CBS-LaTeX

Peter Mosses

November 2020

Samples

The LaTeX markup, embedded in Markdown, is to be generated from plain CBS specifications using Spoofax. Before implementing it, an editor was used to transform two plain text specifications from CBS-beta into the intended markup.

> All the syntax and semantics names below are linked to their (local or online) declarations, but links for funcon names are generally omitted.

Language "SIMPLE"

Statements

Syntax *Block* : block ::= `{` stmts? `}`

Stmts : stmts ::= stmt stmts?

Stmt : stmt ::= imp-stmt | vars-decl

ImpStmt : imp-stmt ::= block
| exp `;`
| `if (` exp `)` block (`else` block)?
| `while (` exp `)` block
| `for (` stmt exp `;` exp `)` block
| `print (` exps `) ;`
| `return` exp? `;`
| `try` block `catch (` id `)` block
| `throw` exp `;`

Rule ⟦ `if (` *Exp* `)` *Block* ⟧ : stmt = ⟦ `if (` *Exp* `)` *Block* `else { }` ⟧

Rule ⟦ `for (` *Stmt* Exp_1 `;` Exp_2 `) {` *Stmts* `}` ⟧ : stmt =
⟦ `{` *Stmt* `while (` Exp_1 `) { {` *Stmts* `}` Exp_2 `; } }` ⟧

Semantics exec⟦ _ : stmts ⟧ : ⇒ null-type

Rule exec⟦ `{ }` ⟧ = null

Rule exec⟦ `{` *Stmts* `}` ⟧ = exec⟦ *Stmts* ⟧

Rule exec⟦ *ImpStmt* *Stmts* ⟧ = sequential(exec⟦ *ImpStmt* ⟧, exec⟦ *Stmts* ⟧)

Rule exec⟦ *VarsDecl* *Stmts* ⟧ = scope(declare⟦ *VarsDecl* ⟧, exec⟦ *Stmts* ⟧)

Rule exec⟦ *VarsDecl* ⟧ = effect(declare⟦ *VarsDecl* ⟧)

Rule exec⟦ *Exp* `;` ⟧ = effect(rval⟦ *Exp* ⟧)

Rule exec⟦ `if (` *Exp* `)` $Block_1$ `else` $Block_2$ ⟧ =
  if-else(rval⟦ *Exp* ⟧, exec⟦ $Block_1$ ⟧, exec⟦ $Block_2$ ⟧)

Rule exec⟦ `while (` *Exp* `)` *Block* ⟧ = while(rval⟦ *Exp* ⟧, exec⟦ *Block* ⟧)

Rule exec⟦ `print (` *Exps* `) ;` ⟧ = print(rvals⟦ *Exps* ⟧)

Rule exec⟦ `return` *Exp* `;` ⟧ = return(rval⟦ *Exp* ⟧)

Rule exec⟦ `return ;` ⟧ = return(null)

Rule exec⟦ `try` $Block_1$ `catch(`*Id*`)` $Block_2$ ⟧ =
  handle-thrown(
    exec⟦ $Block_1$ ⟧,
    scope(
      bind(id⟦ *Id* ⟧, allocate-initialised-variable(values, given)),
      exec⟦ $Block_2$ ⟧))

Rule exec⟦ `throw` *Exp* `;` ⟧ = throw(rval⟦ *Exp* ⟧)

Funcons-beta/Computations/Normal

Binding

Contents

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

Environments

Type environments $\rightsquigarrow$ 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, _ : maps)

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 : ⇒ identifiers

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

Rule fresh-identifier ⇝ identifier-tagged("`generated`", fresh-atom)

Current bindings

Entity environment(_ : environments) ⊢ _ ⟶ _
Alias env = environment

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

Funcon initialise-binding(X : ⇒ T) : ⇒ T
⇝ 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 : maps) : ⇒ environments
⇝ $\{I \mapsto V\}$
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) : ⇒ environments
⇝ $\{I \mapsto (\,)\}$

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

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

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

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

Funcon bound-value(I : identifiers) : $\Rightarrow$ maps
$\rightsquigarrow$ 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 : $\Rightarrow T$) : $\Rightarrow T$

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

Rule $\dfrac{\text{environment}(\text{map}(\ )) \vdash X \longrightarrow X'}{\text{environment}(\_) \vdash \text{closed}(X) \longrightarrow \text{closed}(X')}$
Rule closed(V : T) $\rightsquigarrow V$

Funcon scope(_ : environments, _ : $\Rightarrow T$) : $\Rightarrow T$

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 $\dfrac{\text{environment}(\text{map-override}(\rho_1, \rho_0)) \vdash X \longrightarrow X'}{\text{environment}(\rho_0) \vdash \text{scope}(\rho_1 : \text{environments}, X) \longrightarrow \text{scope}(\rho_1, X')}$
Rule scope(_ : environments, V : T) $\rightsquigarrow V$

Funcon accumulate(_ : ($\Rightarrow$ environments)*) : $\Rightarrow$ 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.

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

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

$$\textit{Rule}\ \mathsf{accumulate}(\ ) \rightsquigarrow \mathsf{map}(\ )$$

$$\textit{Rule}\ \mathsf{accumulate}(D_1) \rightsquigarrow D_1$$

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

$$\textit{Funcon}\ \mathsf{collateral}(\rho^* : \mathsf{environments}^*) : \Rightarrow \mathsf{environments} \rightsquigarrow \mathsf{checkedmap\text{-}unite}(\rho^*)$$

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

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

Recurse

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

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

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

$\mathsf{recursive}(SI, D)$ 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).

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

$\mathsf{re\text{-}close}(M, D)$ 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

bind-to-forward-links(SI : sets(identifiers)) : $\Rightarrow$ maps(identifiers, links)
  $\rightsquigarrow$ map-unite(interleave-map(bind-value(given, fresh-link(maps)),
    set-elements(SI)))

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

Auxiliary Funcon

set-forward-links(M : maps(identifiers, links)) : $\Rightarrow$ null-type
  $\rightsquigarrow$ 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.