

Funcons-beta: Storing *

The PLaNCompS Project

Storing.cbs | PLAIN | PRETTY

OUTLINE

Storing
Stores
Simple variables
Structured variables

Storing

[*Datatype* locations
 Alias locs
 Type stores
 Entity store
Funcon initialise-storing
Funcon store-clear
Datatype variables
 Alias vars
 Funcon variable
 Alias var
 Funcon allocate-variable
 Alias alloc
 Funcon recycle-variables
 Alias recycle
 Funcon initialise-variable
 Alias init
 Funcon allocate-initialised-variable
 Alias alloc-init
 Funcon assign
 Funcon assigned
 Funcon current-value
 Funcon un-assign
 Funcon structural-assign
 Funcon structural-assigned]

Meta-variables $T, T' <: \text{values}$

*Suggestions for improvement: plancomps@gmail.com.
Reports of issues: <https://github.com/plancomps/CBS-beta/issues>.

Stores

Type `locations` \rightsquigarrow `atoms`

Alias `locs` = `locations`

A storage location is represented by an atom.

Type `stores` \rightsquigarrow `maps(locations, values?)`

The domain of a store is the set of currently allocated locations. Mapping a location to `()` models the absence of its stored value; removing it from the store allows it to be re-allocated.

Entity $\langle _ , \text{store}(_ : \text{stores}) \rangle \longrightarrow \langle _ , \text{store}(_ : \text{stores}) \rangle$

The current store is a mutable entity. A transition $\langle X, \text{store}(\sigma) \rangle \longrightarrow \langle X', \text{store}(\sigma') \rangle$ models a step from X to X' where the difference between σ and σ' (if any) corresponds to storage effects.

Funcon `store-clear` : \Rightarrow `null-type`

Rule $\langle \text{store-clear}, \text{store}(_) \rangle \longrightarrow \langle \text{null-value}, \text{store}(\text{map}(_)) \rangle$

`store-clear` ensures the store is empty.

Funcon `initialise-storing`($X : \Rightarrow T$) : $\Rightarrow T$

\rightsquigarrow `sequential`(

`store-clear`,

`initialise-giving(initialise-generating(X))`)

Alias `init-storing` = `initialise-storing`

`initialise-storing`(X) ensures that the entities used by the funcons for storing are properly initialised.

Simple variables Simple variables may store primitive or structured values. The type of values stored by a variable is fixed when it is allocated. For instance, `allocate-variable(integers)` allocates a simple integer variable, and `allocate-variable(vectors(integers))` allocates a structured variable for storing vectors of integers, which can be updated only monolithically.

Datatype `variables` ::= `variable`($_ : \text{locations}, _ : \text{value-types}$)

Alias `vars` = `variables`

Alias `var` = `variable`

`variables` is the type of simple variables that can store values of a particular type.

`variable`(L, T) constructs a simple variable for storing values of type T at location L . Variables at different locations are independent.

Note that `variables` is a subtype of `datatype-values`.

Funcon `allocate-variable`($T : \text{types}$) : \Rightarrow `variables`

Alias `alloc` = `allocate-variable`

`allocate-variable`(T) gives a simple variable whose location is not in the current store. Subsequent uses of `allocate-variable`(T') give independent variables, except after `recycle-variables`(V, \dots) or `store-clear`.

$$\text{Rule } \frac{\langle \text{use-atom-not-in}(\text{dom}(\sigma)), \text{store}(\sigma) \rangle \longrightarrow \langle L, \text{store}(\sigma') \rangle \quad \text{map-override}(\{L \mapsto (\)\}, \sigma') \rightsquigarrow \sigma''}{\langle \text{allocate-variable}(T : \text{types}), \text{store}(\sigma) \rangle \longrightarrow \langle \text{variable}(L, T), \text{store}(\sigma'') \rangle}$$

Funcon `recycle-variables(_ : variables+) : ⇒ null-type`

Alias `recycle = recycle-variables`

`recycle-variables(Var, ...)` removes the locations of `Var, ...`, from the current store, so that they may subsequently be re-allocated.

$$\text{Rule } \frac{\text{is-in-set}(L, \text{dom}(\sigma)) == \text{true}}{\langle \text{recycle-variables}(\text{variable}(L : \text{locations}, T : \text{types})), \text{store}(\sigma) \rangle \longrightarrow \langle \text{null-value}, \text{store}(\text{map-delete}(\sigma, \{L\})) \rangle}$$

$$\text{Rule } \frac{\text{is-in-set}(L, \text{dom}(\sigma)) == \text{false}}{\langle \text{recycle-variables}(\text{variable}(L : \text{locations}, T : \text{types})), \text{store}(\sigma) \rangle \longrightarrow \langle \text{fail}, \text{store}(\sigma) \rangle}$$

$$\text{Rule } \text{recycle-variables}(Var : \text{variables}, Var^+ : \text{variables}^+) \rightsquigarrow \text{sequential}(\text{recycle-variables}(Var), \text{recycle-variables}(Var^+))$$

Funcon `initialise-variable(_ : variables, _ : values) : ⇒ null-type`

Alias `init = initialise-variable`

`initialise-variable(Var, Val)` assigns `Val` as the initial value of `Var`, and gives `null-value`. If `Var` already has an assigned value, it fails.

$$\text{Rule } \frac{\text{and}(\text{is-in-set}(L, \text{dom}(\sigma)), \text{not is-value}(\text{map-lookup}(\sigma, L)), \text{is-in-type}(Val, T)) == \text{true}}{\langle \text{initialise-variable}(\text{variable}(L : \text{locations}, T : \text{types}), Val : \text{values}), \text{store}(\sigma) \rangle \longrightarrow \langle \text{null-value}, \text{store}(\text{map-override}(\{L \mapsto Val\}, \sigma)) \rangle}$$

$$\text{Rule } \frac{\text{and}(\text{is-in-set}(L, \text{dom}(\sigma)), \text{not is-value}(\text{map-lookup}(\sigma, L)), \text{is-in-type}(Val, T)) == \text{false}}{\langle \text{initialise-variable}(\text{variable}(L : \text{locations}, T : \text{types}), Val : \text{values}), \text{store}(\sigma) \rangle \longrightarrow \langle \text{fail}, \text{store}(\sigma) \rangle}$$

Funcon `allocate-initialised-variable(T, Val : T) : ⇒ variables`

$$\rightsquigarrow \text{give}(\text{allocate-variable}(T), \text{sequential}(\text{initialise-variable}(\text{given}, Val), \text{given}))$$

Alias `alloc-init = allocate-initialised-variable`

`allocate-initialised-variable(T, Val)` allocates a simple variable for storing values of type `T`, initialises its value to `Val`, and returns the variable.

Funcon `assign(_ : variables, _ : values) : ⇒ null-type`

`assign(Var, Val)` assigns the value `Val` to the variable `Var`, provided that `Var` was allocated with a type that contains `Val`.

Rule
$$\frac{\text{and}(\text{is-in-set}(L, \text{dom}(\sigma)), \text{is-in-type}(Val, T)) == \text{true}}{\langle \text{assign}(\text{variable}(L : \text{locations}, T : \text{types}), Val : \text{values}), \text{store}(\sigma) \rangle \longrightarrow \langle \text{null-value}, \text{store}(\text{map-override}(\{L \mapsto Val\}, \sigma)) \rangle}$$

Rule
$$\frac{\text{and}(\text{is-in-set}(L, \text{dom}(\sigma)), \text{is-in-type}(Val, T)) == \text{false}}{\langle \text{assign}(\text{variable}(L : \text{locations}, T : \text{types}), Val : \text{values}), \text{store}(\sigma) \rangle \longrightarrow \langle \text{fail}, \text{store}(\sigma) \rangle}$$

Funcon `assigned(_ : variables) : ⇒ values`

`assigned(Var)` gives the value assigned to the variable `Var`, failing if no value is currently assigned.

Rule
$$\frac{\text{map-lookup}(\sigma, L) \rightsquigarrow (Val : \text{values})}{\langle \text{assigned}(\text{variable}(L : \text{locations}, T : \text{types})), \text{store}(\sigma) \rangle \longrightarrow \langle Val, \text{store}(\sigma) \rangle}$$

Rule
$$\frac{\text{map-lookup}(\sigma, L) == ()}{\langle \text{assigned}(\text{variable}(L : \text{locations}, T : \text{types})), \text{store}(\sigma) \rangle \longrightarrow \langle \text{fail}, \text{store}(\sigma) \rangle}$$

Funcon `current-value(_ : values) : ⇒ values`

`current-value(V)` gives the same result as `assigned(V)` when `V` is a simple variable, and otherwise gives `V`.

It represents implicit dereferencing of a value that might be a variable.

Rule `current-value(Var : variables) ∼ assigned(Var)`

Rule `current-value(U : ∼ variables) ∼ U`

Funcon `un-assign(_ : variables) : ⇒ null-type`

`un-assign(Var)` remove the value assigned to the variable `Var`.

Rule
$$\frac{\text{is-in-set}(L, \text{dom}(\sigma)) == \text{true}}{\langle \text{un-assign}(\text{variable}(L : \text{locations}, T : \text{types})), \text{store}(\sigma) \rangle \longrightarrow \langle \text{null-value}, \text{store}(\text{map-override}(\{L \mapsto ()\}, \sigma)) \rangle}$$

Rule
$$\frac{\text{is-in-set}(L, \text{dom}(\sigma)) == \text{false}}{\langle \text{un-assign}(\text{variable}(L : \text{locations}, T : \text{types})), \text{store}(\sigma) \rangle \longrightarrow \langle \text{fail}, \text{store}(\sigma) \rangle}$$

Structured variables Structured variables are structured values where some components are simple variables. Such component variables can be selected using the same funcons as for selecting components of structured values.

Structured variables containing both simple variables and values correspond to hybrid structures where particular components are mutable.

All datatypes (except for abstractions) can be used to form structured variables. So can maps, but not sets or multisets.

Structural generalisations of `assign(Var, Val)` and `assigned(Var)` access all the simple variables contained in a structured variable. Assignment requires each component value of a hybrid structured variable to be equal to the corresponding component of the structured value.

Funcon `structural-assign(_ : values, _ : values) : ⇒ null-type`

`structural-assign(V1, V2)` takes a (potentially) structured variable V₁ and a (potentially) structured value V₂. Provided that the structure and all non-variable values in V₁ match the structure and corresponding values of V₂, all the simple variables in V₁ are assigned the corresponding values of V₂; otherwise the assignment fails.

Rule `structural-assign(V1 : variables, V2 : values) ⇔`
`assign(V1, V2)`
`V1 : ~ (variables)`
`V1 ⇔ datatype-value(I1 : identifiers, V1* : values*)`
`V2 ⇔ datatype-value(I2 : identifiers, V2* : values*)`

Rule `structural-assign(V1 : datatype-values, V2 : datatype-values) ⇔`
`sequential(`
`check-true(is-equal(I1, I2)),`
`effect(`
`tuple(`
`interleave-map(`
`structural-assign(tuple-elements(given)),`
`tuple-zip(tuple(V1*), tuple(V2*))))),`
`null-value)`

Note that simple variables are datatype values.

Rule $\frac{\text{dom}(M_1) == \{ \}}{\text{structural-assign}(M_1 : \text{maps}(-, -), M_2 : \text{maps}(-, -)) \rightsquigarrow \text{check-true}(\text{is-equal}(\text{dom}(M_2), \{ \}))}$

Rule $\frac{\text{some-element}(\text{dom}(M_1)) \rightsquigarrow K}{\text{structural-assign}(M_1 : \text{maps}(-, -), M_2 : \text{maps}(-, -)) \rightsquigarrow \text{sequential}(\text{check-true}(\text{is-in-set}(K, \text{dom}(M_2))), \text{structural-assign}(\text{map-lookup}(M_1, K), \text{map-lookup}(M_2, K)), \text{structural-assign}(\text{map-delete}(M_1, \{K\}), \text{map-delete}(M_2, \{K\}))})}$

Rule $\frac{V_1 : \sim(\text{datatype-values} \mid \text{maps}(-, -))}{\text{structural-assign}(V_1 : \text{values}, V_2 : \text{values}) \rightsquigarrow \text{check-true}(\text{is-equal}(V_1, V_2))}$

Funcon `structural-assigned(_ : values) : ⇒ values`

`structural-assigned(V)` takes a (potentially) structured variable V, and computes the value of V with all simple variables in V replaced by their assigned values, failing if any of them do not have assigned values.

When V is just a simple variable or a (possibly structured) value with no component variables, `structural-assigned(V)` gives the same result as `current-value(V)`.

Rule $\text{structural-assigned}(Var : \text{variables}) \rightsquigarrow \text{assigned}(Var)$

Rule $\frac{V : \sim(\text{variables}) \quad V \rightsquigarrow \text{datatype-value}(I : \text{identifiers}, V^* : \text{values}^*)}{\text{structural-assigned}(V : \text{datatype-values}) \rightsquigarrow \text{datatype-value}(I, \text{interleave-map}(\text{structural-assigned}(\text{given}), V^*))}$

Note that simple variables are datatype values.

Rule $\text{structural-assigned}(M : \text{maps}(-, -)) \rightsquigarrow \text{map}(\text{interleave-map}(\text{structural-assigned}(\text{given}), \text{map-elements}(M)))$

Rule $\frac{U : \sim(\text{datatype-values} \mid \text{maps}(-, -))}{\text{structural-assigned}(U : \text{values}) \rightsquigarrow U}$