Unstable-Languages-beta: LD-Start

The PLanCompS Project

`Unstable-Languages-beta/LD/LD-Start/LD-Start.cbs`[*]

Language "`LD`"

Lexical syntax:

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

$$
\begin{array}{rcl}
\textit{Lexis}\ X : \mathsf{id} & ::= & (\mathtt{a}-\mathtt{z})\ (\mathtt{a}-\mathtt{z} \mid 0-9)^{*} \\
N : \mathsf{int} & ::= & (0-9)^{+} \\
\mathsf{keyword} & ::= & \mathtt{do} \\
 & \mid & \mathtt{else} \\
 & \mid & \mathtt{fork} \\
 & \mid & \mathtt{if} \\
 & \mid & \mathtt{in} \\
 & \mid & \mathtt{join} \\
 & \mid & \mathtt{lambda} \\
 & \mid & \mathtt{let} \\
 & \mid & \mathtt{ref} \\
 & \mid & \mathtt{spawn} \\
 & \mid & \mathtt{then} \\
 & \mid & \mathtt{while}
\end{array}
$$

Context-free syntax:

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 ld-values $\rightsquigarrow$ functions(values, values) | integers | booleans | variables | null-type | thread-ids

Semantics eval$[\![$ _ : exp $]\!]$: $\Rightarrow$ ld-values

1 Lexical constructs

Rule eval$[\![$ N $]\!]$ =
    decimal "N"

Rule eval$[\![$ X $]\!]$ =
    bound "X"

2 Call-by-value lambda-calculus

```
Rule eval⟦ lambda X . E ⟧ =
        function closure scope(bind("X",
                      given),
                   eval⟦ E ⟧)
Rule eval⟦ E₁ E₂ ⟧ =
        apply(eval⟦ E₁ ⟧,
          eval⟦ E₂ ⟧)
Rule eval⟦ let X = E₁ in E₂ ⟧ =
        scope(bind("X",
             eval⟦ E₁ ⟧),
          eval⟦ E₂ ⟧)
```

Desugaring (alternative to the above rule):

```
Rule ⟦ let X = E₁ in E₂ ⟧ : exp =
     ⟦ ( lambda X . E₂ ) ( E₁ ) ⟧

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* ⟧