Funcons-beta: Flowing *

The PLanCompS Project

`Flowing.cbs` | PLAIN | PRETTY

OUTLINE

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`.
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$(\cdots)$ 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$(\cdots)$ 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$(\cdots)$ 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$(\cdots)$ 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($\_ : (\Rightarrow$ null-type$)^*, _ : \Rightarrow T) : \Rightarrow T$

Alias seq = sequential

sequential$(X, \cdots)$ 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^+) $\rightsquigarrow$ sequential(Y^+)

Rule sequential$(Y) \rightsquigarrow Y$

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

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

Choosing

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

choice$(Y, \cdots)$ 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 $\text{if-else} = \text{if-true-else}$

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

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 handle-break(while-true(B, X)), and that of abruptly continuing the iteration by while-true(B, 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 $\text{do-while} = \text{do-while-true}$

do-while-true(X, B) is equivalent to sequential(X, while-true(B, X)).

Interleaving

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

interleave($\cdots$) 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.

atomic(X) prevents interleaving in X, except after transitions that emit a yielded(signal).

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

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

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

yielded(signal) in a label on a transition allows interleaving at that point in the enclosing atomic computation. 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 $\text{yield-on-value}(\_ : T) : \Rightarrow T$

yield-on-value(X) allows interleaving in an enclosing atomic computation on normal termination of X.

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

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

yield-on-abrupt(X) ensures that abrupt termination of X is propagated through an enclosing atomic computation.

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

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

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

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

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

$$\text{Rule}\quad \frac{\begin{array}{l} X \xrightarrow{\text{yielded}(\ )}_1 X' \\ \text{atomic}(X') \xrightarrow{\text{yielded}(\ )}_2 X'' \end{array}}{\text{atomic}(X) \xrightarrow{\text{yielded}(\ )}_1 ; \xrightarrow{\text{yielded}(\ )}_2 X''}$$

$$\text{Rule}\quad \frac{\begin{array}{l} X \xrightarrow{\text{yielded}(\ )} V \\ V : T \end{array}}{\text{atomic}(X) \xrightarrow{\text{yielded}(\ )} V}$$

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

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