rho') & \text{otherwise} \end{cases} \\
Merge(C,C',L)_{lin}(\rho) & = \begin{cases} C_{lin}(\rho) & \text{if } \rho \in L \\ C'_{lin}(\rho) & \text{otherwise} \end{cases} \\
Filter(C,L)(\rho) & = C(\rho) \quad \rho \in L \\
Filter(C,L)_{lin}(\rho) & = \begin{cases} C_{lin}(\rho) & \text{if } \rho \in L \\ 0 & \text{otherwise} \end{cases} \\
Assign(C,\rho' \colon \tau)(\rho) & = \begin{cases} \tau' \text{ where } \tau \preceq \tau' & \text{if } \rho = \rho' \wedge C_{lin}(\rho) \neq \omega \\ \tau \sqcup C(\rho) & \text{if } \rho = \rho' \wedge C_{lin}(\rho) = \omega \\ C(\rho) & \text{otherwise} \end{cases} \\
Assign(C,\rho' : \tau)_{lin}(\rho) & = C_{lin}(\rho)
\end{array}
$$

restricts the domain of C to L. $Assign(C, \rho\colon\tau)$ overrides C by mapping ρ to a type τ' such that $\tau \preceq \tau'$. The condition $\tau \preceq \tau'$ allows assigning a subtype τ of resulting type τ' to ρ. If ρ is linear then its type in $Assign(C, \rho : \tau)$ is τ'; otherwise its type is conservatively the least-upper bound of τ and its previous type $C(\rho)$.

The type inference system generates subtyping constraints between stores. We define store subtyping in Figure 1.

$$
\textsc{Int}_{\preceq}\ \frac{Q \preceq Q'}{Q\ int \preceq Q'\ int} \qquad \textsc{Ref}_{\preceq}\ \frac{Q \preceq Q'}{Q\ ref(\rho) \preceq Q'\ ref(\rho)}
$$

$$
\textsc{Fun}_{\preceq}\ \frac{Q \preceq Q' \qquad \tau_2 \preceq \tau_1 \qquad \tau_1' \preceq \tau_2' \qquad C_2 \preceq C_1 \qquad C_1' \preceq C_2'}{Q\ (C_1,\tau_1) \to^L (C_1',\tau_1') \preceq Q'\ (C_2,\tau_2) \to^L (C_2',\tau_2')}
$$

$$
\textsc{Store}_{\preceq}\ \frac{\tau_i \preceq \tau_i' \qquad \eta_i \preceq \eta_i' \qquad i = 1..n}{\{\rho_1^{\eta_1} : \tau_1, ..., \rho_n^{\eta_n} : \tau_1\} \preceq \{\rho_1^{\eta_1'} : \tau_1', ..., \rho_1^{\eta_n'} : \tau_n'\}}
$$

$$
\textsc{Pair}_{\preceq}\ \frac{Q \preceq Q' \qquad \tau_1 \preceq \tau_1' \qquad \tau_1 \preceq \tau_2'}{Q\ \langle \tau_1, \tau_2 \rangle \preceq Q'\ \langle \tau_1', \tau_2' \rangle}
$$

Fig. 1: Store subtyping.

Constraints between stores yield constraints between linearities and types, which in turn yield constraints between qualifiers and linearities. The rule $\textsc{Int}_{\preceq}$ requires a corresponding subtyping relation for the qualifiers of the type int. The rule $\textsc{Ref}_{\preceq}$ requires the same subtyping relation between qualifiers and further, the equality of the two locations. The rule $\textsc{Fun}_{\preceq}$ requires the subtyping relation between the top-level qualifiers, and contra-variance for the argument and input store and co-variance for the return value and output store. The rule $\textsc{Store}_{\preceq}$ requires both subtyping and stronger linearity for corresponding locations. The rule $\textsc{Pair}_{\preceq}$ requires subtyping between the top-level qualifiers, and also subtyping for corresponding elements of the two pair type.

C. Type Inference System

We present the complete rules of the type inference system in Figure 2. The judgments are of the form $\Gamma, C \vdash e\colon\tau, C'$ that is read as in type environment Γ and store C, evaluating e yields a result of type τ and a new store C'. The rules VAR and INT are standard. The rule REF creates a location and adds it to the store. The type τ of the expression e that is stored in the new location is constrained to be a subtype of the type of ρ in the post-store. The qualifier of the new location is initialized. The rule DEREF checks that the dereferenced expression is of a reference type $ref(\rho)$ and retrieves the type of the value stored at the location ρ from the store. Qualifiers are checked by the single check expression described before (and not when references are dereferenced). The rule ASSIGN checks that the left-hand side expression is of a reference type and checks that the type of the right-hand side is a subtype of the type of the value that the reference stores. It also checks that the right-hand side can be assigned to the left-hand side considering the linearity and type of the left-hand side reference and the type of the right-hand side expression (as described in the definition of $Assign$ above). The rule LAM type-checks the function body e in a fresh initial store ϵ and with the parameter bound to a type with fresh qualifier variables. The resulting post-store of the function body C' should be a subtype of the post-store of the function ϵ'. This step essentially creates a function summary, which has been explained in the paper section 4.3. We use the function $sp(t)$ to decorate a standard type t with fresh qualifier and store variables:

$$
\begin{array}{rcll}
sp(\alpha) & = & \kappa\ \alpha & \kappa \text{ fresh} \\
sp(int) & = & \kappa\ int & \kappa \text{ fresh} \\
sp(ref(\rho)) & = & \kappa\ ref(\rho) & \kappa \text{ fresh} \\
sp(t \to^L t') & = & \kappa\ (\epsilon, sp(t)) \to^L (\epsilon', sp(t')) & \kappa, \epsilon, \epsilon' \text{ fresh} \\
sp(\langle t, t' \rangle) & = & \kappa\ \langle sp(t), sp(t') \rangle & \kappa \text{ fresh}
\end{array}
$$

The rule APP checks that the type of e_2 is a subtype of the parameter type of e_1, Further, with the condition $Filter(C, L) \preceq \epsilon$, it checks that state of the locations that e_1 uses (captured by its effect set L) in the post-store C'' of e_2 are compatible with the store ϵ that the function e_1 expects. The resulting store $Merge(\epsilon', C'', L)$ joins the store C'' before the function call with the result store ϵ' of the function. Filtering and merging according to the effect set provides polymorphism as functions do not affect the locations they do not use. The rule ASSERT adds a qualifier annotation to the program, and the rule CHECK checks that the top-level qualifier Q' of e is more specific or equal to the the expected qualifier Q.

The rule PAIR type-checks the expressions e_1 and e_2 in order and results in an initialized pair type. The rule FST checks that the expression e is of a pair type and types $\mathsf{fst}(e)$ as the first element of the pair type. The qualifier Q of the pair type is unconstrained; qualifiers are only checked by the check expressions presented above. The rule FSTASSIGN checks that the expression e_1 is of a reference type $ref(\rho)$, the post-store C'' (after checking e_1 and e_2) maps the reference ρ to a supertype of a pair type $\kappa\ \langle \alpha_1, \alpha_2 \rangle$, and the type τ_1 of e_2 is a subtype of α_1. The resulting store remaps ρ to a new pair type where the first element is the type of τ_1 and the second element is unchanged. More precisely, as described in the definition of $Assign$ above, the $Assign$ store updates ρ to the new pair type if ρ is linear; otherwise updates ρ to the least upper bound of the old and new pair types. We elide the rules for snd that are similar to the rules for fst. The constraints generated by the new rules PAIR, FST and FSTASSIGN are type and store subtyping constraints that the previous rules generated too.

$$\textsc{Var}\ \frac{x \in dom(\Gamma)}{\Gamma, C \vdash x : \Gamma(x), C} \qquad \textsc{Int}\ \frac{\kappa \quad fresh}{\Gamma, C \vdash n : \kappa\ int, C} \qquad \textsc{Ref}\ \frac{\Gamma, C \vdash e{:}\,\tau, C' \qquad \tau \preceq C'(\rho)}{\Gamma, C \vdash \mathsf{ref}^{\rho} e{:}\,\kappa\ ref(\rho),\ Alloc(C', \rho)} \qquad \textsc{Deref}\ \frac{\Gamma, C \vdash e : Q\ ref(\rho), C'}{\Gamma, C \vdash\ !e : C'(\rho), C'}$$

$$\textsc{Assign}\ \frac{\Gamma, C \vdash e_1{:}\,Q\ ref(\rho), C' \qquad \Gamma, C' \vdash e_2{:}\,\tau, C'' \qquad \tau \preceq C''(\rho)}{\Gamma, C \vdash e_1 := e_2{:}\,\tau,\ Assign(C'', \rho{:}\,\tau)} \qquad \textsc{Lam}\ \frac{\tau = sp(t) \qquad \epsilon, \epsilon', \kappa \quad fresh \qquad \Gamma[x \mapsto \tau], \epsilon \vdash e : \tau', C' \qquad C' \preceq \epsilon'}{\Gamma, C \vdash \lambda^{L} x : t.e : \kappa(\epsilon, \tau) \rightarrow^{L} (\epsilon', \tau'), C}$$

$$\textsc{App}\ \frac{\Gamma, C \vdash e_1 : Q(\epsilon, \tau) \rightarrow^{L} (\epsilon', \tau'), C' \qquad \Gamma, C' \vdash e_2 : \tau_2, C'' \qquad \tau_2 \preceq \tau \qquad Filter(C'', L) \preceq \epsilon}{\Gamma, C \vdash e_1\ e_2 : \tau',\ Merge(\epsilon', C'', L)} \qquad \textsc{Assert}\ \frac{\Gamma, C \vdash e : Q'\ \sigma, C' \qquad Q' \preceq Q}{\Gamma, C \vdash assert(e, Q) : Q\ \sigma, C'}$$

$$\textsc{Check}\ \frac{\Gamma, C \vdash e : Q'\ \sigma, C' \qquad Q' \preceq Q}{\Gamma, C \vdash check(e, Q) : Q'\ \sigma, C'} \qquad \textsc{Pair}\ \frac{\Gamma, C \vdash e_1{:}\,\tau_1, C' \qquad \Gamma, C' \vdash e_2{:}\,\tau_2, C''}{\Gamma, C \vdash \langle e_1, e_2\rangle{:}\,\kappa\ \langle \tau_1, \tau_2\rangle, C''} \qquad \textsc{Fst}\ \frac{\Gamma, C \vdash e{:}\,Q\ \langle \tau_1, \tau_2\rangle, C'}{\Gamma, C \vdash \mathsf{fst}(e){:}\,\tau_1, C'}$$

$$\textsc{FstAssign}\ \frac{\Gamma, C \vdash e_1{:}\,Q\ ref(\rho), C' \qquad \Gamma, C' \vdash e_2{:}\,\tau_1, C'' \qquad \kappa\ \langle \alpha_1, \alpha_2\rangle \preceq C''(\rho) \qquad \tau_1 \preceq \alpha_1 \qquad \kappa, \alpha_1, \alpha_2\ \text{fresh}}{\Gamma, C \vdash \mathsf{fst}(e_1) := e_2{:}\,\tau_1,\ Assign(C'', \rho{:}\,\langle \tau_1, \mathsf{snd}(C''(\rho))\rangle)}$$

Fig. 2: Type inference system.

Further, by the rule $\textsc{Pair}_{\preceq}$, the subtyping constraints between pair types are decomposed into subtyping constraints between qualifier and simpler types that are inductively decomposed into constraints between qualifiers and linearities. Thus, the added inference rules do not increase the complexity of the generated constraints.

D. Soundness

The type inference has the following soundness property. Consider a given expression e. Consider the set of conditions $\mathcal{C}$ generated during type inference for e in the empty environment and empty store *i.e.,* the constraints generated to derive the judgment $\emptyset, \emptyset \vdash e{:}\,\tau, C'$ for some type τ and store C'. A solution $\mathcal{S}$ for the constraints $\mathcal{C}$ is a mapping from store variables ϵ to concrete stores, from qualifier variables κ to concrete qualifiers, and from type variables α to concrete types such that $\mathcal{S}$ satisfies the constraints $\mathcal{C}$. In other words, substituting each variable in the constraints $\mathcal{C}$ with its mapping in $\mathcal{S}$ results in valid constraints. If there is a solution $\mathcal{S}$ for the constraints $\mathcal{C}$ then the evaluation of e cannot get stuck. The evaluation of an expression can get stuck if a non-reference value is dereferenced, a value is assigned to a non-reference value, a value of a mismatching type is assigned to a reference to a location of a specific type, the parameter of a function is instantiated with an argument of a mismatched type, and more importantly a qualifier check or assertion fails, *i.e.,* the qualifier of a value is not a subtype of the expected qualifier.

REFERENCES

[1] Foster, J.S., Terauchi, T. and Aiken, A., 2002, May. Flow-sensitive type qualifiers. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (pp. 1-12).