Funcons-beta: Giving

The PLanCompS Project

`Funcons-beta/Computations/Normal/Giving/Giving.cbs`*

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$^?$) ⊢ _ ⟶ _

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 : ( ) ⇒ T') : ( ) ⇒ T'
⇝ no-given(X)

*Suggestions for improvement: `plancomps@gmail.com`.
Issues: `https://github.com/plancomps/CBS-beta/issues`.

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

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

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.

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

Rule give($\_ : T, W : T') \rightsquigarrow W$

Funcon given $: T \Rightarrow T$

given refers to the current given value.

Rule given-value(V : values) $\vdash$ given $\longrightarrow V$

Rule given-value( ) $\vdash$ given $\longrightarrow$ fail

Funcon no-given($\_ : (\ ) \Rightarrow T') : (\ ) \Rightarrow T'$

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

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

Rule no-given($U : T') \rightsquigarrow U$

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

Funcon 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.

Rule left-to-right-map($F, V : T, V^* : (T)^*) \rightsquigarrow$ left-to-right(give(V, F), left-to-right-map(F, V^*))

Rule left-to-right-map($\_, (\ )) \rightsquigarrow (\ )$

Funcon 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.

Rule interleave-map(F, $V : T$, $V^* : (T)^*$) $\rightsquigarrow$ interleave(give(V, F), interleave-map(F, V^*))

Rule interleave-map(_, ( )) $\rightsquigarrow$ ( )

Funcon left-to-right-repeat(_ : integers $\Rightarrow T'$, _ : integers, _ : 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.

Rule $\dfrac{\text{is-less-or-equal}(M, N) == \text{true}}{\text{left-to-right-repeat}(F, M : \text{integers}, N : \text{integers}) \rightsquigarrow \text{left-to-right}(\text{give}(M, F), \text{left-to-right-repeat}(F, \text{int-a}}$

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

Funcon interleave-repeat(_ : integers $\Rightarrow T'$, _ : integers, _ : 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 $\dfrac{\text{is-less-or-equal}(M, N) == \text{true}}{\text{interleave-repeat}(F, M : \text{integers}, N : \text{integers}) \rightsquigarrow \text{interleave}(\text{give}(M, F), \text{interleave-repeat}(F, \text{int-add}(M,}$

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

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(_ : $T \Rightarrow$ 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 left-to-right-filter(P, $V : T$, $V^* : (T)^*$) $\rightsquigarrow$ left-to-right(when-true(give(V, P), V), left-to-right-filter(P, V

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

Funcon interleave-filter(_ : $T \Rightarrow$ 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 interleave-filter(P, $V : T$, $V^* : (T)^*$) $\rightsquigarrow$ interleave(when-true(give(V, P), V), interleave-filter(P, V^*))

Rule interleave-filter(_) $\rightsquigarrow$ ( )

Folding

Funcon fold-left(_ : 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 fold-left(_, $A : T$, ( )) $\rightsquigarrow A$

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

Funcon fold-right(_ : 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 fold-right(_, $A : T'$, ( )) $\rightsquigarrow A$

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