Unstable-Languages-beta: LD-Start [*]

The PLanCompS Project

`LD-Start.cbs` | PLAIN | PRETTY

OUTLINE

Language “`LD`”

[**1** Lexical constructs
2 Call-by-value lambda-calculus
3 Arithmetic and Boolean expressions
4 References and imperatives
5 Multithreading
6 Programs
A Disambiguation]

Lexical syntax:

Lexis X : id ::= (`'a'`–`'z'`) (`'a'`–`'z'` | `'0'`–`'9'`)*
N : int ::= (`'0'`–`'9'`)$^+$
keyword ::= `'do'` | `'else'` | `'fork'` | `'if'`
| `'in'` | `'join'` | `'lambda'` | `'let'`
| `'ref'` | `'spawn'` | `'then'` | `'while'`

Context-free syntax:

[*]Suggestions for improvement: `plancomps@gmail.com`.
Reports of issues: `https://github.com/plancomps/CBS-beta/issues`.

```
Syntax   E : exp  ::=  int
                     | id
                     | 'lambda' id '.' exp
                     | exp exp
                     | 'let' id '=' exp 'in' exp
                     | '(' exp ')'
                     | exp '+' exp
                     | exp '*' exp
                     | exp '/' exp
                     | exp '<=' exp
                     | exp '&&' exp
                     | 'if' exp 'then' exp 'else' exp
                     | 'ref' exp
                     | exp ':=' exp
                     | '!' exp
                     | exp ';' exp
                     | '(' ')'
                     | 'while' exp 'do' exp
                     | 'spawn' exp
                     | 'join' exp
```

Expression evaluation:

```
Type   Id-values
          ⇝ functions(values, values)
             | integers
             | booleans
             | variables
             | null-type
             | thread-ids
```

Semantics $\text{eval}[\![\ \_ : \text{exp}\ ]\!] : \Rightarrow \text{Id-values}$

1 Lexical constructs

Rule $\text{eval}[\![\ N\ ]\!] = \text{decimal}\ \text{“}N\text{”}$

Rule $\text{eval}[\![\ X\ ]\!] = \text{bound}\ \text{“}X\text{”}$

2 Call-by-value lambda-calculus

```
Rule  eval⟦ ‘lambda’ X ‘.’ E ⟧ =
        function closure
          scope(
            bind(“X”, given),
            eval⟦ E ⟧)
Rule  eval⟦ E_1 E_2 ⟧ =
        apply(eval⟦ E_1 ⟧, eval⟦ E_2 ⟧)
Rule  eval⟦ ‘let’ X ‘=’ E_1 ‘in’ E_2 ⟧ =
        scope(
          bind(“X”, eval⟦ E_1 ⟧),
          eval⟦ E_2 ⟧)
```

Desugaring (alternative to the above rule):

```
Rule  ⟦ ‘let’ X ‘=’ E_1 ‘in’ E_2 ⟧ : exp =
      ⟦ ‘(’ ‘lambda’ X ‘.’ E_2 ‘)’ ‘(’ E_1 ‘)’ ⟧

Rule  eval⟦ ‘(’ E ‘)’ ⟧ = eval⟦ E ⟧
```

3 Arithmetic and Boolean expressions

```
Rule  eval⟦ E_1 ‘+’ E_2 ⟧ =
        int-add(eval⟦ E_1 ⟧, eval⟦ E_2 ⟧)
Rule  eval⟦ E_1 ‘*’ E_2 ⟧ =
        int-mul(eval⟦ E_1 ⟧, eval⟦ E_2 ⟧)
Rule  eval⟦ E_1 ‘/’ E_2 ⟧ =
        checked int-div(eval⟦ E_1 ⟧, eval⟦ E_2 ⟧)
Rule  eval⟦ E_1 ‘<=’ E_2 ⟧ =
        is-less-or-equal l-to-r(eval⟦ E_1 ⟧, eval⟦ E_2 ⟧)
Rule  eval⟦ E_1 ‘&&’ E_2 ⟧ =
        if-true-else(eval⟦ E_1 ⟧, eval⟦ E_2 ⟧, false)
Rule  eval⟦ ‘if’ E_1 ‘then’ E_2 ‘else’ E_3 ⟧ =
        if-true-else(eval⟦ E_1 ⟧, eval⟦ E_2 ⟧, eval⟦ E_3 ⟧)
```

4 References and imperatives

```
Rule  eval⟦ ‘ref’ E ⟧ =
        allocate-initialised-variable(ld-values, eval⟦ E ⟧)
Rule  eval⟦ E_1 ‘:=’ E_2 ⟧ =
        assign(eval⟦ E_1 ⟧, eval⟦ E_2 ⟧)
Rule  eval⟦ ‘!’ E ⟧ = assigned(eval⟦ E ⟧)
Rule  eval⟦ E_1 ‘;’ E_2 ⟧ =
        sequential(effect(eval⟦ E_1 ⟧), eval⟦ E_2 ⟧)
Rule  eval⟦ ‘(’ ‘)’ ⟧ = null-value
Rule  eval⟦ ‘while’ E_1 ‘do’ E_2 ⟧ =
        while-true(eval⟦ E_1 ⟧, effect(eval⟦ E_2 ⟧))
```

5 Multithreading

N.B. The funcons for multithreading have not yet been fully validated, so they are defined in Unstable-Funcons-beta instead of Funcons-beta.

```
Rule   eval⟦ 'spawn' E ⟧ =
         thread-activate thread-joinable thunk closure eval⟦ E ⟧
Rule   eval⟦ 'join' E ⟧ = thread-join(eval⟦ E ⟧)
```

6 Programs

```
Syntax   START : start  ::=  exp
```

```
Semantics   start⟦ _ : start ⟧ : ⇒ values
     Rule   start⟦ E ⟧ =
              initialise-binding
                initialise-storing
                  finalise-failing
                    multithread
                      eval⟦ E ⟧
```