3.3 Instantiating anodization: Recency abstraction

In recency abstraction [2], the most-recently allocated abstract variant of a resource is tracked distinctly from previously allocated variants. Anodization makes it straightforward to model recency in a higher-order setting. In a language with mutation, recency abstraction solves the initialization problem, whereby addresses are allocated with a default value, but then set to another shortly thereafter. Recency abstraction prevents the default value from appearing as a possibility for every address, which is directly useful in eliminating null-pointer checks. In a higher-order setting, recency permits precise computation of binding equivalence for variables that are bound in non-recursive and tail-recursive procedures or that die before the recursive call.

3.4 Instantiating anodization: Closure-focusing

Anodization enables another shape-analytic technique known as focusing [15, 23]. In focusing, a specific, previously-allocated variant is split into the singleton variant under focus—and all other variants. In a higher-order language, there is a natural opportunity to focus on all of the bindings of a closure when it is created. Focusing provides a way to solve the environment problem for closures which capture variables which have been re-bound since closure-creation.

4 Analogy: Binding invariants as shape predicates

Anodization can solve the environment problem, but it cannot solve the generalized environment problem, where we need to be able to reason about the equality of bindings to *different* variables in *different* environments. To solve this problem, we cast shape predicates as *binding invariants*. A binding invariant is an equivalence relation over abstract bindings, and it can be considered as a separate, relational abstraction of program state, $\alpha^{\eta}_{\equiv} : \Sigma \to \hat{\Sigma}_{\equiv}$, where:

$$\hat{\Sigma}_{\equiv} = \mathcal{P}\left(\widehat{Bind} \times \widehat{Bind}\right),$$

such that:

$$\alpha^{\eta}_{\equiv}(call, \beta, ve, t) = \left\{(\hat{b}, \hat{b}') : ve(b) = ve(b') \text{ if } \eta(b) = \hat{b} \text{ and } \eta(b') = \hat{b}'\right\}.$$

In contrast with earlier work, binding-invariant abstraction is a relational abstract domain over *abstract bindings* rather than *program variables* [7, 8].

Informally, if $(\hat{b}, \hat{b}') \in \alpha^{\eta}_{\equiv}(\varsigma)$, it means that all of the concrete constituents of the bindings $\hat{b}$ and $\hat{b}'$ agree in value. To create the analysis, we can formulate a new abstraction as the direct product of the abstractions α^{η} and $\alpha^{\eta}_{\equiv}$:

$$\dot{\alpha^{\eta}} : \Sigma \to \hat{\Sigma} \times \hat{\Sigma}_{\equiv}$$
$$\dot{\alpha^{\eta}}(\varsigma) = (\alpha^{\eta}(\varsigma), \alpha^{\eta}_{\equiv}(\varsigma)).$$