Funcons-beta: Binding [*]

The PLanCompS Project

`Binding.cbs` | PLAIN | PRETTY

OUTLINE

Binding

[*Type*	environments
Alias	envs
Datatype	identifiers
Alias	ids
Funcon	identifier-tagged
Alias	id-tagged
Funcon	fresh-identifier
Entity	environment
Alias	env
Funcon	initialise-binding
Funcon	bind-value
Alias	bind
Funcon	unbind
Funcon	bound-directly
Funcon	bound-value
Alias	bound
Funcon	closed
Funcon	scope
Funcon	accumulate
Funcon	collateral
Funcon	bind-recursively
Funcon	recursive]

Meta-variables $T <:$ values

[*] Suggestions for improvement: `plancomps@gmail.com`.
Reports of issues: `https://github.com/plancomps/CBS-beta/issues`.

Environments

Type environments $\leadsto$ maps(identifiers, values?)

Alias envs = environments

An environment represents bindings of identifiers to values. Mapping an identifier to ( ) represents that its binding is hidden.

Circularity in environments (due to recursive bindings) is represented using bindings to cut-points called links. Funcons are provided for making declarations recursive and for referring to bound values without explicit mention of links, so their existence can generally be ignored.

Datatype identifiers ::= {_ : strings} | identifier-tagged(_ : identifiers, _ : values)

Alias ids = identifiers

Alias id-tagged = identifier-tagged

An identifier is either a string of characters, or an identifier tagged with some value (e.g., with the identifier of a namespace).

Funcon fresh-identifier : $\Rightarrow$ identifiers

fresh-identifier computes an identifier distinct from all previously computed identifiers.

Rule fresh-identifier $\leadsto$ identifier-tagged(“`generated`”, fresh-atom)

Current bindings

Entity environment(_ : environments) $\vdash$ _ $\longrightarrow$ _

Alias env = environment

The environment entity allows a computation to refer to the current bindings of identifiers to values.

Funcon initialise-binding(X : $\Rightarrow T$) : $\Rightarrow T$
    $\leadsto$ initialise-linking(initialise-generating(closed(X)))

initialise-binding(X) ensures that X does not depend on non-local bindings. It also ensures that the linking entity (used to represent potentially cyclic bindings) and the generating entity (for creating fresh identifiers) are initialised.

Funcon bind-value(I : identifiers, V : values) : $\Rightarrow$ environments
    $\leadsto \{I \mapsto V\}$

Alias bind = bind-value

bind-value(I, X) computes the environment that binds only I to the value computed by X.

Funcon unbind(I : identifiers) : $\Rightarrow$ environments
    $\leadsto \{I \mapsto (\ )\}$

unbind(I) computes the environment that hides the binding of I.

Funcon bound-directly(_ : identifiers) : ⇒ values

bound-directly(I) returns the value to which I is currently bound, if any, and otherwise fails.

bound-directly(I) does *not* follow links. It is used only in connection with recursively-bound values when references are not encapsulated in abstractions.

Rule
$$\frac{\text{lookup}(\rho, I) \rightsquigarrow (V : \text{values})}{\text{environment}(\rho) \vdash \text{bound-directly}(I : \text{identifiers}) \longrightarrow V}$$

Rule
$$\frac{\text{lookup}(\rho, I) \rightsquigarrow (\ )}{\text{environment}(\rho) \vdash \text{bound-directly}(I : \text{identifiers}) \longrightarrow \text{fail}}$$

Funcon bound-value(I : identifiers) : ⇒ values
  ⇝ follow-if-link(bound-directly(I))

Alias bound = bound-value

bound-value(I) inspects the value to which I is currently bound, if any, and otherwise fails. If the value is a link, bound-value(I) returns the value obtained by following the link, if any, and otherwise fails. If the inspected value is not a link, bound-value(I) returns it.

bound-value(I) is used for references to non-recursive bindings and to recursively-bound values when references are encapsulated in abstractions.

Scope

Funcon closed(X : ⇒ T) : ⇒ T

closed(X) ensures that X does not depend on non-local bindings.

Rule
$$\frac{\text{environment}(\text{map}(\ )) \vdash X \longrightarrow X'}{\text{environment}(\_) \vdash \text{closed}(X) \longrightarrow \text{closed}(X')}$$

Rule $\text{closed}(V : T) \rightsquigarrow V$

Funcon scope(_ : environments, _ : ⇒ T) : ⇒ T

scope(D, X) executes D with the current bindings, to compute an environment ρ representing local bindings. It then executes X to compute the result, with the current bindings extended by ρ, which may shadow or hide previous bindings.

closed(scope(ρ, X)) ensures that X can reference only the bindings provided by ρ.

Rule
$$\frac{\text{environment}(\text{map-override}(\rho_1, \rho_0)) \vdash X \longrightarrow X'}{\text{environment}(\rho_0) \vdash \text{scope}(\rho_1 : \text{environments}, X) \longrightarrow \text{scope}(\rho_1, X')}$$

Rule $\text{scope}(\_ : \text{environments}, V : T) \rightsquigarrow V$

Funcon accumulate(_ : (⇒ environments)*) : ⇒ environments

accumulate(D_1, D_2) executes D_1 with the current bindings, to compute an environment ρ_1 representing some local bindings. It then executes D_2 to compute an environment ρ_2 representing further local bindings, with the current bindings extended by ρ_1, which may shadow or hide previous current bindings. The result is ρ_1 extended by ρ_2, which may shadow or hide the bindings of ρ_1.

accumulate(_, _) is associative, with map( ) as unit, and extends to any number of arguments.

Rule $$\frac{D_1 \longrightarrow D_1'}{\text{accumulate}(D_1, D_2) \longrightarrow \text{accumulate}(D_1', D_2)}$$

Rule $\text{accumulate}(\rho_1 : \text{environments}, D_2) \rightsquigarrow \text{scope}(\rho_1, \text{map-override}(D_2, \rho_1))$

Rule $\text{accumulate}(\ ) \rightsquigarrow \text{map}(\ )$

Rule $\text{accumulate}(D_1) \rightsquigarrow D_1$

Rule $\text{accumulate}(D_1, D_2, D^+) \rightsquigarrow \text{accumulate}(D_1, \text{accumulate}(D_2, D^+))$

Funcon $\text{collateral}(\rho^* : \text{environments}^*) : \Rightarrow \text{environments}$
$\quad \rightsquigarrow \text{checked map-unite}(\rho^*)$

collateral$(D_1, \cdots)$ pre-evaluates its arguments with the current bindings, and unites the resulting maps, which fails if the domains are not pairwise disjoint.

collateral(D_1, D_2) is associative and commutative with map$(\ )$ as unit, and extends to any number of arguments.

Recurse

Funcon $\text{bind-recursively}(I : \text{identifiers}, E : \Rightarrow \text{values}) : \Rightarrow \text{environments}$
$\quad \rightsquigarrow \text{recursive}(\{I\}, \text{bind-value}(I, E))$

bind-recursively(I, E) binds I to a link that refers to the value of E, representing a recursive binding of I to the value of E. Since bound-value(I) follows links, it should not be executed during the evaluation of E.

Funcon $\text{recursive}(SI : \text{sets}(\text{identifiers}), D : \Rightarrow \text{environments}) : \Rightarrow \text{environments}$
$\quad \rightsquigarrow \text{re-close}(\text{bind-to-forward-links}(SI), D)$

recursive(SI, D) executes D with potential recursion on the bindings of the identifiers in the set SI (which need not be the same as the set of identifiers bound by D).

Auxiliary Funcon $\text{re-close}(M : \text{maps}(\text{identifiers}, \text{links}), D : \Rightarrow \text{environments}) : \Rightarrow \text{environments}$
$\quad \rightsquigarrow \text{accumulate}(\text{scope}(M, D), \text{sequential}(\text{set-forward-links}(M), \text{map}(\ )))$

re-close(M, D) first executes D in the scope M, which maps identifiers to freshly allocated links. This computes an environment ρ where the bound values may contain links, or implicit references to links in abstraction values. It then sets the link for each identifier in the domain of M to refer to its bound value in ρ, and returns ρ as the result.

Auxiliary Funcon $\text{bind-to-forward-links}(SI : \text{sets}(\text{identifiers})) : \Rightarrow \text{maps}(\text{identifiers}, \text{links})$

```
⇝ map-unite(
      interleave-map(
         bind-value(given, fresh-link(values)),
         set-elements(SI)))
```

bind-to-forward-links(SI) binds each identifier in the set SI to a freshly allocated link.

Auxiliary Funcon $\text{set-forward-links}(M : \text{maps}(\text{identifiers}, \text{links})) : \Rightarrow \text{null-type}$

```
⇝ effect(
      interleave-map(
         set-link(map-lookup(M, given), bound-value(given)),
         set-elements(map-domain(M))))
```

For each identifier I in the domain of M, set-forward-links(M) sets the link to which I is mapped by M to the current bound value of I.