

# Funcons-beta: Flowing \*

The P<sub>Plan</sub>CompS Project

Flowing.cbs | PLAIN | PRETTY

## OUTLINE

Flowing  
  Sequencing  
  Choosing  
  Iterating  
  Interleaving

---

## Flowing

[ *Funcon* left-to-right  
  *Alias* l-to-r  
  *Funcon* right-to-left  
  *Alias* r-to-l  
  *Funcon* sequential  
  *Alias* seq  
  *Funcon* effect  
  *Funcon* choice  
  *Funcon* if-true-else  
  *Alias* if-else  
  *Funcon* while-true  
  *Alias* while  
  *Funcon* do-while-true  
  *Alias* do-while  
  *Funcon* interleave  
*Datatype* yielding  
  *Funcon* signal  
  *Funcon* yielded  
  *Funcon* yield  
  *Funcon* yield-on-value  
  *Funcon* yield-on-abrupt  
  *Funcon* atomic ]

*Meta-variables*  $T <:$  values  
                   $T^* <:$  values\*

---

\*Suggestions for improvement: [plancomps@gmail.com](mailto:plancomps@gmail.com).  
Reports of issues: <https://github.com/plancomps/CBS-beta/issues>.

## Sequencing

*Funcon* **left-to-right**( $\_ : (\Rightarrow (T)^*)^*$ ) :  $\Rightarrow (T)^*$

*Alias* **l-to-r** = **left-to-right**

**left-to-right**( $\dots$ ) computes its arguments sequentially, from left to right, and gives the resulting sequence of values, provided all terminate normally. For example, **integer-add**( $X, Y$ ) may interleave the computations of  $X$  and  $Y$ , whereas **integer-add left-to-right**( $X, Y$ ) always computes  $X$  before  $Y$ .

When each argument of **left-to-right**( $\dots$ ) computes a single value, the type of the result is the same as that of the argument sequence. For instance, when  $X : T$  and  $Y : T'$ , the result of **left-to-right**( $X, Y$ ) is of type  $(T, T')$ . The only effect of wrapping an argument sequence in **left-to-right**( $\dots$ ) is to ensure that when the arguments are to be evaluated, it is done in the specified order.

*Rule* 
$$\frac{Y \longrightarrow Y'}{\text{left-to-right}(V^* : (T)^*, Y, Z^*) \longrightarrow \text{left-to-right}(V^*, Y', Z^*)}$$

*Rule* **left-to-right**( $V^* : (T)^*$ )  $\rightsquigarrow V^*$

*Funcon* **right-to-left**( $\_ : (\Rightarrow (T)^*)^*$ ) :  $\Rightarrow (T)^*$

*Alias* **r-to-l** = **right-to-left**

**right-to-left**( $\dots$ ) computes its arguments sequentially, from right to left, and gives the resulting sequence of values, provided all terminate normally.

Note that **right-to-left**( $X^*$ ) and **reverse left-to-right reverse**( $X^*$ ) are not equivalent: **reverse**( $X^*$ ) interleaves the evaluation of  $X^*$ .

*Rule* 
$$\frac{Y \longrightarrow Y'}{\text{right-to-left}(X^*, Y, V^* : (T)^*) \longrightarrow \text{right-to-left}(X^*, Y', V^*)}$$

*Rule* **right-to-left**( $V^* : (T)^*$ )  $\rightsquigarrow V^*$

*Funcon* **sequential**( $\_ : (\Rightarrow \text{null-type})^*$ ,  $\_ : \Rightarrow T$ ) :  $\Rightarrow T$

*Alias* **seq** = **sequential**

**sequential**( $X, \dots$ ) computes its arguments in the given order. On normal termination, it returns the value of the last argument; the other arguments all compute **null-value**.

Binary **sequential**( $X, Y$ ) is associative, with unit **null-value**.

*Rule* 
$$\frac{X \longrightarrow X'}{\text{sequential}(X, Y^+) \longrightarrow \text{sequential}(X', Y^+)}$$

*Rule* **sequential**(**null-value**,  $Y^+$ )  $\rightsquigarrow \text{sequential}(Y^+)$

*Rule* **sequential**( $Y$ )  $\rightsquigarrow Y$

*Funcon* **effect**( $V^* : T^*$ ) :  $\Rightarrow \text{null-type}$   
 $\rightsquigarrow \text{null-value}$

**effect**( $\dots$ ) interleaves the computations of its arguments, then discards all the computed values.

## Choosing

*Funcon* **choice**( $\_ : (\Rightarrow T)^+$ ) :  $\Rightarrow T$

**choice**( $Y, \dots$ ) selects one of its arguments, then computes it. It is associative and commutative.

*Rule*  $\text{choice}(X^*, Y, Z^*) \rightsquigarrow Y$

*Funcon*  $\text{if-true-else}(\_ : \text{booleans}, \_ : \Rightarrow T, \_ : \Rightarrow T) : \Rightarrow T$

*Alias*  $\text{if-else} = \text{if-true-else}$

$\text{if-true-else}(B, X, Y)$  evaluates  $B$  to a Boolean value, then reduces to  $X$  or  $Y$ , depending on the value of  $B$ .

*Rule*  $\text{if-true-else}(\text{true}, X, Y) \rightsquigarrow X$

*Rule*  $\text{if-true-else}(\text{false}, X, Y) \rightsquigarrow Y$

## Iterating

*Funcon*  $\text{while-true}(B : \Rightarrow \text{booleans}, X : \Rightarrow \text{null-type}) : \Rightarrow \text{null-type}$

$\rightsquigarrow \text{if-true-else}(B, \text{sequential}(X, \text{while-true}(B, X)), \text{null-value})$

*Alias*  $\text{while} = \text{while-true}$

$\text{while-true}(B, X)$  evaluates  $B$  to a Boolean value. Depending on the value of  $B$ , it either executes  $X$  and iterates, or terminates normally.

The effect of abruptly breaking the iteration is obtained by the combination  $\text{handle-break}(\text{while-true}(B, X))$ , and that of abruptly continuing the iteration by  $\text{while-true}(B, \text{handle-continue}(X))$ .

*Funcon*  $\text{do-while-true}(X : \Rightarrow \text{null-type}, B : \Rightarrow \text{booleans}) : \Rightarrow \text{null-type}$

$\rightsquigarrow \text{sequential}(X, \text{if-true-else}(B, \text{do-while-true}(X, B), \text{null-value}))$

*Alias*  $\text{do-while} = \text{do-while-true}$

$\text{do-while-true}(X, B)$  is equivalent to  $\text{sequential}(X, \text{while-true}(B, X))$ .

## Interleaving

*Funcon*  $\text{interleave}(\_ : T^*) : \Rightarrow T^*$

$\text{interleave}(\dots)$  computes its arguments in any order, possibly interleaved, and returns the resulting sequence of values, provided all terminate normally. Fairness of interleaving is not required, so pure left-to-right computation is allowed.

$\text{atomic}(X)$  prevents interleaving in  $X$ , except after transitions that emit a  $\text{yielded}(\text{signal})$ .

*Rule*  $\text{interleave}(V^* : T^*) \rightsquigarrow V^*$

*Datatype*  $\text{yielding} ::= \text{signal}$

*Entity*  $\_ \xrightarrow{\text{yielded}(\_ \text{yielding}?)}$

$\text{yielded}(\text{signal})$  in a label on a transition allows interleaving at that point in the enclosing atomic computation.  $\text{yielded}()$  indicates interleaving at that point in an atomic computation is not allowed.

*Funcon*  $\text{yield} : \Rightarrow \text{null-type}$

$\rightsquigarrow \text{yield-on-value}(\text{null-value})$

Funcon  $\text{yield-on-value}(\_ : T) : \Rightarrow T$

$\text{yield-on-value}(X)$  allows interleaving in an enclosing atomic computation on normal termination of  $X$ .

Rule  $\text{yield-on-value}(V : T) \xrightarrow{\text{yielded}(\text{signal})} V$

Funcon  $\text{yield-on-abrupt}(\_ : \Rightarrow T) : \Rightarrow T$

$\text{yield-on-abrupt}(X)$  ensures that abrupt termination of  $X$  is propagated through an enclosing atomic computation.

Rule 
$$\frac{X \xrightarrow{\text{abrupt}(V:T), \text{yielded}(\_?)}}{X'} \quad \text{yield-on-abrupt}(X) \xrightarrow{\text{abrupt}(V), \text{yielded}(\text{signal})} \text{yield-on-abrupt}(X')$$

Rule 
$$\frac{X \xrightarrow{\text{abrupt}(\_)}}{X'} \quad \text{yield-on-abrupt}(X) \xrightarrow{\text{abrupt}(\_)} \text{yield-on-abrupt}(X')$$

Rule  $\text{yield-on-abrupt}(V : T) \rightsquigarrow V$

Funcon  $\text{atomic}(\_ : \Rightarrow T) : \Rightarrow T$

$\text{atomic}(X)$  computes  $X$ , but controls its potential interleaving with other computations: interleaving is only allowed following a transition of  $X$  that emits  $\text{yielded}(\text{signal})$ .

Rule 
$$\frac{X \xrightarrow{\text{yielded}(\_)}_1 X' \quad \text{atomic}(X') \xrightarrow{\text{yielded}(\_)}_2 X''}{\text{atomic}(X) \xrightarrow{\text{yielded}(\_)}_1; \xrightarrow{\text{yielded}(\_)}_2 X''}$$

Rule 
$$\frac{X \xrightarrow{\text{yielded}(\_)} V \quad V : T}{\text{atomic}(X) \xrightarrow{\text{yielded}(\_)} V}$$

Rule  $\text{atomic}(V : T) \rightsquigarrow V$

Rule 
$$\frac{X \xrightarrow{\text{yielded}(\text{signal})}}{X'} \quad \text{atomic}(X) \xrightarrow{\text{yielded}(\_)} \text{atomic}(X')$$