Syntactic Qualifier Report

Siyuan He
Purdue University
USA
he662@purdue.edu

Abstract

Syntactic qualifier is an implementation of the reachability tracking qualifier in λ^* system based on the idea of intersection and union types. The syntactic qualifier is constructed from a set of axiomatic rules, where the properties of qualifiers are proved from the basic rules. We expect the syntactic qualifier to substitute for the original set qualifier to solve the potential problem on the unclear expressive power. We have achieved several milestones on the syntactic qualifier including opening, sub-qualifier relation, weakening, and equivalence. However, we stuck on the substitution lemma due to the lack of distributive law in the intersection and union types.

1 Introduction

Ownership has been paid growing attention by researches on type systems because ownership type systems add safety and expressiveness to programming language systems. Reachability type system λ^* [Bao et al. 2021] is one of the promising works, which extends the simply-typed λ-calculus typing system with mutable references by introducing a qualifier to track the set of its reachable free variables.

In syntax, the λ^* system has an additional type qualifier in the type system to track the variables that the term may reach. However, the qualifier is implemented on the finite set and the behavior of the qualifier lacks clear formalization. One remarkable weak point of the set implementation is that the expressive power of finite set is either limited or unnecessary. Namely, if we only use the set inclusion, the extension would be hard because the set inclusion is not expressive enough to represent a set of complex tracking rules. Some attempts on extending the λ^* system such as the must-reachability tracking extension [He 2021] have shown the lack of expressive power of set qualifiers, i.e. the set inclusion is not able to represent a tracking interval. Meanwhile, the original λ^* work has introduced set equality to handle the general cases that self-references are included in the typing rules, which is a weak point of the work because the base is no longer pure sets with only inclusion.

In order to clearly formalize the expected behavior of the qualifier and select proper base theory behind, we come up with the idea of syntactic qualifier based on a set of axiomatic rules. Instead of using an existing structure, we start the qualifier with a set of pre-defined rules and construct anything else using the set of rules, so that the qualifier is axiomatic. One remarkable advantage compared with the original set version is that the potential expressive power, i.e. the scope able to represent, is limited by the axiomatic rules, thus we can choose a proper set of base rules for different extension variants.

Intersection and union type [Pierce 1990] is a set-theoretic type system, which represents sets of values by logical combinators. Intersection type, $\tau \wedge \sigma$ means a value is both τ and σ. While the duality, union type $\tau \vee \sigma$ means a value is either τ or σ.

We presents a syntactic reachability tracking qualifier for the λ^* system as a substitution to the set qualifier in the original work [Bao et al. 2021]. The syntactic qualifier is based on the idea of intersection and union types, which is saying that the sub-qualifier relations come from the subtyping in the intersection and union typing system. We expect the syntactic qualifier to have the same expressive power as the set qualifier that would be able to maintain the type safety proof of the λ^* system.

In summary, the report contains following content:

- We provide an informal technique overview of the syntactic qualifier design (Section 2).
- We summarize the current progress and difficulties in the attempt (Section 3)
- We discuss the potential problems in the current design and next steps (Section 4)

2 Overview of Syntactic Qualifiers

The core of our syntactic qualifier is a set of axiomatic rules on the sub-qualifier relation, so almost all properties of qualifiers are constructed from the sub-qualifier relation in our syntactic design.

2.1 Qualifier Definition

Since the syntactic qualifier is based on the idea of intersection and union types, we reflect the intersection and union with `qand` and `qor` constructors in the qualifier definition. Meanwhile, like the set qualifier system, we reflect the de Bruijn variable and the location with `qvar` and `qloc` constructors.

By these constructors, we can build a tree structure to represent the qualifiers with the set format. For example, a

Author's address: Siyuan He, Purdue University, USA, he662@purdue.edu.

2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/

qualifier containing `[varF 1, varB 2, loc 3]` can be intuitively represented as

```
qor ((qvar (varF 1)) (qor (qvar (varB 2)) (qloc 3)))
```

Note that the `qor` and `qand` constructors have only syntactic meaning, where the meaning from the intersection and union types are achieved by the sub-qualifier relations.

One potential problem caused by this syntactic definition is that we can have various different syntactic tree representation for a same qualifier. As a trivial example, two syntactic qualifiers `(qor empty q)` and `q` reflects the same qualifier in set implementation, but different in syntactic implementation because the constructors have no actual meaning. We solve this problem by proving the equivalence, which will be discussed later.

2.2 Sub-qualifier Relations

In our syntactic qualifier system, the sub-qualifier relations follow the subtyping relation of the intersection and union types, i.e. our sub-qualifier rules are one-to-one reflection of the subtyping rules in the intersection and union rules.

The subqualifier rules in our system can be divided into two parts: reflexivity of the values and the combination rules of two qualifiers derived from intersection and union types. For the reflexivity of values, these are the simple rules indicating that a qualifier having only a value is the sub-qualifier of itself, so we skip them here. The combination rules allow reasoning the sub-qualifier relation of the `qand` and `qor` constructors, with which we can extend the sub-qualifier relation to complex syntactic trees. The sub-qualifier rules are summerized in Fig. 1.

The interesting part is the relation between our combination sub-qualifier rules and the subtyping in the intersection and union types. An intuitive explanation for this is that a value in a qualifier can be viewed as a term having a type, so the sub-qualifier relation can naturally be viewed as the subtyping relation. For example in the `qstp_sub_or` case,

```
| qstp_sub_or : ∀ Γ Σ q1 q2 q3,
    qstp Γ Σ q1 q3 ->
    qstp Γ Σ q2 q3 ->
    qstp Γ Σ (qor q1 q2) q3
```

which corresponds to the `union-low` subtyping rule,

$$\frac{q_1 <: q_3\,,\ q_2 <: q_3}{(q_1 \vee q_2) <: q_3}\ (\textsc{union-low})$$

This rule means that if we know a a value is in either qualifier q_1 or q_2, then we know that it is definitely in the union of two qualifiers, which is similar to the combination use of subsumption rule and union-low subtyping rule in the union type system. The other rules can be explained similarly, so that we borrow the full intersection and union type system.

2.3 Transitivity

Besides the necessary rules to construct the sub-qualifier relation, we add an additional rule of transitivity as an axiom to the system.

```
| qstp_trans : ∀ (Γ Σ q1 q2 q3),
    qstp Γ Σ q1 q2 -> qstp Γ Σ q2 q3 ->
    qstp Γ Σ q1 q3
```

Transitivity law is not an axiomatic rule which must be included for the system, in other word, the transitivity law should be provable with other rules. However, proving transitivity law is obviously hard and requires much efforts. Meanwhile, the transitivity property of the qualifier is not the focus on the work, so we temporarily add the rule as an axiom and proceeds to the λ^* system.

This rule is widely used in the type safety proof of the λ^* system. One remarkable application is to rewrite a sub-qualifier relation with an equivalent expression.

2.4 Equivalence

Equivalence is an important strategy in the proof of sub-qualifier related lemmas. Namely, showing an equivalence relation allows us to simplifying a syntactic qualifier tree and rewrite sub-qualifier goals with an equivalent expression.

In the set qualifier, we are free to use the equality because the each qualifier has a unique expression in set qualifiers. However, recap that the syntactic qualifiers are not unique while have infinite tree structure for a same meaning, so restricting two qualifiers are strictly equal is meaningless. Hence, equivalence is meaningful in the syntactic qualifier system instead of equality. Without equality, we are unable to directly replace or rewrite expressions as convenient as in the set system, which sometimes prevents the proof from progressing. For this reason, we have to select a definition for "equivalence" and construct a rewrite strategy based on the equivalence.

We define the equivalence in our system as the bi-directional sub-qualifier relation, i.e.

```
Inductive eqqual Γ Σ q1 q2 : Prop :=
| eqq: qstp Γ Σ q1 q2 -> qstp Γ Σ q2 q1 ->
    eqqual Γ Σ q1 q2
```

This definition for equivalence is nature because our system is based on the axiomatic sub-qualifier rules, so it would be better to rely on the sub-qualifier relationship. Likewise, the equality relation in sets with only set inclusion is also defined as the bi-directional set inclusion. The bi-directional sub-qualifier equivalence works in our system intuitively because the sub-qualifier relation has some extent of order, i.e. we don't allow the cyclic sub-qualifier relation or the asymmetry. (However, there is a potential problem that we can't prove only an empty qualifier can be the sub-qualifier of an empty qualifier, see 4.1.)

With the equivalence and the transitivity rule, we can develop a rewrite strategy for equivalence expressions in the sub-qualifier judgement. To rewrite, we just need to apply the transitivity rule on a side of sub-qualifier relations in the equivalence, while in one direction in assumptions and the other direction in the goals. The equivalence is a stronger precondition than necessary to do the rewrite, and we adopt

Sub-qualifier Rules

$$\frac{}{\Gamma \vdash \text{empty} \sqsubseteq q}\ (\text{S-EMPTY}) \qquad \frac{}{\Gamma \vdash q \sqsubseteq q}\ (\text{S-REFL}) \qquad \frac{\Gamma \vdash q_1 \sqsubseteq q_2}{\Gamma \vdash q_1 \sqsubseteq (q_2 \vee q_3)}\ (\text{S-UNION-UP-L})$$

$$\frac{\Gamma \vdash q_1 \sqsubseteq q_2}{\Gamma \vdash q_1 \sqsubseteq (q_3 \vee q_2)}\ (\text{S-UNION-UP-R}) \qquad \frac{\Gamma \vdash q_1 \sqsubseteq q_3 \qquad \Gamma \vdash q_2 \sqsubseteq q_3}{\Gamma \vdash (q_1 \vee q_2) \sqsubseteq q_3}\ (\text{S-UNION-LOW})$$

$$\frac{\Gamma \vdash q_1 \sqsubseteq q_2}{\Gamma \vdash (q_1 \wedge q_3) \sqsubseteq q_2}\ (\text{S-INTER-LOW-L}) \qquad \frac{\Gamma \vdash q_1 \sqsubseteq q_2}{\Gamma \vdash (q_3 \wedge q_1) \sqsubseteq q_2}\ (\text{S-INTER-LOW-R}) \qquad \frac{\Gamma \vdash q_1 \sqsubseteq q_2 \qquad \Gamma \vdash q_1 \sqsubseteq q_3}{\Gamma \vdash q_1 \sqsubseteq (q_2 \wedge q_3)}\ (\text{S-INTER-UP})$$

$$\frac{\Gamma \vdash q_1 \sqsubseteq q_2 \qquad \Gamma \vdash q_2 \sqsubseteq q_3}{\Gamma \vdash q_1 \sqsubseteq q_3}\ (\text{S-TRANS})$$

Figure 1. Sub-qualifier rules for the syntactic qualifier.

this strategy because it's more close to the original proof tree.

2.5 Opening and Substituting

The opening and substitution operation are two main operations performed on qualifiers. In our system, the two operations are defined as normal functions. The two functions will recursively traverse the syntactic tree to find the variable to substitute or open and recursively do the replacement.

Note that in the syntactic qualifier system, a value might occur multiple times in the tree structure, for example,

```
qor ((qloc 1) (qand (qloc 1) (qloc 1)))
```

is a possible qualifier tree. To handle these conditions, we simply replace every appearance of the target value without simplifying or evaluating the intersection and union constructors.

3 Progress and Difficulties

Until this report, we have achieved several milestones as well as met several difficulties in the development of the syntactic qualifier. Though it seems most proofs have been reconstructed with the syntactic qualifier, the last few lemmas are the most hard parts and require much efforts, so we might still have a long way to go on this topic.

The first remarkable progress is to modify almost all the basic lemmas related to qualifiers in the original system. The modification is necessary because the sub-qualifier relations of syntactic qualifier rely on the context while that of original set qualifier do not, so that we have to embed context into these judgements. Meanwhile, since the definition of opening and substituting is completely different from the set qualifier, so we adopt new proof trees for these lemmas.

As an important part in the type safety proof, we finish the weakening lemma of the λ^* system with syntactic qualifiers. For these lemmas, we replace the equality in the assumptions of set qualifiers with equivalence assumptions of syntactic qualifiers. We adopt the rewrite strategy for equivalent expressions instead of direct rewrite for equal expression to preserve the proof trees. As a side note, for these expressions not related to sub-qualifier relations, we can keep the equality because the operations besides sub-qualifier relations are normal functions.

We are currently stuck and working on the substitution lemma, with which we are able to finish the type safety theorem proof. There are three non-trivial cases in the substitution lemma: variable case, function abstraction case, and application case. We have completed the application case, almost finish the variable case with two minor goals related to context issue, and stuck on the function abstraction case. Under the current proof tree, we require the distributive law to finish the goal but the distributive law is not supported in our syntactic qualifier system (see discussion 4.2). This difficulty reflects that our equivalence rewrite is not as strong as the equality rewrite in set qualifiers. Possible ways to solve this problem might be either: adopt a new proof tree to avoid this subcase, prove a fake "distributive" lemma in this special case instead of prove a general distributive law (it's the only case requires distributive law till now), or extend the equivalence to a stronger one (see Section 5).

4 Discussion

When developing the syntactic qualifier, we discover several critical points worth further research. These points are the unsolved obstacles in the research on the syntactic qualifier, while we either avoid them or try to avoid them instead of solving these problems. Hence, further research one these points should definitely improve the system.

4.1 Sub-qualifier of Empty Problem

In our design, the empty qualifier is the sub-qualifier of every qualifier, and in the contrary, if a qualifier is the sub-qualifier of an empty qualifier, then we expect the qualifier to be also the empty qualifier. However, the inverse is true

but meaningless because the non-uniqueness of the empty qualifier and the existence of transitive rule. For example, by inverting `q <: qempty`, we can always apply the transitivity rule to get `q <: q' ∧ q' <: qempty`, where the `q'` can be any qualifier equivalent to the empty qualifier. This, this turns the goal back to the initial point so we are unable to prove that.

However, this problem won't harm the system, because we can always derive that the qualifier is equivalent to an empty qualifier and do rewrite.

4.2 Distributive Law

At the first place, remark that our syntactic qualifier is based on the intersection and union types, i.e. a variant of set-theoretic types. The set-theoretic types are sound but incomplete. For example, we can't conclude that

`(Int ∧ Str) ∨ Bool <: (Int ∨ Bool) ∧ (Str ∨ Bool)`

by simply selecting the left branch. Hence, we can never have a proof of distributive law for our syntactic qualifier.

The only case that might need the distributive law is an interesting case. We want to show that

```
q ∧ (q' ∨ (varF (length Γ))) ≡ q ∧ q'
```

, given the assumption that `q` is closed in the context Γ. We know it is true because `(varF (length Γ))` is out of the range of `q`, so it will be dropped because we intersect it with `q`. However, without the distributive law, we can't turn the expression to any cases where we can apply sub-qualifier rules locally to drop the value.

Luckily, we have a chance to avoid this issue since we don't actually need such as strong lemma. The reason why we can drop the value is because it is out of an "active context", so we might alternatively adopt a lemma saying that we're safe to drop any values out of the final approximated context.

4.3 Adding Values, Self-References

This is only a minor problem about the implementation details, not the design of the system. Currently, we adopt a very native method to "add" a variable to the syntactic qualifier by simply union the value, i.e. `v ∨ q`. By using this method, we will have to deal with a union at the top level each time we need to apply opening or substitution. This might add difficulty to the proof work because our equivalence rewrite is too weak to rewrite the sub-expressions.

However, there seems no alternatives more reasonable than this one. If we want to generalize the λ^* type system by adding function references and self-references, this problem will cause dramatic waste of effort. Hence, though it's not important until now, we might need to redesign the interface in the future.

4.4 Extension of Equality

During the development of the syntactic qualifier, we have thought several times about extending the equivalence to equality to our system. Namely, we have tried to develop a set of equality rules allowing us to "evaluate" the qualifier. We want the extension of equality because the equivalence relation is quite weak that having an equivalence relation have only limited help to our proof (though at most cases it's enough). However, we have a lot of equivalence assumptions in the lemmas. With the equivalence only, the expressive power of the type system will be greater because the equivalence assumption is weaker.

5 Next Steps

The next focus will mainly on solving the problem of the distributive law (Section 4.2), so that we can finish a proved type system first. Based on the possible solutions discussed in Section 4, we will perform these steps in order:

- Make attempt on the proof of the substitution lemma using different proof strategies, so that we might avoid the occurrence of the case requiring distributive law. Indeed, we should have confidence because there were many difficulties were solved by changing a proof strategy.
- Develop a set of context narrowing and weakening lemmas that allows us to drop these variables that must not appear in the "evaluated qualifier". However, from my experience, these kinds of lemmas will only take effect in some specialized cases, i.e. probably only in this case.
- Research on the extension of equality. We should try our best to avoid introducing the extension of equality until last point, because it actually harms the aim of the axiomatic qualifier.

As long as we finalize the substitution lemma, the rest should be trivial. So the next step will includes:

- Add bottom to the qualifier. With bottom introduced, we have to modify the definition of the qualifier, the sub-qualifier rules, and the equivalence rewrite. It might cause another round of reconstruction on all the proofs.
- Support overlap. However, one remarkable advantage for using syntactic qualifier is that the overlap is naturally permitted due to the pre-defined rules.
- Add self-references for function abstraction. This might requires redesign the interface of adding a value to the qualifier.

References

Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability Types: Tracking Aliasing and Separation in Higher-Order Functional Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 139 (oct 2021), 32 pages.

Siyuan He. 2021. Must-reachability Tracking Report.

Benjamin C Pierce. 1990. *Intersection and union types.* Technical Report. Citeseer.