Typed regions

Stefan Monnier

Université de Montréal

Introduction

Formal methods on the rise: language-based security

Maturity: bugs are more than an annoyance

Verify more properties than memory safety:

Java brought memory safety to the masses, now they want more

Type check low-level code: distrust tricky code

Type systems are oblivious to side-effects

Contributions

A new type system:

- Hybrid: regions $\otimes$ alias types $\otimes$ proofs
- *Strong update* with arbitrary aliasing
- Manipulate and reason about arbitrary state properties
- modularity of type systems and power of Hoare logic
- Able to type check a realistic GC
- With a better replacement for *widen*

Regions

A region holds multiple objects, deallocated all at once

Every allocation specifies the region: $\mathsf{put}[\rho]\ v$

A new type for pointers: $\tau\ \mathsf{at}\ \rho$

$$\begin{aligned}(\textit{types})\ \tau ::= {} & t \mid \mathsf{int} \mid \tau \times \tau \mid \tau\ \mathsf{at}\ \rho \\ & \mid \forall[\vec{t}]\{\vec{\rho}\}(\vec{\tau}) \to 0\end{aligned}$$

If a region does not appear in an object's type, it is not needed

$\mathsf{free}\ \rho$ can now be checked for safety

Simple and efficient

Alias types

A pointer to address ℓ has type: $\textsf{ptr}\ \ell$

$$(\textit{types})\ \tau ::= t \mid \textsf{int} \mid \tau \times \tau \mid \textsf{ptr}\ \ell$$
$$\mid \forall[\vec{t}]\{\overrightarrow{\ell \mapsto \tau}\}(\vec{\tau}) \to 0$$

The heap has its own, separately maintained type. For example:

$$\{\ell_1 \mapsto (\textsf{int}, \textsf{int}), \ell_2 \mapsto (\textsf{ptr}\ \ell_1, \textsf{ptr}\ \ell_2)\}$$

Dangling pointers like $\textsf{ptr}\ \ell_3$ are allowed but unusable

$$\begin{array}{ll} p : \textsf{ptr}\ \ell = \textsf{new}\ 2; & \{\ell \mapsto (\top, \top), \ldots\} \\ p.0 := 1; & \{\ell \mapsto (\textsf{int}, \top), \ldots\} \\ p.1 := p; & \{\ell \mapsto (\textsf{int}, \textsf{ptr}\ \ell), \ldots\} \end{array}$$

Low-level, very powerful, but restrictive

A fundamental principle of type soundness

A memory location cannot have 2 types at the same time

Enforced in the following ways:

regions The type of a location is immutable

$\Longrightarrow$ can be copied freely

alias types The type of a location is never copied

$\Longrightarrow$ can be changed at any time

Very few exceptions

	intuitionistic value	linear state
regions	τ at ρ	$\{\vec{\rho}\}$
alias types	ptr ℓ	$\{\overrightarrow{\ell \mapsto \tau}\}$

Typed regions

A hybrid between simple regions and alias types:

Part immutable/copiable and part mutable/centralized

$$\begin{array}{rl} (\textit{types})\ \tau ::= & t \mid \text{int} \mid \tau \times \tau \mid \tau \text{ at } \rho.n \\ & \mid \forall[\vec{t}]\{\overrightarrow{\rho \mapsto \varphi}\}(\vec{\tau}) \to 0 \end{array}$$

Pointers have type τ at $\rho.n$: the n^{th} object in ρ, of *intended* type τ

Every region has its own type φ, maintained separately

The *intended* type of a location is not necessarily its *actual* type

The actual type of a location depends on the region's type

The type of a region

A region's type is a type function of 2 parameters: n and τ

$$\text{fun } \textit{plain}\ n\ t = t$$

$$\text{fun } \textit{alias}\ n\ t = \text{if } n = 1 \text{ then } (\text{int}, \text{int}) \text{ else } \bot$$

A region of type *plain* is like a traditional region: $\exists n.\tau \text{ at } \rho.n \;\simeq\; \tau \text{ at } \rho$

alias corresponds to alias types: $\exists t.t \text{ at } \rho.n \;\simeq\; \text{ptr } \rho.n$

The type language is the calculus of inductive constructions, a powerful λ-calculus.

Why?

Copy/create arbitrary cycles: no base case, unknown aliasing

Alias types are too restrictive: pointer reversal with unknown aliasing

Expose memory layout: scanning a region

$$\textit{next} : \tau \text{ at } \rho.n \rightarrow \exists t.t \text{ at } \rho.(n+1)$$

Used by all forms of GC

Clean up *widen*