Funcons-beta: Giving *

The PLanCompS Project

`Giving.cbs` | PLAIN | PRETTY

OUTLINE

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 $T, T' <:$ values
$T^? <:$ values$^?$

Entity given-value($\_ :$ values?) $\vdash \_ \longrightarrow \_$

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

Funcon initialise-giving($X : (\,) \Rightarrow T'$) $: (\,) \Rightarrow T'$
$\rightsquigarrow$ 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`.

$\mathsf{give}(X, Y)$ 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.

$$\textit{Rule}\quad \frac{\mathsf{given\text{-}value}(V) \vdash Y \longrightarrow Y'}{\mathsf{given\text{-}value}(\_?) \vdash \mathsf{give}(V : T, Y) \longrightarrow \mathsf{give}(V, Y')}$$

$$\textit{Rule}\quad \mathsf{give}(\_ : T, W : T') \rightsquigarrow W$$

$$\textit{Funcon}\quad \mathsf{given} : T \Rightarrow T$$

given refers to the current given value.

$$\textit{Rule}\quad \mathsf{given\text{-}value}(V : \mathsf{values}) \vdash \mathsf{given} \longrightarrow V$$

$$\textit{Rule}\quad \mathsf{given\text{-}value}(\ ) \vdash \mathsf{given} \longrightarrow \mathsf{fail}$$

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

$\mathsf{no\text{-}given}(X)$ computes X without references to the current given value.

$$\textit{Rule}\quad \frac{\mathsf{given\text{-}value}(\ ) \vdash X \longrightarrow X'}{\mathsf{given\text{-}value}(\_?) \vdash \mathsf{no\text{-}given}(X) \longrightarrow \mathsf{no\text{-}given}(X')}$$

$$\textit{Rule}\quad \mathsf{no\text{-}given}(U : T') \rightsquigarrow U$$

Mapping Maps on collection values can be expressed directly, e.g., $\mathsf{list}(\mathsf{left\text{-}to\text{-}right\text{-}map}(F, \mathsf{list\text{-}elements}(L)))$.

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

$\mathsf{left\text{-}to\text{-}right\text{-}map}(F, V^*)$ computes F for each value in V^* from left to right, returning the sequence of resulting values.

$$\textit{Rule}\quad \mathsf{left\text{-}to\text{-}right\text{-}map}(F, V : T, V^* : (T)^*) \rightsquigarrow \mathsf{left\text{-}to\text{-}right}(\mathsf{give}(V, F), \mathsf{left\text{-}to\text{-}right\text{-}map}(F, V^*))$$

$$\textit{Rule}\quad \mathsf{left\text{-}to\text{-}right\text{-}map}(\_, (\ )) \rightsquigarrow (\ )$$

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

$\mathsf{interleave\text{-}map}(F, V^*)$ computes F for each value in V^* interleaved, returning the sequence of resulting values.

$$\textit{Rule}\quad \mathsf{interleave\text{-}map}(F, V : T, V^* : (T)^*) \rightsquigarrow \mathsf{interleave}(\mathsf{give}(V, F), \mathsf{interleave\text{-}map}(F, V^*))$$

$$\textit{Rule}\quad \mathsf{interleave\text{-}map}(\_, (\ )) \rightsquigarrow (\ )$$

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

$\mathsf{left\text{-}to\text{-}right\text{-}repeat}(F, M, N)$ computes F for each value from M to N sequentially, returning the sequence of resulting values.

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

$$\textit{Rule}\quad \frac{\mathsf{is\text{-}less\text{-}or\text{-}equal}(M, N) == \mathsf{false}}{\mathsf{left\text{-}to\text{-}right\text{-}repeat}(\_, M : \mathsf{integers}, N : \mathsf{integers}) \rightsquigarrow (\ )}$$

Funcon interleave-repeat(_ : integers ⇒ T', _ : integers, _ : integers) : ⇒(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 ⇒ booleans, _ : (T)*) : ⇒(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 left-to-right-filter($P, V : T, V^* : (T)^*$) ⇝
left-to-right(when-true(give(V, P), V), left-to-right-filter(P, V^*))
Rule left-to-right-filter(_) ⇝ ( )

Funcon interleave-filter(_ : T ⇒ booleans, _ : (T)*) : ⇒(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 interleave-filter($P, V : T, V^* : (T)^*$) ⇝
interleave(when-true(give(V, P), V), interleave-filter(P, V^*))
Rule interleave-filter(_) ⇝ ( )

Folding

Funcon fold-left(_ : tuples(T, T') ⇒ T, _ : T, _ : (T')*) : ⇒ 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 fold-left(_, $A : T$, ( )) ⇝ A
Rule fold-left($F, A : T, V : T', V^* : (T')^*$) ⇝ fold-left($F$, give(tuple($A, V$), F), V^*)

Funcon fold-right(_ : tuples(T, T') ⇒ T', _ : T', _ : (T)*) : ⇒ 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 fold-right(_, $A : T'$, ( )) ⇝ A
Rule fold-right($F, A : T', V^* : (T)^*, V : T$) ⇝ give(tuple($V$, fold-right($F, A, V^*$)), F)