

Funcons-beta: Giving *

The PPlanCompS Project

Giving.cbs | PLAIN | PRETTY

OUTLINE

Giving
Mapping
Filtering
Folding

Giving

[*Entity* **given-value**
Funcon **initialise-giving**
Funcon **give**
Funcon **given**
Funcon **no-given**
Funcon **left-to-right-map**
Funcon **interleave-map**
Funcon **left-to-right-repeat**
Funcon **interleave-repeat**
Funcon **left-to-right-filter**
Funcon **interleave-filter**
Funcon **fold-left**
Funcon **fold-right**]

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

Entity **given-value**($_ : \text{values?}$) $\vdash _ \longrightarrow _$

The given-value entity allows a computation to refer to a single previously-computed $V : \text{values}$. The given value () represents the absence of a current given value.

Funcon **initialise-giving**($X : () \Rightarrow T'$) : $() \Rightarrow T'$
 $\rightsquigarrow \text{no-given}(X)$

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

Funcon **give**($_ : T, _ : T \Rightarrow T'$) : $\Rightarrow T'$

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

`give(X, Y)` executes X , possibly referring to the current `given` value, to compute a value V . It then executes Y with V as the `given` value, to compute the result.

$$\begin{array}{l} \text{Rule} \quad \frac{\text{given-value}(V) \vdash Y \longrightarrow Y'}{\text{given-value}(_) \vdash \text{give}(V : T, Y) \longrightarrow \text{give}(V, Y')} \\ \text{Rule} \quad \text{give}(_ : T, W : T') \rightsquigarrow W \end{array}$$

$$\text{Funcon} \quad \text{given} : T \Rightarrow T$$

`given` refers to the current given value.

$$\begin{array}{l} \text{Rule} \quad \text{given-value}(V : \text{values}) \vdash \text{given} \longrightarrow V \\ \text{Rule} \quad \text{given-value}(_) \vdash \text{given} \longrightarrow \text{fail} \end{array}$$

$$\text{Funcon} \quad \text{no-given}(_ : (_) \Rightarrow T') : (_) \Rightarrow T'$$

`no-given(X)` computes X without references to the current given value.

$$\begin{array}{l} \text{Rule} \quad \frac{\text{given-value}(_) \vdash X \longrightarrow X'}{\text{given-value}(_) \vdash \text{no-given}(X) \longrightarrow \text{no-given}(X')} \\ \text{Rule} \quad \text{no-given}(U : T') \rightsquigarrow U \end{array}$$

Mapping Maps on collection values can be expressed directly, e.g., `list(left-to-right-map(F, list-elements(L)))`.

$$\text{Funcon} \quad \text{left-to-right-map}(_ : T \Rightarrow T', _ : (T)^*) : \Rightarrow (T')^*$$

`left-to-right-map(F, V*)` computes F for each value in V^* from left to right, returning the sequence of resulting values.

$$\begin{array}{l} \text{Rule} \quad \text{left-to-right-map}(F, V : T, V^* : (T)^*) \rightsquigarrow \\ \quad \text{left-to-right}(\text{give}(V, F), \text{left-to-right-map}(F, V^*)) \\ \text{Rule} \quad \text{left-to-right-map}(_, (_)) \rightsquigarrow (_) \end{array}$$

$$\text{Funcon} \quad \text{interleave-map}(_ : T \Rightarrow T', _ : (T)^*) : \Rightarrow (T')^*$$

`interleave-map(F, V*)` computes F for each value in V^* interleaved, returning the sequence of resulting values.

$$\begin{array}{l} \text{Rule} \quad \text{interleave-map}(F, V : T, V^* : (T)^*) \rightsquigarrow \\ \quad \text{interleave}(\text{give}(V, F), \text{interleave-map}(F, V^*)) \\ \text{Rule} \quad \text{interleave-map}(_, (_)) \rightsquigarrow (_) \end{array}$$

$$\text{Funcon} \quad \text{left-to-right-repeat}(_ : \text{integers} \Rightarrow T', _ : \text{integers}, _ : \text{integers}) : \Rightarrow (T')^*$$

`left-to-right-repeat(F, M, N)` computes F for each value from M to N sequentially, returning the sequence of resulting values.

$$\begin{array}{l} \text{Rule} \quad \frac{\text{is-less-or-equal}(M, N) == \text{true}}{\text{left-to-right-repeat}(F, M : \text{integers}, N : \text{integers}) \rightsquigarrow \\ \quad \text{left-to-right}(\text{give}(M, F), \text{left-to-right-repeat}(F, \text{int-add}(M, 1), N))} \\ \text{Rule} \quad \frac{\text{is-less-or-equal}(M, N) == \text{false}}{\text{left-to-right-repeat}(_, M : \text{integers}, N : \text{integers}) \rightsquigarrow (_)} \end{array}$$

Funcon `interleave-repeat`($_ : \text{integers} \Rightarrow T', _ : \text{integers}, _ : \text{integers}$) : $\Rightarrow (T')^*$

`interleave-repeat`(F, M, N) computes F for each value from M to N interleaved, returning the sequence of resulting values.

Rule
$$\frac{\text{is-less-or-equal}(M, N) == \text{true}}{\text{interleave-repeat}(F, M : \text{integers}, N : \text{integers}) \rightsquigarrow \text{interleave}(\text{give}(M, F), \text{interleave-repeat}(F, \text{int-add}(M, 1), N))}$$

Rule
$$\frac{\text{is-less-or-equal}(M, N) == \text{false}}{\text{interleave-repeat}(_, M : \text{integers}, N : \text{integers}) \rightsquigarrow ()}$$

Filtering Filters on collections of values can be expressed directly, e.g., `list(left-to-right-filter(P , list-elements(L)))` to filter a list L .

Funcon `left-to-right-filter`($_ : T \Rightarrow \text{booleans}, _ : (T)^*$) : $\Rightarrow (T)^*$

`left-to-right-filter`(P, V^*) computes P for each value in V^* from left to right, returning the sequence of argument values for which the result is `true`.

Rule
$$\text{left-to-right-filter}(P, V : T, V^* : (T)^*) \rightsquigarrow \text{left-to-right}(\text{when-true}(\text{give}(V, P), V), \text{left-to-right-filter}(P, V^*))$$

Rule
$$\text{left-to-right-filter}(_) \rightsquigarrow ()$$

Funcon `interleave-filter`($_ : T \Rightarrow \text{booleans}, _ : (T)^*$) : $\Rightarrow (T)^*$

`interleave-filter`(P, V^*) computes P for each value in V^* interleaved, returning the sequence of argument values for which the result is `true`.

Rule
$$\text{interleave-filter}(P, V : T, V^* : (T)^*) \rightsquigarrow \text{interleave}(\text{when-true}(\text{give}(V, P), V), \text{interleave-filter}(P, V^*))$$

Rule
$$\text{interleave-filter}(_) \rightsquigarrow ()$$

Folding

Funcon `fold-left`($_ : \text{tuples}(T, T') \Rightarrow T, _ : T, _ : (T')^*$) : $\Rightarrow T$

`fold-left`(F, A, V^*) reduces a sequence V^* to a single value by folding it from the left, using A as the initial accumulator value, and iteratively updating the accumulator by giving F the pair of the accumulator value and the first of the remaining arguments.

Rule
$$\text{fold-left}(_, A : T, ()) \rightsquigarrow A$$

Rule
$$\text{fold-left}(F, A : T, V : T', V^* : (T')^*) \rightsquigarrow \text{fold-left}(F, \text{give}(\text{tuple}(A, V), F), V^*)$$

Funcon `fold-right`($_ : \text{tuples}(T, T') \Rightarrow T', _ : T', _ : (T)^*$) : $\Rightarrow T'$

`fold-right`(F, A, V^*) reduces a sequence V^* to a single value by folding it from the right, using A as the initial accumulator value, and iteratively updating the accumulator by giving F the pair of the the last of the remaining arguments and the accumulator value.

Rule
$$\text{fold-right}(_, A : T', ()) \rightsquigarrow A$$

Rule
$$\text{fold-right}(F, A : T', V^* : (T)^*, V : T) \rightsquigarrow \text{give}(\text{tuple}(V, \text{fold-right}(F, A, V^*)), F)$$