Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs (Appendix)

GUANNAN WEI, Purdue University, USA
OLIVER BRAČEVAC*, Purdue University, USA and Galois, Inc., USA
SONGLIN JIA, Purdue University, USA
YUYAN BAO, Augusta University, USA
TIARK ROMPF, Purdue University, USA

A REVISITING λ^* AND ITS LIMITATIONS

This section gives an overview of the λ^*-calculus [Bao et al. 2021] and analyzes its limitaions in reachability polymorphism. Table 1 summarizes the key differences between the λ^* system and our new system, as well as highlights the main improvements made in the main paper [Wei et al. 2024].

A.1 Fresh and Untrack Qualifier

Just like the $\lambda^{\blacklozenge}/\mathsf{F}^{\blacklozenge}_{<:}$-calculus in the main paper, the λ^* type system qualifies types with reachability sets, tracking the variables in the current environment that may be reached by following memory references from the result of an expression.

However, λ^* treats fresh/untrack resources different from the main paper. For example, consider an `alloc()` function that yields a new resource of fixed type `T` (*e.g.*, a file handle). The λ^* system assigns the empty set `alloc()`: $\mathsf{T}^{\varnothing}$ as the qualifier, indicating that it returns a *fresh* value and cannot reach any variables in the current environment. When bound to a variable `x`, an invocation of `alloc()` is not considered fresh anymore as `x` reaches `x` itself:

```
val x = alloc()  // : T^{x}
```

The λ^* system assigns the bottom qualifier $\bot$ (often omitted) to untracked values. These usually include base types, *e.g.*, `42: Int`$^{\bot}$. Untracked values can be treated as tracked by subtyping, but not vice versa, *i.e.*, $\bot <: \varnothing$ and $\varnothing \not<: \bot$.

A.2 Reachability Polymorphism

λ^* provides a *lightweight* form of reachability polymorphism via dependent applications, *e.g.*, the `id` function whose return type qualifier depends on the argument:

```
def id(x: T^∅): T^{x} = x        // : ((x: T^∅) => T^{x})^∅
val x: T^{x,a,b} = ...; id(x)   // : T^{x}[x↦{x,a,b}]  = T^{x,a,b}
val y: T^{y,z}   = ...; id(y)   // : T^{x}[x↦{y,z}]    = T^{y,z}
```

The type of `id` mentions no explicit quantifiers, and yet can be regarded as polymorphic over a fixed base type `T` with any reachability qualifier q, as long as q is disjoint from `id`'s reachability set.

*Work completed while at Purdue University

Authors' addresses: Guannan Wei, Purdue University, West Lafayette, IN, USA, guannanwei@purdue.edu; Oliver Bračevac, Purdue University, West Lafayette, IN, USA and Galois, Inc., Portland, OR, USA, oliver@galois.com; Songlin Jia, Purdue University, West Lafayette, IN, USA, jia137@purdue.edu; Yuyan Bao, Augusta University, USA, yubao@augusta.edu; Tiark Rompf, Purdue University, USA, tiark@purdue.edu.

2024. ACM 2475-1421/2024/1-ART1
https://doi.org/

Table 1. Overview and comparison of λ^* and this work. "–" indicates there is no equivalent notion in the system. The `id` function is the polymorphic identity function as defined in the respective system. We use "MP" to stand for the main paper.

	λ^* [Bao et al. 2021]	$\lambda^\blacklozenge$/$\mathsf{F}^\blacklozenge_{<:}$ (this work)
Untracked Primitive/atomic values	$T^\bot$ `val x = 42 // : Int`$^\bot$	$T^\varnothing$ `val x = 42 // : Int`$^\varnothing$
Reachability Assignment Transitive closure vs. immediate reachability	Reflexive & transitive `val z = x // z : T`$^{\{z,x,...\}}$	One-step by default, transitive on demand (Sec. 3.1.1 in MP) `val z = x // z : T`$^{\{z\}}$
Fresh and Tracked Tracked but unbound in the context	$T^\varnothing$ `alloc() : T`$^\varnothing$	$T^{\{\blacklozenge,...\}}$ (Sec. 3.1.2 in MP) `alloc() : T`$^\blacklozenge$
Reachability Polymorphism Functions preserving reachability that depends on arguments	Non-parametric & imprecise (Sec. A.3) `id(42) : Int`$^\varnothing$ `id(alloc()) : Int`$^\varnothing$	Parametric & precise (Sec. 3.1.3 in MP) `id(42) : Int`$^\varnothing$ `id(alloc()) : Int`$^\blacklozenge$
Qualifier Subtyping How qualifiers can be upcast	Set inclusion $T^{q_1} <: T^{q_2}$ if $q_1 \subseteq q_2$	Context dependent (Sec. 3.1.4 in MP) $\Gamma = \mathtt{x}: \mathtt{T}^\varnothing,\ \mathtt{y}: \mathtt{T}^\blacklozenge$ $\Gamma \vdash \mathtt{T}^{\{x\}} <: \mathtt{T}^\varnothing$ $\Gamma \vdash \mathtt{T}^{\{y\}} \not<: \mathtt{T}^\blacklozenge$
"Maybe" Tracked Variable-dependent tracking status	–	T^q if $\blacklozenge \notin q$ (Sec. 3.1.4 in MP) $\Gamma \vdash \mathtt{T}^{\{x\}} \equiv \mathtt{T}^\varnothing$
Transitive Reachability When transitive closure is used	Always saturated	On-demand when checking overlap (Sec 3.1.5 in MP)
Qualifier-Dependent Application Permitted argument dependency in the return type	Shallow $(x : T^q) \to S^p$ $x \notin fv(S)$	Deep (Sec. 3.1.6 in MP) $(x : T^q) \to S^p$ $x \in fv(S)$ if $\blacklozenge \notin q$
Type Abstraction Quantification over types	–	Bounded abstraction à la $\mathsf{F}_{<:}$ $\forall X <: T.S^p$ (Sec. 3.2 in MP)
Reachability Abstraction Quantification over reachability	–	Bounded abstraction à la $\mathsf{F}_{<:}$ $\forall X^x <: T^q.S^p$ (Sec. 3.2 in MP)
Mutable References Values stored in references	Only flat & untracked Ref[$T^\bot$]	Possibly nested & tracked Ref[T^q] (Sec. 7.1 in MP)

Since `id` itself has an empty qualifier, any q is acceptable. We can apply `id` with an argument with non-fresh tracked qualifiers, and the result precisely preserves the reachability by substituting `x` in the return qualifier with the actual argument qualifier.

A.3 The Root of the Problem: Confusing Untracked with Fresh Values

The problem with reachability polymorphism in λ^* is its non-parametric treatment of untracked versus tracked arguments, *e.g.*, the `id` function conflates these two different instantiations:

```
val z = ...            // : T^⊥                  ← z is untracked
id(z)                  // : T^{x}[x↦⊥] = T^∅     ← untracked value now considered tracked
id(alloc())            // : T^{x}[x↦∅] = T^∅
```

Qualifier substitution with the untracked status yields $\{x\}[x \mapsto \bot] = \varnothing$ a tracked qualifier without known aliases (*i.e.*, fresh). Bao et al. (Section 3.4) made this design choice to ensure soundness, but it introduces imprecision in tracking status and constitutes a severe limitation in expressiveness. No code path can be generic with respect to the tracking status of arguments! To see why admitting a more precise qualifier $\mathtt{T}^\bot$ for `id(z)` is unsound, we can postulate this "more precise" behavior (*i.e.*, assuming $\{x\}[x \mapsto \bot] = \bot$) and subvert the type system. Consider the function `fakeid` returning a fresh tracked value each time:

```
def fakeid(x: T^∅): T^{x} = alloc()
```

This function typechecks since the body expression has type $T^{\varnothing}$, which is a subtype of the declared return type $T^{\{x\}}$. Under the postulate, applying `fakeid` with a non-tracking arguments results in

```
val y = ...                  // : T^⊥
fakeid(y)                    // : T^{x}[x↦⊥] = T^⊥ ⟵ unsound!
```

But `fakeid(y)` actually returns a fresh value of qualifier $\varnothing$ that should never be down-cast to untracked! This violates the *separation guarantee* of the type system: a tracked value cannot escape as an untracked value. Otherwise, it can no longer be kept separate from other tracked values.

To summarize, reachability polymorphism via dependent application in λ^* must sacrifice parametricity and precision for soundness, leading to a confusion of untracked with fresh values. There is no easy fix with the binary track/untrack distinction, and we must rethink reachability polymorphism and the notion of freshness.

B TYPING POLYMORPHIC DATA STRUCTURES

In this section, we discuss typing common polymorphic data types in $\mathsf{F}^{\blacklozenge}_{<:}$. Here we present them as built-in language constructs for clarity. However, they can be encoded in the $\mathsf{F}^{\blacklozenge}_{<:}$-calculus as well, in similar ways as exemplified in Section 8.1 of the main paper. All data types come with self-references (*i.e.*, z in $\mu z.\mathrm{Box}[Q]$), so that qualifiers in Q can refer to the instance itself by name z. Similar to function self-references, they admit Q-TSELF for subtyping (generalizing Q-SELF in Fig. 5 of the main paper, which only works for function's self-references), thus allowing the introduction and elimination of self-references.

$$\frac{z : \mu z.T^{q} \in \Gamma \qquad \blacklozenge \notin q}{\Gamma \vdash q, z <: z} \tag{Q-TSelf}$$

These data structures have standard dynamic semantics (*e.g.* tagging runtime values), thus are omitted.

B.1 Boxes

We start from box types, which is the simplest polymorphic data type. It comes with two qualifiers, one for its content and one for the box itself. Creating a box with a non-fresh value results in the same inner and outer qualifier. When creating a box with a fresh value, the inner qualifier is the box itself z (T-BOX). This is necessary to maintain sharing when eliminating multiple times.

Getting the content of a box value yields the qualifier that replaces the self-reference z with its its own outer qualifier (T-BOX-GET). Furthermore, the content types of boxes are covariant (SQ-BOX), so when a variable name goes out of its binding scope, the inner qualifier can be upcast to z by Q-TSELF as if it was created over a fresh value.

B.2 Pairs

Pairs follow the same pattern with boxes, with an additional field, and thus type argument and elimination function. Note that pairs over non-fresh values correspond to the "transparent" pairs presented in Section 8 of the main paper, while those over fresh values correspond to the "opaque" pairs. The two variants also connect via subtyping, as component types in pairs are covariant.

B.3 Options

Option types are similar to boxes, which can optionally hold no value. Since there are two variant values of option types, applying `get` to values of type `Option` can result in an exception if the

Syntax $\boxed{\mathsf{F}^{\blacklozenge}_{<:}}$

$$\begin{array}{llll} T & ::= & \cdots \mid \mu z.\mathsf{Box}[Q] & \text{Types} \\ t & ::= & \cdots \mid \mathsf{Box}(t) \mid \mathsf{get}(t) & \text{Terms} \end{array}$$

Term Typing $\boxed{\Gamma^{\varphi} \vdash t : Q}$

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad q' = \text{if } \blacklozenge \in q \text{ then } z \text{ else } q}{\Gamma^{\varphi} \vdash \mathsf{Box}(t) : \mu z.\mathsf{Box}[T^{q'}]^{q}} \quad \text{(T-BOX)}$$

$$\frac{\Gamma^{\varphi} \vdash t : \mu z.\mathsf{Box}[T^{q_1}]^{q}}{\Gamma^{\varphi} \vdash \mathsf{get}(t) : T^{q_1[q/z]}} \quad \text{(T-BOX-GET)}$$

Subtyping $\boxed{\Gamma \vdash Q <: Q}$

$$\frac{\Gamma, z : \mu z.\mathsf{Box}[Q_1]^{q} \vdash Q_1 <: Q_2}{\Gamma \vdash \mu z.\mathsf{Box}[Q_1]^{q} <: \mu z.\mathsf{Box}[Q_2]^{q}} \quad \text{(SQ-BOX)}$$

Fig. 1. Extension: Box types.

Syntax $\boxed{\mathsf{F}^{\blacklozenge}_{<:}}$

$$\begin{array}{llll} T & ::= & \cdots \mid \mu z.\mathsf{Pair}[Q_1, Q_2] & \text{Types} \\ t & ::= & \cdots \mid \mathsf{Pair}(t_1, t_2) \mid \mathsf{fst}(t) \mid \mathsf{snd}(t) & \text{Terms} \end{array}$$

Term Typing $\boxed{\Gamma^{\varphi} \vdash t : Q}$

$$\frac{\begin{array}{ll} \Gamma^{\varphi} \vdash t_1 : T_1^{q_1} & q_1' = \text{if } \blacklozenge \in q_1 \text{ then } z \text{ else } q_1 \\ \Gamma^{\varphi} \vdash t_2 : T_2^{q_2} & q_2' = \text{if } \blacklozenge \in q_2 \text{ then } z \text{ else } q_2 \end{array}}{\Gamma^{\varphi} \vdash \mathsf{Pair}(t_1, t_2) : \mu z.\mathsf{Pair}[T_1^{q_1'}, T_2^{q_2'}]^{q_1, q_2}} \quad \text{(T-PAIR)}$$

$$\frac{\Gamma^{\varphi} \vdash t : \mu z.\mathsf{Pair}[T_1^{q_1}, T_2^{q_2}]^{q}}{\Gamma^{\varphi} \vdash \mathsf{fst}(t) : T_1^{q_1[q/z]}} \quad \text{(T-FST)}$$

$$\frac{\Gamma^{\varphi} \vdash t : \mu z.\mathsf{Pair}[T_1^{q_1}, T_2^{q_2}]^{q}}{\Gamma^{\varphi} \vdash \mathsf{snd}(t) : T_2^{q_2[q/z]}} \quad \text{(T-SND)}$$

Subtyping $\boxed{\Gamma \vdash Q <: Q}$

$$\frac{\begin{array}{c} \Gamma, z : \mu z.\mathsf{Pair}[Q_1, R_1]^{q} \vdash Q_1 <: Q_2 \\ \Gamma, z : \mu z.\mathsf{Pair}[Q_1, R_1]^{q} \vdash R_1 <: R_2 \end{array}}{\Gamma \vdash \mu z.\mathsf{Pair}[Q_1, R_1]^{q} <: \mu z.\mathsf{Pair}[Q_2, R_2]^{q}} \quad \text{(SQ-PAIR)}$$

Fig. 2. Extension: Pair types.

Syntax $\boxed{F^{\blacklozenge}_{<:}}$

T	$::=$	$\cdots \mid \bot \mid \mu z.\mathsf{Option}[Q]$	Types
t	$::=$	$\cdots \mid \mathsf{Some}(t) \mid \mathsf{None} \mid \mathsf{isEmpty}(t) \mid \mathsf{get}(t)$	Terms

Term Typing $\boxed{\Gamma^{\varphi} \vdash t : Q}$

$$\frac{\Gamma^{\varphi} \vdash t : T^{q} \qquad q' = \text{if } \blacklozenge \in q \text{ then } z \text{ else } q}{\Gamma^{\varphi} \vdash \mathsf{Some}(t) : \mu z.\mathsf{Option}[T^{q'}]^{q}} \quad \text{(T-SOME)}$$

$$\frac{}{\Gamma^{\varphi} \vdash \mathsf{None} : \mu z.\mathsf{Option}[\bot^{\varnothing}]^{\varnothing}} \quad \text{(T-NONE)}$$

$$\frac{\Gamma^{\varphi} \vdash t : \mu z.\mathsf{Option}[T^{q_1}]^{q}}{\Gamma^{\varphi} \vdash \mathsf{isEmpty}(t) : \mathsf{Bool}^{\varnothing}} \quad \text{(T-OPTION-TEST)}$$

$$\frac{\Gamma^{\varphi} \vdash t : \mu z.\mathsf{Option}[T^{q_1}]^{q}}{\Gamma^{\varphi} \vdash \mathsf{get}(t) : T^{q_1[q/z]}} \quad \text{(T-OPTION-GET)}$$

Subtyping $\boxed{\Gamma \vdash T <: T}$ $\boxed{\Gamma \vdash Q <: Q}$

$$\frac{}{\Gamma \vdash \bot <: T} \quad \text{(S-BOT)}$$

$$\frac{\Gamma, z : \mu z.\mathsf{Option}[Q_1]^{q} \vdash Q_1 <: Q_2}{\Gamma \vdash \mu z.\mathsf{Option}[Q_1]^{q} <: \mu z.\mathsf{Option}[Q_2]^{q}} \quad \text{(SQ-OPTION)}$$

Fig. 3. Extension: Option type.

underlying is `None`. The predicate `isEmpty` is added to safely use `get` with runtime guards. Moreover, to construct the `None` case, we introduce the bottom type $\bot$, which is a subtype of all types. The content type of options are also covariant.

B.4 Lists

Lists further extend options in a recursive manner. The `Cons` constructor takes a value and a `List` of the same type. The qualifier of the result is then the union of both the head and tail qualifier, modulo the adaption for self-references. The `hd` eliminator follows the same pattern for retrieving content, while the `tl` eliminator returns the same `List` type unchanged. Note that given a well-typed list of type `List[T`^{q}`]`, its inner qualifier `q` subsumes all element qualifiers.

B.5 Example: Non-Overlapping Lists

Here we showcase how to interact with our polymorphic recursive data type, *i.e.*, `List`. The function `makeList` below recursively creates a fresh list of `Ref[Int]`. By T-CONS, we keep the inner qualifier of its return type to self-reference z, and the outer qualifier to freshness. By creating two lists `c1` and `c2`, we also unpack the self-references of lists to their concrete bindings.

```
def makeList(n: Int): μz.List[Ref[Int]^z]^♦ =
  if (n == 0) Nil else Cons(new Ref(n), makeList(n - 1))
```

Syntax $\boxed{F^{\blacklozenge}_{<:}}$

$$
\begin{array}{lcll}
T & ::= & \cdots \mid \mu z.\mathsf{List}[Q] & \text{Types} \\
t & ::= & \cdots \mid \mathsf{Cons}(t_1, t_2) \mid \mathsf{Nil} \mid \mathsf{isEmpty}(t) \mid \mathsf{hd}(t) \mid \mathsf{tl}(t) & \text{Terms}
\end{array}
$$

Term Typing $\boxed{\Gamma^{\varphi} \vdash t : Q}$

$$\frac{\Gamma^{\varphi} \vdash t_1 : T^{q_1} \qquad \Gamma^{\varphi} \vdash t_2 : \mu z.\mathsf{List}[T^{q_2}]^{q_3} \qquad q_1' = \text{if } \blacklozenge \in q_1 \text{ then } z \text{ else } q_1}{\Gamma^{\varphi} \vdash \mathsf{Cons}(t_1, t_2) : \mu z.\mathsf{List}[T^{q_1', q_2}]^{q_1, q_3}} \quad \text{(T-CONS)}$$

$$\frac{}{\Gamma^{\varphi} \vdash \mathsf{Nil} : \mu z.\mathsf{List}[\bot^{\varnothing}]^{\varnothing}} \quad \text{(T-NIL)}$$

$$\frac{\Gamma^{\varphi} \vdash t : \mu z.\mathsf{List}[T^{q_1}]^{q}}{\Gamma^{\varphi} \vdash \mathsf{isEmpty}(t) : \mathsf{Bool}^{\varnothing}} \quad \text{(T-LIST-TEST)}$$

$$\frac{\Gamma^{\varphi} \vdash t : \mu z.\mathsf{List}[T^{q_1}]^{q}}{\Gamma^{\varphi} \vdash \mathsf{hd}(t) : T^{q_1[q/z]}} \quad \text{(T-HEAD)}$$

$$\frac{\Gamma^{\varphi} \vdash t : \mu z.\mathsf{List}[T^{q_1}]^{q}}{\Gamma^{\varphi} \vdash \mathsf{tl}(t) : \mu z.\mathsf{List}[T^{q_1}]^{q}} \quad \text{(T-TAIL)}$$

Subtyping $\boxed{\Gamma \vdash Q <: Q}$

$$\frac{\Gamma, z : \mu z.\mathsf{List}[Q_1]^{q} \vdash Q_1 <: Q_2}{\Gamma \vdash \mu z.\mathsf{List}[Q_1]^{q} <: \mu z.\mathsf{List}[Q_2]^{q}} \quad \text{(SQ-LIST)}$$

Fig. 4. Extension: Lists.

```
val c1 = makeList(10); // : List[Ref[Int]^c1]^c1 ⊣ c1: μz.List[Ref[Int]^z]^♦
val c2 = makeList(20); // : List[Ref[Int]^c2]^c2 ⊣ c2: μz.List[Ref[Int]^z]^♦
```

We then process the two lists in parallel using the parallel combinator `par` (Section 2.1 in the main paper), which takes two separate thunks. The `parProc` function takes two lists argument `xs` and `ys`, which are required to be separate. Since the curried function `parProc` takes `xs` as the first argument, the remainder function (taking `ys`) already captures `xs`, and thus the freshness marker on `ys` signifies its separation from precisely `xs`.

```
// def par(a: (() => Unit)^♦)(b: (() => Unit)^♦): Unit
def parProc(xs: μz.List[Ref[Int]^z]^♦)(ys: μz.List[Ref[Int]^z]^♦): Unit =
  par { foreach(xs) { x := !x + 1 } }
      { foreach(ys) { y := !y + 1 } }
parProc(c1)(c2) // ok
parProc(c1)(c1) // type error
```

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 (2021), 1–32.

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic reachability types: Tracking freshness, aliasing, and separation in higher-order generic programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 393–424.