Languages-beta: OC-L-12-Core-Library [*]

The PLanCompS Project

`OC-L-12-Core-Library.cbs` | PLAIN | PRETTY

OUTLINE

Language "`OCaml Light`"

12 Core library

[*Funcon* ocaml-light-core-library
Funcon ocaml-light-match-failure
Funcon ocaml-light-is-structurally-equal
Funcon ocaml-light-to-string
Funcon ocaml-light-define-and-display
Funcon ocaml-light-evaluate-and-display]

Meta-variables $R, S, S_1, S_2, S_3, T, U$ <: values
S^* <: values*
T^+ <: values$^+$

Abbreviations

The following funcons take computations X and return (curried) functions. X refers to a single function argument as arg, or to individual arguments of a curried function of several arguments as arg-1, arg-2, arg-3.

Auxiliary Funcon op-1($X : S \Rightarrow T$) : $\Rightarrow$ functions(S, T)
$\rightsquigarrow$ function abstraction X

Auxiliary Funcon op-2(X : tuples(S_1, S_2) $\Rightarrow T$) : $\Rightarrow$ functions(S_1, functions(S_2, T))
$\rightsquigarrow$ curry function abstraction X

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

Auxiliary Funcon op-3(X : tuples(S_1, S_2, S_3) $\Rightarrow$ T) : $\Rightarrow$ functions(S_1, functions(S_2, functions(S_3, T)))
$\rightsquigarrow$ function abstraction(
curry partial-apply-first(function abstraction X, given))

Auxiliary Funcon partial-apply-first(F : functions(tuples(R, S, T^+), U), V : R)
: $\Rightarrow$ functions(tuples(S, T^+), U)
$\rightsquigarrow$ function abstraction(apply(F, tuple(V, tuple-elements given)))

partial-apply-first(F, V) provides V as the first argument to a function expecting a tuple of 3 or more arguments, returning a function expecting a tuple of one fewer arguments.

Auxiliary Funcon arg : $T \Rightarrow T$
$\rightsquigarrow$ given

Auxiliary Funcon arg-1 : tuples(S_1, S^*) $\Rightarrow$ S_1
$\rightsquigarrow$ checked index(1, tuple-elements given)

Auxiliary Funcon arg-2 : tuples(S_1, S_2, S^*) $\Rightarrow$ S_2
$\rightsquigarrow$ checked index(2, tuple-elements given)

Auxiliary Funcon arg-3 : tuples(S_1, S_2, S_3, S^*) $\Rightarrow$ S_3
$\rightsquigarrow$ checked index(3, tuple-elements given)

Library

The ocaml-light-core-library environment maps most of the names defined in OCaml Module Pervasives (the initially opened module) to funcon terms. See https://caml.inria.fr/pub/docs/manual-ocaml-4.06/core.html for further details and comments.

It also maps some other names defined in the OCaml Standard Libarary to funcon terms (to support tests using them without opening those modules).

```
Funcon   ocaml-light-core-library : ⇒ environments
    ⇝ { “Match_failure” ↦
              op-1(variant(“Match_failure”, arg)),
          “Invalid_argument” ↦
            op-1(variant(“Invalid_argument”, arg)),
          “Division_by_zero” ↦
            variant(“Division_by_zero”, tuple( )),
          “raise” ↦
            op-1(throw(arg)),
          “(=)” ↦
            op-2(ocaml-light-is-structurally-equal(arg-1, arg-2)),
          “(<>)” ↦
            op-2(not(ocaml-light-is-structurally-equal(arg-1, arg-2))),
          “(<)” ↦
            op-2(is-less(arg-1, arg-2)),
          “(>)” ↦
            op-2(is-greater(arg-1, arg-2)),
          “(<=)” ↦
            op-2(is-less-or-equal(arg-1, arg-2)),
          “(>=)” ↦
            op-2(is-greater-or-equal(arg-1, arg-2)),
          “min” ↦
            op-2(if-true-else(is-less(arg-1, arg-2), arg-1, arg-2)),
          “max” ↦
            op-2(if-true-else(is-greater(arg-1, arg-2), arg-1, arg-2)),
          “(==)” ↦
            op-2(
              if-true-else(
                and(is-in-type(arg-1, ground-values), is-in-type(arg-2, ground-values)),
                is-equal(arg-1, arg-2),
                throw(variant(“Invalid_argument”, “equal: functional value”)))),
          “(!=)” ↦
            op-2(
              if-true-else(
                and(is-in-type(arg-1, ground-values), is-in-type(arg-2, ground-values)),
                not is-equal(arg-1, arg-2),
                throw(variant(“Invalid_argument”, “equal: functional value”)))),
          “not” ↦
            op-1(not(arg)),
          “(~-)” ↦
            op-1(implemented-integer integer-negate(arg)),
          “(~+)” ↦
            op-1(implemented-integer arg),
          “succ” ↦
            op-1(implemented-integer integer-add(arg, 1)),
          “pred” ↦
            op-1(implemented-integer integer-subtract(arg, 1)),
          “(+)” ↦
            op-2(implemented-integer integer-add(arg-1, arg-2)),
          “(-)” ↦
            op-2(implemented-integer integer-subtract(arg-1, arg-2)),
          “(*)” ↦
```

Language-specific funcons

Exception values

Funcon ocaml-light-match-failure : ⇒ variants(tuples(strings, integers, integers))
⇝ variant(“`Match_failure`”, tuple(“”, 0, 0))

ocaml-light-match-failure gives a value to be thrown when a match fails. The variant value should consist of the source program text, line, and column, but these are currently not included in the translation of OCaml Light.

Funcon ocaml-light-assert-failure : ⇒ variants(tuples(strings, integers, integers))
⇝ variant(“`Assert_failure`”, tuple(“”, 0, 0))

ocaml-light-assert-failure gives a value to be thrown when an assertion fails. The variant value should consist of the source program text, line, and column, but these are currently not included in the translation of OCaml Light.

Structural equality

Funcon ocaml-light-is-structurally-equal(_ : implemented-values, _ : implemented-values)
: ⇒ booleans

ocaml-light-is-structurally-equal(V_1, V_2) is false whenever V_1 or V_2 contains a function. For vectors, it compares all their respective assigned values. It is equality on primitive values, and defined inductively on composite values.

Unit Type

Rule ocaml-light-is-structurally-equal(null-value, null-value) ⇝ true

Booleans

Rule ocaml-light-is-structurally-equal(B_1 : booleans, B_2 : booleans) ⇝ is-equal(B_1, B_2)

Integers

Rule ocaml-light-is-structurally-equal(I_1 : implemented-integers, I_2 : implemented-integers) ⇝
is-equal(I_1, I_2)

Floats

Rule ocaml-light-is-structurally-equal(F_1 : implemented-floats, F_2 : implemented-floats) ⇝
is-equal(F_1, F_2)

Characters

Rule ocaml-light-is-structurally-equal(C_1 : implemented-characters, C_2 : implemented-characters) ⇝
is-equal(C_1, C_2)

Strings

Rule ocaml-light-is-structurally-equal(S_1 : implemented-strings, S_2 : implemented-strings) ⇝
is-equal(S_1, S_2)

Tuples

Rule ocaml-light-is-structurally-equal(tuple( ), tuple( )) $\rightsquigarrow$ true

Rule ocaml-light-is-structurally-equal(tuple( ), tuple(V^+)) $\rightsquigarrow$ false

Rule ocaml-light-is-structurally-equal(tuple(V^+), tuple( )) $\rightsquigarrow$ false

Rule ocaml-light-is-structurally-equal(tuple(V, V^*), tuple(W, W^*)) $\rightsquigarrow$
  and(
    ocaml-light-is-structurally-equal(V, W),
    ocaml-light-is-structurally-equal(tuple(V^*), tuple(W^*)))

Lists

Rule ocaml-light-is-structurally-equal([ ], [ ]) $\rightsquigarrow$ true

Rule ocaml-light-is-structurally-equal([ ], [V^+]) $\rightsquigarrow$ false

Rule ocaml-light-is-structurally-equal([V^+], [ ]) $\rightsquigarrow$ false

Rule ocaml-light-is-structurally-equal([V, V^*], [W, W^*]) $\rightsquigarrow$
  and(
    ocaml-light-is-structurally-equal(V, W),
    ocaml-light-is-structurally-equal([V^*], [W^*]))

Records

Rule
$$\frac{\text{dom}(Map_1) == \text{dom}(Map_2)}{\text{ocaml-light-is-structurally-equal}(\text{record}(Map_1 : \text{maps}(\_, \_)), \text{record}(Map_2 : \text{maps}(\_, \_))) \rightsquigarrow}$$
  not(
    is-in-set(
      false,
      set(
        interleave-map(
          ocaml-light-is-structurally-equal(
            checked lookup(Map_1, given),
            checked lookup(Map_2, given)),
          set-elements(dom(Map_1))))))

References

Rule ocaml-light-is-structurally-equal(V_1 : variables, V_2 : variables) $\rightsquigarrow$
  ocaml-light-is-structurally-equal(assigned(V_1), assigned(V_2))

Vectors

Rule ocaml-light-is-structurally-equal(Vec_1 : vectors(values), Vec_2 : vectors(values)) $\rightsquigarrow$
  ocaml-light-is-structurally-equal([vector-elements(Vec_1)], [vector-elements(Vec_2)])

Variants

```
Rule  ocaml-light-is-structurally-equal(variant($Con_1$, $V_1$), variant($Con_2$, $V_2$)) ⇝
        if-true-else(
          is-equal($Con_1$, $Con_2$),
          if-true-else(
            or(is-equal(tuple( ), $V_1$), is-equal(tuple( ), $V_2$)),
            and(is-equal(tuple( ), $V_1$), is-equal(tuple( ), $V_2$)),
            ocaml-light-is-structurally-equal($V_1$, $V_2$)),
          false)
```

Functions

```
Rule  ocaml-light-is-structurally-equal(_ : functions(_, _), _ : functions(_, _)) ⇝
        throw(variant("Invalid_argument", "equal: functional value"))
```

Console display

```
Funcon  ocaml-light-to-string(_ : values) : ⇒ strings
```

ocaml-light-to-string(V) gives the string represention of OCaml Light values as implemented by the ocaml interpreter.

```
Rule  ocaml-light-to-string(null-value) ⇝ "()"
Rule  ocaml-light-to-string(B : booleans) ⇝ to-string(B)
Rule  ocaml-light-to-string(I : integers) ⇝ to-string(I)
Rule  ocaml-light-to-string(F : implemented-floats) ⇝ to-string(F)
Rule  ocaml-light-to-string(C : implemented-characters) ⇝
        string-append("'", to-string(C), "'")

                                      S ≠ [ ]
Rule  ---------------------------------------------------------------------------------
      ocaml-light-to-string(S : implemented-strings) ⇝ string-append(""", S, """)

Rule  ocaml-light-to-string(_ : functions(_, _)) ⇝ "<fun>"
Rule  ocaml-light-to-string(V : variables) ⇝
        string-append("ref ", ocaml-light-to-string(assigned(V)))
Rule  ocaml-light-to-string(variant(Con, Arg)) ⇝
        if-true-else(
          is-equal(tuple( ), Arg),
          Con,
          string-append(Con, " ", ocaml-light-to-string(Arg)))
Rule  ocaml-light-to-string(tuple(V : values, V+ : values+)) ⇝
        string-append(
          "(",
          intersperse(", ", interleave-map(ocaml-light-to-string(given), V, V+)),
          ")")
Rule  ocaml-light-to-string([V* : values*]) ⇝
        string-append(
          "[",
          intersperse("; ", interleave-map(ocaml-light-to-string(given), V*)),
          "]")
Rule  ocaml-light-to-string(V : implemented-vectors) ⇝
        string-append(
          "[|",
          intersperse(
            "; ",
            interleave-map(
              ocaml-light-to-string(assigned(given)),
              vector-elements(V))),
          "|]")
Rule  ocaml-light-to-string(record(M : maps(_, _))) ⇝
        string-append(
          "{",
          intersperse(
            "; ",
            interleave-map(
              string-append(arg-1, " = ", ocaml-light-to-string(arg-2)),
              map-elements(M))),
          "}")
```

```
Funcon   ocaml-light-define-and-display(Env : envs) : ⇒ envs
      ⇝ sequential(
              effect left-to-right-map(
                 print(arg-1, “ = ”, ocaml-light-to-string arg-2, “\n”),
                 map-elements Env),
              Env)

Funcon   ocaml-light-evaluate-and-display(V : implemented-values) : ⇒ envs
      ⇝ sequential(
              print(“- = ”, ocaml-light-to-string V, “\n”),
              map( ))
```