Component-based semantics

Peter D Mosses, Swansea University

IFIP WG 2.2 Meeting, 24-26 September 2012, CWI, Amsterdam

PLanCompS: Programming Language Components and Specifications

Peter D Mosses

Swansea University

IFIP WG 2.2 Meeting, Paris
22 September 2011

Main idea

A component-based framework

- to support design, specification, implementation of programming and domain-specific languages

with some novel aspects:

- an open-ended collection of highly reusable language components: **funcons** (fundamental constructs)
- each funcon is to be specified **independently**
- a **digital library** of language and funcon specifications

Background

Programming languages

- **many hundreds of them**
  - **older:** Fortran, Cobol, Ada, C++, Scheme, Haskell, …
  - **newer:** Java, C#, Python, Ruby, OCaml, VBScript, …
  - **domain-specific:** PHP, Javascript, …
- **continually evolving**
  - **new versions** of existing languages
  - **new languages**

Background

Programming language specifications

- **program syntax – formal**
  - lexical symbols
  - phrase structure
- **program semantics – informal**
  - static (compile-time) behaviour
  - dynamic (run-time) behaviour

Background

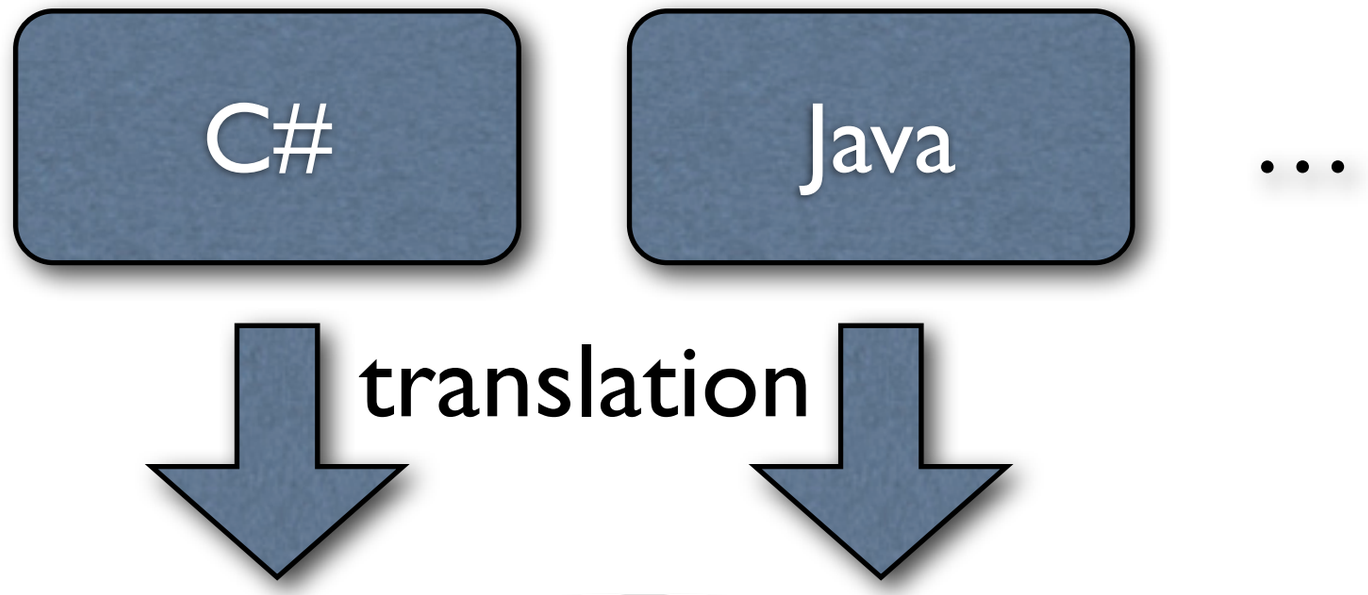
Current pragmatic problems with formal semantics concerning:

- **reuse**
- **co-evolution**
- **tool support**

when specifying ***major*** or ***rapidly-evolving*** languages

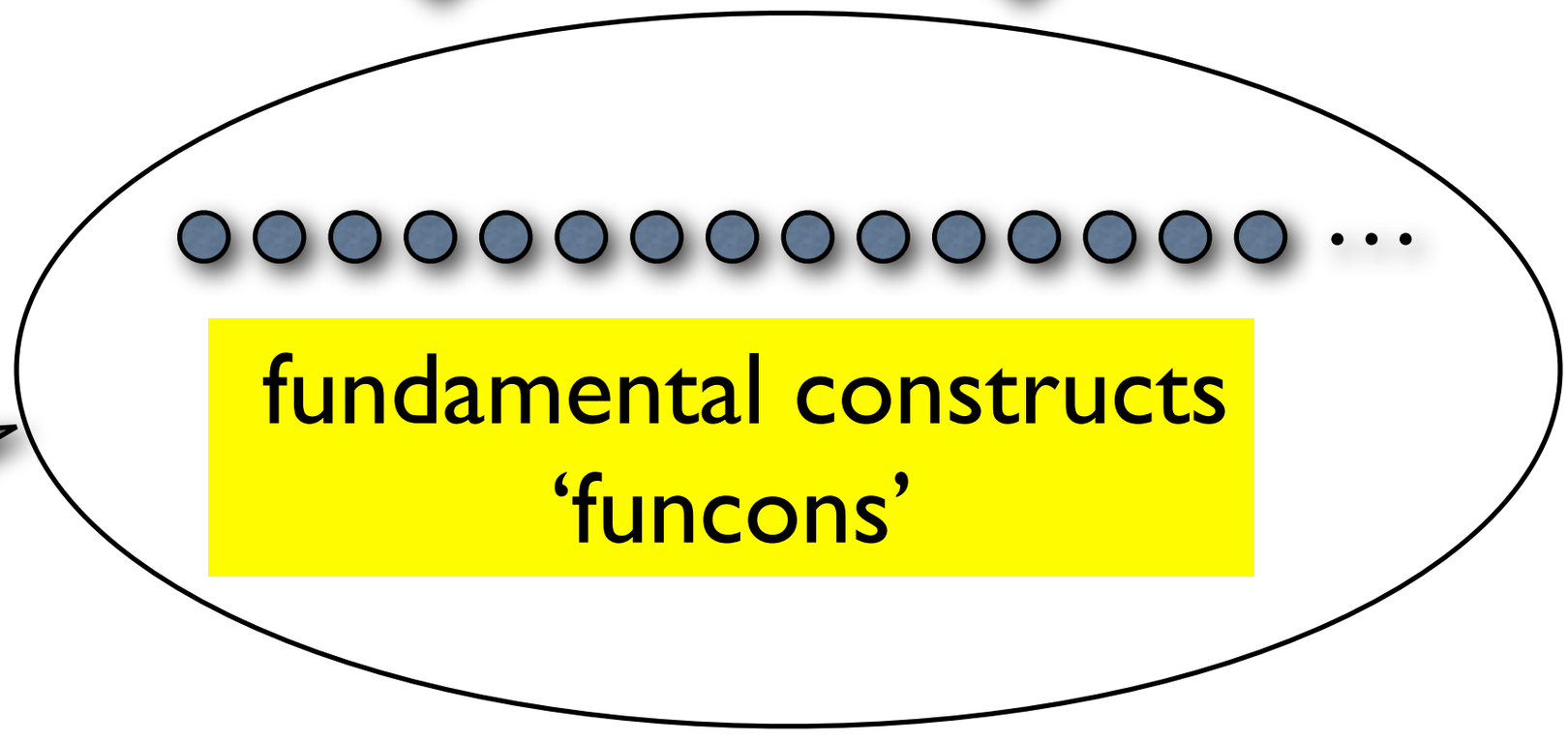
PLANCOMPS

Programming Languages (incl. DSLs)



Components and their Specifications

reusable



Component-based specifications

Syntax (concrete, abstract)

- BNF + regular expressions

Semantics (static, dynamic)

- context-free translation to funcons

Funcons

- modular SOS rules
- modular bisimilarity theory

OCaml Light

OCaml (Objective Caml, F#)

- a popular OO functional language

Caml Light

- a pedagogical sublanguage of Caml

OCaml Light

- defined in SOS using Ott by Scott Owens

OCaml Light

syntax

OCaml Light syntax

```
        Outlines
========

Lexical non-terminals
---------------------
...

Names
-----
I : ident           ::= lowercase_ident
                    |   capitalized_ident

VN : value_name     ::= lowercase_ident
                    |   '(' operator_name ')'

ON : operator_name  ::= prefix_symbol
                    |   infix_op

infix_op            ::= infix_symbol
                |   '*'    | '='   | 'or'   | '&'
                |   ':='   | 'mod' | 'land' | 'lor'
                |   'lxor' | 'asr' | 'lsl'  | 'lsr'

IO : infix_op \ ( '&&' | '&' | '||' | 'or' )

K : constr      ::= capitalized_ident

TK : typeconstr ::= lowercase_ident

F : field       ::= lowercase_ident

Type expressions
----------------

T : typexpr ::= ...

Constants
---------

C : constant    ::= integer_literal
            |   float_literal
            |   char_literal
            |   string_literal
            |   constr
            |   'false'
            |   'true'
            |   '[' ']'
            |   '(' ')'
```

```
Expressions
-----------

E : expr        ::= value_name
            |   constant
            |   '(' expr ')'
            |   'begin' expr 'end'
            |   '(' expr ':' typexpr ')'
            |   expr ( ',' expr )+
            |   constr expr
            |   expr '::' expr
            |   '[' expr ( ';' expr )* ']'
            |   '{' field '=' expr
                    ( ';' field '=' expr )* '}'
            |   '{' expr 'with'
                    field '=' expr
                    ( ';' field '=' expr )* '}'
            |   expr expr+
            |   prefix_symbol expr
            |   ( '+' | '-' | '+.' | '-.' ) expr
            |   expr infix_op expr
            |   expr '.' field
            |   'if' expr 'then' expr ( 'else' expr )?
            |   'while' expr 'do' expr 'done'
            |   'for' value_name '=' expr
                    ( 'to' | 'downto' ) expr
                    'do' expr 'done'
            |   expr ';' expr
            |   'match' expr 'with' pattern_matching
            |   'function' pattern_matching
            |   'fun' pattern+ '->' expr
            |   'try' expr 'with' pattern_matching
            |   'let' 'rec'? let_binding
                    ( 'and'  let_binding )*
                    'in' expr
            |   'assert' expr

Patterns
--------

P : pattern     ::= value_name
            |   '_'
            |   constant
            |   pattern 'as' value_name
            |   '(' pattern ')'
            |   '(' pattern ':' typexpr ')'
            |   pattern '|' pattern
            |   constr pattern
            |   pattern ( ',' pattern )+
            |   '{' field '=' pattern
                    ( ';' field '=' pattern )* '}'
            |   '[' pattern ( ';' pattern )* ']'
            |   pattern '::' pattern
```

```
Pattern matching
----------------

PM : pattern_matching   ::=
            '|'? pattern '->' expr
                ( '|' pattern '->' expr )*

Declarations and definitions
----------------------------

LB : let_binding    ::=
                pattern '=' expr
            |   value_name pattern+ '=' expr
            |   value_name ':' typexpr '=' expr
            |   value_name pattern+ ':' typexpr '=' expr

TD : type_definition    ::= ...

ED : exception_definition ::= ...

D : definition  ::=
                'let' 'rec'? let_binding
                    ( 'and' let_binding )*
            |   type_definition
            |   exception_definition

TP : toplevel_phrase    ::= definition | expr

TI : toplevel_input ::= toplevel_phrase* ';;'

Programs
--------
PR: program         ::= toplevel_input+
```

OCaml Light syntax

```
   Lexical non-terminals
---------------------
lowercase_ident
capitalized_ident
prefix_symbol
infix_symbol
integer_literal
float_literal
char_literal
string_literal

Names
-----
ident
operator_name
infix_op
constr
typeconstr
field

Type expressions
----------------
typexpr

Constants
---------
constant
```

```
Expressions
-----------
expr

Patterns
--------
pattern

Pattern matching
----------------
pattern_matching

Declarations and definitions
----------------------------
let_binding
type_definition
typedef
type_params
type_param
constr_decl
field_decl
exception_definition
definition
toplevel_phrase
toplevel_input
program
```

OCaml Light syntax

```
   Lexical non-terminals
---------------------

...
Names
-----
I : ident           ::= lowercase_ident | capitalized_ident
VN : value_name     ::= lowercase_ident | '(' operator_name ')'
ON : operator_name  ::= prefix_symbol   | infix_op

infix_op            ::= infix_symbol
                |   '*'    | '='   | 'or'   | '&'
                |   ':='   | 'mod' | 'land' | 'lor'
                |   'lxor' | 'asr' | 'lsl'  | 'lsr'

IO : infix_op \ ( '&&' | '&' | '||' | 'or' )

K : constr      ::= capitalized_ident
F : field       ::= lowercase_ident
```

OCaml Light syntax

```
   Type expressions
----------------
T : typexpr ::= ...

Constants
---------
C : constant::= integer_literal
            |   float_literal
            |   char_literal
            |   string_literal
            |   constr
            |   'false'
            |   'true'
            |   '[' ']'
            |   '(' ')'
```

OCaml Light syntax

```
   Expressions
-----------
E : expr    ::= value_name
            |   constant
            |   '(' expr ')'
            |   'begin' expr 'end'
            |   '(' expr ':' typexpr ')'
            |   expr ( ',' expr )+
            |   constr expr
            |   expr '::' expr
            |   '[' expr ( ';' expr )* ']'
            |   '{' field '=' expr ( ';' field '=' expr )* '}'
            |   '{' expr 'with'
                    field '=' expr ( ';' field '=' expr )* '}'
            |   expr expr+
            |   prefix_symbol expr
            |   ( '+' | '-' | '+.' | '-.' ) expr
            |   expr infix_op expr
            |   ...
```

OCaml Light syntax

```
   Expressions
-----------
E : expr    ::= ...
            |   expr '.' field
            |   'if' expr 'then' expr ( 'else' expr )?
            |   'while' expr 'do' expr 'done'
            |   'for' value_name '=' expr
                        ( 'to' | 'downto' ) expr
                    'do' expr 'done'
          |   expr ';' expr
          |   'match' expr 'with' pattern_matching
          |   'function' pattern_matching
          |   'fun' pattern+ '->' expr
          |   'try' expr 'with' pattern_matching
          |   'let' 'rec'? let_binding
                 ( 'and'  let_binding )*
                 'in' expr
          |   'assert' expr
```

OCaml Light syntax

```
   Patterns
________

P : pattern ::= value_name
             |   '_'
             |   constant
             |   pattern 'as' value_name
             |   '(' pattern ')'
             |   '(' pattern ':' typexpr ')'
             |   pattern '|' pattern
             |   constr pattern
             |   pattern ( ',' pattern )+
             |   '{' field '=' pattern ( ';' field '=' pattern )* '}'
             |   '[' pattern ( ';' pattern )* ']'
             |   pattern '::' pattern

Pattern matching
________________

PM : pattern_matching   ::=
           '|'? pattern '->' expr ( '|' pattern '->' expr )*
```

OCaml Light syntax

```
   Declarations and definitions
---------------------------
LB : let_binding ::= pattern '=' expr
                 |   value_name pattern+ '=' expr
                 |   value_name ':' typexpr '=' expr
                 |   value_name pattern+ ':' typexpr '=' expr

TD : type_definition      ::= ...
ED : exception_definition ::= ...

D : definition   ::= 'let' 'rec'? let_binding ( 'and' let_binding )*
                 |   type_definition
                 |   exception_definition
TP : toplevel_phrase ::= definition | expr
TI : toplevel_input  ::= toplevel_phrase* ';;'

Programs
--------
PR: program ::= toplevel_input+
```

OCaml Light Semantics

Translation to funcons

```
id[[ VN ]]    : id          VN : value_name
value[[ C ]]  : value       C  : constant
expr[[ E ]]   : expr        E  : expr
patt[[ P ]]   : patt        P  : pattern
abs[[ PM ]]   : abs         PM : pattern_matching
```

```
decl[[ LB ]] : decl          LB : let_binding
decl[[ D ]]  : decl          D  : definition
decl[[ TP ]] : decl          TP : toplevel_phrase
decl[[ TI ]] : decl          TI : toplevel_input
decl[[ PR ]] : decl          PR : program
```

Funcons

Computation types

```
Type computes(X)

Type comm = computes(skip)

Type decl = computes(env)

Type expr = computes(value)
```

```
Type depends(X,Y)

Type abs = depends(passable,value)

Type patt = depends(value,env)
```

Value types

```
atom      < value
boolean   < value
int       < value
float     < value
char      < value
string    < value
tuple     < value
record    < value
variant   < value
function  < value
variable  < value
skip      < value
```

```
passable  = value
throwable = value
bindable  = value
storable  = value
```

Preliminary tool support

IDE :

- **Spoofax**

Parsing, translation to funcons :

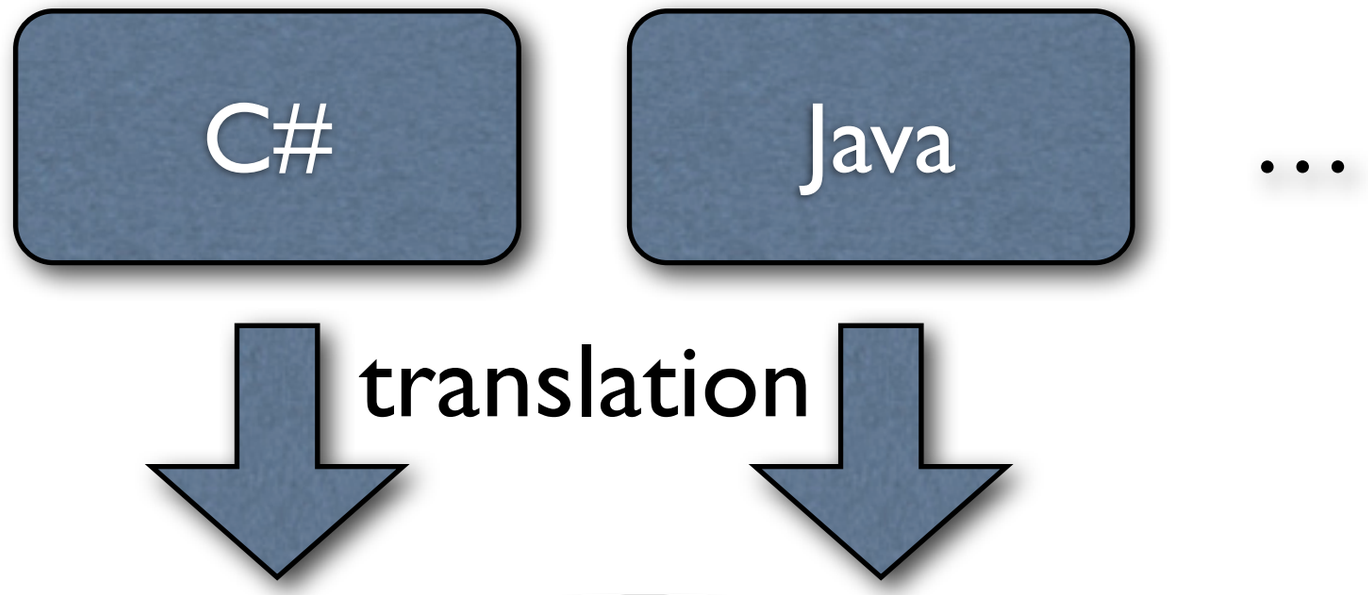
- **ASF+SDF**

Funcon interpretation :

- **Prolog**

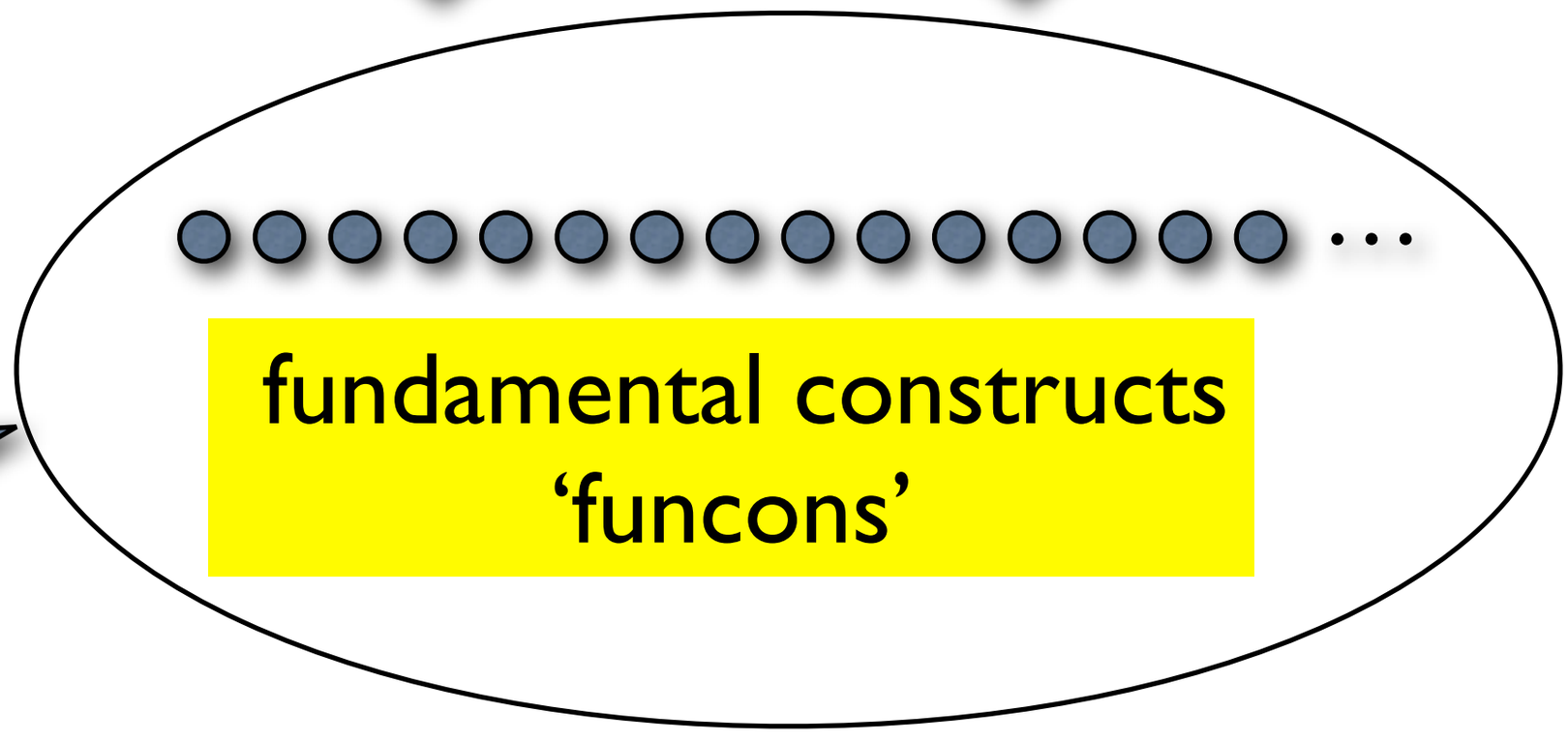
PLANCOMPS

Programming Languages (incl. DSLs)



Components and their Specifications

reusable



OCaml Light

Semantics

Translation to funcons

```
id[[ VN ]]   : id
value[[ C ]] : value
expr[[ E ]]  : expr
patt[[ P ]]  : patt
abs[[ PM ]]  : abs
```

```
VN : value_name
C  : constant
E  : expr
P  : pattern
PM : pattern_matching
```

```
decl[[ LB ]] : decl
decl[[ D ]]  : decl
decl[[ TP ]] : decl
decl[[ TI ]] : decl
decl[[ PR ]] : decl
```

```
LB : let_binding
D  : definition
TP : toplevel_phrase
TI : toplevel_input
PR : program
```

Value and operator names

```
SYNTAX      VN : value_name

SEMANTICS   id[[ VN ]] : id

* value_name ::= lowercase_ident

id[[ LI ]] = id(LI)

* value_name ::= '(' operator_name ')'

id[[ ( ON ) ]] = id(ON)

SYNTAX      ON : operator_name

* operator_name ::= prefix_symbol

* operator_name ::= infix_op
```

Literal constants

```
SYNTAX      C : constant

SEMANTICS   value[[ C ]] : value

* constant ::= integer_literal

value[[ IL ]] = IL

int < value

* constant ::= float_literal

value[[ FL ]] = FL

float < value

...
char < value

...
string < value
```

Constructors and booleans

```
SYNTAX      C : constant

SEMANTICS   value[[ C ]] : value


* constant ::= constr

value[[ K ]] = K

atom < value

* constant ::= 'false' | 'true'

value[[ false ]] = false

value[[ true ]] = true

boolean < value
```

Empty list and tuple

```
SYNTAX      C : constant

SEMANTICS   value[[ C ]] : value


* constant ::= '[' ']'

value[[ [ ] ]] = '[]'


atom < value


* constant ::= '(' ')'

value[[ ( ) ]] = tuple_empty

tuple < value
```

Expressions

Simple expressions

```
SYNTAX      E : expr

SEMANTICS   expr[[ E ]] : expr

* expr ::= value_name

expr[[ VN ]] = bound_value(id[[ VN ]])

* expr ::= constant

expr[[ C ]] = value[[ C ]]
```

Grouped expressions

```
* expr ::= '(' expr ')'

expr[[ ( E ) ]] = expr[[ E ]]


* expr ::= 'begin' expr 'end'

expr[[ begin E end ]] = expr[[ E ]]
```

```
* expr ::= '(' expr ':' typexpr ')'

expr[[ ( E : T ) ]] = expr[[ E ]]
// for dynamic semantics only
```

Tuple expressions

```
* expr ::= expr ( ',' expr )+

expr[[ E1 , E2 ... ]] =
            tuple_prefix(expr[[ E1 ]],
                         expr_tuple[[ E2 ... ]])

tuple < value

... : (',' expr)*
```

```
SEMANTICS   expr_tuple[[ E ... ]] : expr

- expr_tuple[[ E1 ]] =
            tuple_prefix(expr[[ E1 ]], tuple_empty)

- expr_tuple[[ E1 , E2 ... ]] =
            tuple_prefix(expr[[ E1 ]],
                         expr_tuple[[ E2 ... ]])
```

Constructors and lists

```
* expr ::= constr expr

expr[[ K E ]] = variant(K, expr[[ E ]])

variant < value

* expr ::= expr '::' expr

expr[[ E1 :: E2 ]] =
                    variant('::', tuple2(expr[[ E1 ]],
                                         expr[[ E2 ]]))
variant < value


* expr ::= '[' expr ( ';' expr )* ']'

expr[[ [ E1 ] ]] = expr[[ E1 :: [ ] ]]

expr[[ [ E1 ; E2 ... ] ]] = expr[[ E1 :: [ E2 ... ] ]]

... : (';' expr)*
```

Record expressions

```
* expr ::= '{' field '=' expr
                ( ';' field '=' expr )* '}'

expr[[ { F1 = E1 } ]] = record1(F1, expr[[ E1 ]])

expr[[ { F1 = E1 ; F2 = E2 ... } ]] =
    record_union(expr[[ { F1 = E1 } ]],
                 expr[[ { F2 = E2 ... } ]])

record < value

... : (';' field '=' expr)*


* expr ::= expr '.' field

expr[[ E . F ]] = record_select(expr[[ E1 ]], F)

record < value
```

Record expressions

```
* expr ::= '{' expr 'with' field '=' expr
                              ( ';' field '=' expr )* '}'
```

```
expr[[ { E with F1 = E1 } ]] =
    record_over(expr[[ { F1 = E1 } ]], expr[[ E ]])

expr[[ { E with F1 = E1 ; F2 = E2 ... } ]] =
    expr[[ { { E with F1 = E1 } with F2 = E2 ... } ]]
```

```
record < value

... : (';' field '=' expr)*
```

Function application

```
* expr ::= expr expr+

expr[[ E1 E2 ]] = apply_function(expr[[ E1 ]],
                                 expr[[ E2 ]])

expr[[ E1 E2 E3 ... ]] = expr[[ (E1 E2) E3 ... ]]

function < value

... : expr*
```

Prefix operators

```
* expr ::= prefix_op expr

expr[[ P0 E1 ]] = expr[[ ( P0 ) E1 ]]


* expr ::= ( '+' | '-' | '+.' | '-.' ) expr

expr[[ + E1 ]] = int_plus(0, expr[[ E1 ]])

expr[[ - E1 ]] = int_minus(0, expr[[ E1 ]])

int < value

expr[[ +. E1 ]] = float_plus(0.0, expr[[ E1 ]])

expr[[ -. E1 ]] = float_minus(0.0, expr[[ E1 ]])

float < value
```

Infix operators

```
* expr ::= expr infix_op expr

expr[[ E1 I0 E2 ]] = expr[[ ( I0 ) E1 E2 ]]

* IO : infix_op \ ( '&&' | '&' | '||' | 'or' )


expr[[ E1 && E2 ]] = if_true(expr[[ E1 ]],
                             expr[[ E2 ]],
                             false)

expr[[ E1 & E2 ]] = expr[[ E1 && E2 ]]

expr[[ E1 || E2 ]] = if_true(expr[[ E1 ]],
                             true,
                             expr[[ E2 ]])

expr[[ E1 or E2 ]] = expr[[ E1 || E2 ]]

boolean < value
```

Conditionals and loops

```
* expr ::= 'if' expr 'then' expr ( 'else' expr )?

expr[[ if E1 then E2 ]] =
    expr[[ if E1 then E2 else ( ) ]]

expr[[ if E1 then E2 else E3 ]] =
    if_true(expr[[ E1 ]], expr[[ E2 ]], expr[[ E3 ]])

boolean < value
```

```
* expr ::= 'while' expr 'do' expr 'done'

expr[[ while E1 do E2 done ]] =
    seq(while_true(expr[[ E1 ]], effect(expr[[ E2 ]])),
        tuple_empty)

boolean < value
```

For-loops

```
* expr ::= 'for' value_name '=' expr
                ( 'to' | 'downto' ) expr
                'do' expr 'done'
```

```
expr[[ for VN = E1 to E2 do E3 done ]] =
    apply_to_each(abs(bind(id[[ VN ]]),
                      effect(expr[[ E3 ]])),
                  int_closed_interval(expr[[ E1 ]],
                                      expr[[ E2 ]]))
```

```
expr[[ for VN = E1 downto E2 do E3 done ]] =
    apply_to_each(abs(bind(id[[ VN ]]),
                      effect(expr[[ E3 ]])),
     list_reverse(int_closed_interval(expr[[ E2 ]],
                                      expr[[ E1 ]])))
```

```
int < passable
```

Sequential expressions

```
* expr ::= expr ';' expr

expr[[ E1 ; E2 ]] = seq(effect(expr[[ E1 ]]),
                        expr[[ E2 ]])

done < value
```

Match expressions

```
* expr ::= 'match' expr 'with' pattern_matching

expr[[ match E with PM ]] =
    apply(prefer_over(abs[[ PM ]],
                      abs(any, throw('Match_failure'))),
          expr[[ E ]])

'Match_failure' : throwable
```

Function abstraction

```
* expr ::= 'function' pattern_matching

expr[[ function PM ]] =
    close function(prefer_over(abs[[ PM ]],
                               abs(any, throw('Match_failure'))))

'Match_failure' : throwable
```

```
* expr ::= 'fun' pattern+ '->' expr

expr[[ fun P1 -> E ]] =
    expr[[ function P1 -> E ]]

expr[[ fun P1 P2 ... -> E ]] =
    expr[[ fun P1 -> fun P2 ... -> E ) ]]

... : pattern*
```

Exception handling

```
* expr ::= 'try' expr 'with' pattern_matching

expr[[ try E with PM ]] =
    catch(expr[[ E ]],
          prefer_over(abs[[ PM ]], abs(throw(given))))
```

Let-expressions

```
* expr ::= 'let' 'rec'? let_binding
                        ( 'and' let_binding )* 'in' expr
```

```
expr[[ let LB ... in E ]] =
    scope(decl[[ LB ... ]], expr[[ E ]])

expr[[ let rec LB ... in E ]] =
    scope(recursive(decl[[ LB ... ]]), expr[[ E ]])
```

```
... : ('and' let_binding)*
```

Assertions

```
* expr ::= 'assert' expr

expr[[ assert E ]] =
    if_true(expr[[ E ]],
            tuple_empty,
            throw('Assert_failure'))

boolean < value
```

Patterns

Simple patterns

```
SYNTAX      P : pattern

SEMANTICS   patt[[ P ]] : patt

* pattern ::= value_name

patt[[ VN ]] = bind(id[[ VN ]])

* pattern ::= '_'

patt[[ _ ]] = any

* pattern ::= constant

patt[[ C ]] = only(value[[ C ]])
```

Composite patterns

* pattern ::= pattern 'as' value_name

```
patt[[ P as VN ]] = map_union(patt[[ P ]],
                              patt[[ VN ]])
```

* pattern ::= pattern '|' pattern

```
patt[[ P1 | P2 ]] = prefer_over(patt[[ P1 ]],
                                patt[[ P2 ]])
```

Constructor patterns

```
* pattern ::= constr pattern

patt[[ K P ]] =
    invert variant(only(K), patt[[ P ]])

variant < value


* pattern ::= pattern '::' pattern

    patt[[ P1 :: P2 ]] =
        invert variant(only('::'),
                       invert tuple2(patt[[ P1 ]],
                                     patt[[ P2 ]]))
```

List patterns

```
* pattern ::= '[' pattern ( ';' pattern )* ']'

patt[[ [ P1 ] ]] = patt[[ P1 :: [ ] ]]

patt[[ [ P1 ; P2 ... ] ]] = patt[[ P1 :: [ P2 ... ] ]]

... : ( ';' pattern )*
```

Tuple patterns

```
* pattern ::= pattern ( ',' pattern )+

patt[[ P1 , P2 ... ]] =
    invert tuple_prefix(patt[[ P1 ]],
                        patt_tuple[[ P2 ... ]])

... : ( ',' pattern )*
```

```
SEMANTICS   patt_tuple[[ P ... ]] : patt

- patt_tuple[[ P1 ]] =
    invert tuple_prefix(patt[[ P1 ]],
                        only(tuple_empty))

- patt_tuple[[ P1 , P2 ... ]] =
    invert tuple_prefix(patt[[ P1 ]],
                        patt_tuple[[ P2 ... ]])

tuple < value
```

Loose record patterns

```
* pattern ::= '{' field '=' pattern
                ( ';' field '=' pattern )* '}'

patt[[ { F1 = P1 ... } ]] =
    loose_record(patt_map[[ F1 = P1 ... ]])

... : ( ';' field '=' pattern )*
```

```
SEMANTICS   patt_map[[ F = P ... ]]  : map

- patt_map[[ F1 = P1 ]] = map1(F1, patt[[ P1 ]])

- patt_map[[ F1 = P1 ; F2 = P2 ... ]] =
        map_union(patt_map[[ F1 = P1 ]],
                  patt_map[[ F2 = P2 ... ]])
```

Pattern abstractions

```
SYNTAX      PM : pattern_matching

SEMANTICS   abs[[ PM ]] : abs


* pattern_matching ::= '|'? pattern '->' expr
                                ( '|' pattern '->' expr )*

abs[[ P1 -> E1 ]] = abs(patt[[ P1 ]], expr[[ E1 ]])

abs[[ P1 -> E1 | P2 -> E2 ... ]] =
    prefer_over(abs[[ P1 -> E1 ]],
                abs[[ P2 -> E2 ... ]])

abs[[ | P1 -> E1 ... ]] = abs[[ P1 -> E1 ... ]]

... : ( '|' pattern '->' expr )*
```

Declarations

Pattern matching declarations

```
SYNTAX      LB : let_binding

SEMANTICS   decl[[ LB ]] : decl

* let_binding ::= pattern '=' expr

decl[[ P = E ]] = match(expr[[ E ]], patt[[ P ]])

* let_binding ::= value_name pattern+ '=' expr

decl[[ VN P ... = E ]] = decl[[ VN = fun P ... -> E ]]

... : pattern*
```

Let-definitions

```
SYNTAX      D : definition

SEMANTICS   decl[[ D ]] : decl


* definition ::= 'let' 'rec'? let_binding
                                ( 'and' let_binding )*

decl[[ let LB ]] = decl[[ LB ]]
```

```
decl[[ let LB1 and LB2 ... ]] =
    map_union(decl[[ LB1 ]], decl[[ LB2 ... ]])


decl[[ let rec LB ... ]] = recursive(decl[[ LB ... ]])
```

Top-level phrases

```
SYNTAX      TP : toplevel_phrase

SEMANTICS   decl[[ TP ]] : decl


* toplevel_phrase ::= definition

decl[[ TP ]] = decl[[ D ]]
  when TP = D
```

```
* toplevel_phrase ::= expr

decl[[ TP ]] = seq(print(expr[[ E ]]), map_empty)
  when TP = E
```

Top-level inputs

```
SYNTAX      TI : toplevel_input

SEMANTICS   decl[[ TI ]] : decl

* toplevel_input ::= toplevel_phrase* ';;'

decl[[ ;; ]] = map_empty

decl[[ TP ... ;; ]] =
    accum(decl[[ TP ]], decl[[ ... ;; ]])

... : toplevel_phrase*
```

Programs

```
SYNTAX      PR : program

SEMANTICS   decl[[ PR ]] : decl


* program ::= toplevel_input+

decl[[ TI ... ]] = scope(ocaml_light_library,
                         decl_seq[[ TI ... ]])
```

```
... : toplevel_input*

SEMANTICS   decl_seq[[ TI ... ]] : decl

- decl_seq[[ TI ]] = decl[[ TI ]]

- decl_seq[[ TI1 TI2 ... ]] =

          accum(decl[[ TI1 ]], decl_seq[[ TI2 ... ]])
```

Funcons

Computation types

```
Type computes(X)

Type comm = computes(skip)

Type decl = computes(env)

Type expr = computes(value)
```

```
Type depends(X,Y)

Type abs = depends(passable,value)

Type patt = depends(value,env)
```

Value types

```
   atom       < value
boolean    < value
int        < value
float      < value
char       < value
string     < value
tuple      < value
record     < value
variant    < value
function   < value
variable   < value
skip       < value
```

```
passable  = value
throwable = value
bindable  = value
storable  = value
```

```
Funcon abs(patt,expr)                    : abs
Funcon accum(env,decl)                   : decl
Funcon any : patt
Funcon apply(depends(X,Y),X)             : computes(Y) { val(X), val(Y) }
Funcon apply_function(function,value)    : expr
Funcon apply_to_each(proc,list(value)): comm
Funcon bind(id)                          : patt
Funcon bound_value(id)                   : computes(bindable)
Funcon catch(expr,abs)                   : expr
Funcon close F(abs)                      : abs { F(abs) : abs }
Funcon effect(X)                         : comm { comp(X) }
Funcon given                             : depends(X,X)
Funcon if_true(boolean,X,X)              : X
Funcon invert F(patt,patt)               : patt { F(X,Y) : Z}
Funcon invert F(patt)                    : patt { F(X) : Y}
Funcon invert F                          : patt { F : X}
Funcon loose_record(map(field,patt))  : patt
Funcon match(value,patt)                 : decl
Funcon only(value)                       : patt
Funcon prefer_over(depends(X,Y),depends(X,Y)) : depends(X,Y)
Funcon print(value)                      : comm
Funcon recursive(decl)                   : decl
Funcon scope(env,X)                      : X { comp(X) }
Funcon seq(comm,X)                       : X { comp(X) }
Funcon throw(throwable)                  : X { comp(X) }
Funcon while_true(computes(boolean),comm) : comm
```

Procedural abstraction

Funcon **abs**(**patt,expr**) **: abs**

Glossary:

$abs(Patt,Expr)$ abstracts an expression over a given pattern. $Patt$ is evaluated in the current (dynamic) environment, and $Expr$ evaluated in the environment generated by $Patt$.

Rules:

$$\frac{Patt \longrightarrow Patt'}{\mathbf{abs}(Patt,Exp) \longrightarrow abs(Patt',Exp)}$$

$$\mathbf{abs}(Env,Exp) \longrightarrow \mathbf{scope}(Env,Exp)$$

Binding accumulation

```
Funcon accum(env,decl) : decl

Glossary:

Elaborates a $Decl$ with $Env$ overiding the current
environment, and then adds elaborations onto that $Env$.

Rules:

        env map_over(Env,Env1) |- Decl --> Decl'
-------------------------------------------------------
env Env1 |- accum(Env,Decl) --> accum(Env,Decl')

accum(Env,Env1) --> map_over(Env1,Env)
```

Anonymous pattern

```
Funcon any : patt

Glossary:

The pattern $any$ matches any value.

Rules:

    given Value |- any --> map_empty
```

Application

```
Funcon apply(depends(X,Y),X) : computes(Y)
                                 { val(X), val(Y) }

Glossary:

Applies an $Abs$ to a $Passable$ to obtain the result.

Rules:
```

$$\frac{\text{given } X \vdash \text{DependsXY} \longrightarrow \text{DependsXY}'}{\text{given Val} \vdash \textbf{apply}(\text{DependsXY},X) \longrightarrow \text{apply}(\text{DependsXY}',X)}$$

$$\textbf{apply}(\text{Val},X) \longrightarrow \text{Val}$$

Function application

```
Funcon apply_function(function,value) : expr

Glossary:

$apply_function(Function,Value)$ applies a $Function$ to
a $Value$ and returns the result.

Rules:

apply_function(function(Abs),Value) = apply(Abs,Value)
```

Execute for each item

```
Funcon apply_to_each(proc,list(value)) : comm

Glossary:

$apply_to_each(Proc,List)$ runs $Proc$ sequentially on
each element of $List$.

Rules:

apply_to_each(Proc,list_empty) = skip

apply_to_each(Proc,list_prefix(Value,List)) =
seq(apply(Proc,Value),apply_to_each(Proc,List))
```

Binding pattern

```
Funcon bind(id) : patt

Glossary:

$bind(Id)$ matches any value, and binds $Id$ to it.

Rules:

given Value |- bind(Id) -->
map_update(map_empty,Id,Value)
```

Bound value

```
Funcon bound_value(id) : computes(bindable)

Glossary:

The expression $bound_value(Id)$ returns the value bound
to $Id$.

Rules:
```

$$\frac{\texttt{map\_select(Env,Id) = Expr}}{\texttt{env Env |- bound\_value(Id) --> Expr}}$$

Catching thrown values

```
Funcon catch(expr,abs) : expr

Glossary: The computation $catch(Expr,Abs)$ evaluates
$Expr$ and returns the result. If $Expr$ throws an
exception, the computation returns $Abs$ applied to that
exception.

Requires: Throwable < Value

Rules:

          Expr -- exception none --> Expr'
-------------------------------------------------------
catch(Expr,Abs) -- exception none --> catch(Expr',Abs)

catch(Value,Abs) --> Value

       Expr -- exception some(Throwable) --> Expr'
-------------------------------------------------------
catch(Expr,Abs) -- exception none -->
apply(Abs,Throwable)
```

Abstraction closure

```
Funcon close F(abs) : abs { F(abs) : abs }

Glossary:

Constructs a statically bound F.

Rules:

     env Env |- close F(Abs) --> F(closure(Abs,Env))
```

Effect

```
Funcon effect(X) : comm { comp(X) }

Glossary:

The computation $effect(X)$ computes $X$ until it is a
value and disregards the result.

Rules:

          X --> X'
-------------------------
effect(X) --> effect(X')

effect(Val) --> skip
```

Given value

```
Funcon given : depends(X,X)

Glossary:

Returns the given value.

Rules:

   ~ X = null
-----------------------
given X |- given --> X
```

Conditional

```
Funcon if_true(boolean,X,X) : X

Glossary:

At run-time, the computation $if_true(Boolean,X1,X2)$
first evaluates $Boolean$. If the result is $true$, it
executes $X1$; if the result is $false$, it executes
$X2$.

Rules:

if_true(true,X1,X2) = X1

if_true(false,X1,X2) = X2
```

Deconstruction

```
   Funcon invert F(Patt,Patt) : Patt { F(X,Y) : Z}
Funcon invert F(Patt)      : Patt { F(X) : Y}
Funcon invert F            : Patt { F : X}
```

Glossary: The pattern invert F(Patt1,Patt2) matches F(X,Y) when X matches Patt1 and Y matches Patt2. We also include unary and nullary versions.

Rules:

```
                given X |- Patt1 --> Patt1'
-------------------------------------------------------
given F(X,Y) |- invert F(Patt1,Patt2) -->
invert F(Patt1',Patt2)

                given Y |- Patt2 --> Patt2'
-------------------------------------------------------
given F(X,Y) |- invert F(Patt1,Patt2) -->
invert F(Patt1,Patt2')

given F(X,Y) |- invert F(Env1,Env2) --> map_over(Env1,Env2)

                         ~ Value = F(X,Y)
-------------------------------------------------------------
given Value |- invert F(Patt1,Patt2) -- not_in_domain true --> stuck

         given X |- Patt --> Patt'
-----------------------------------------
given F(X) |- invert F(Patt) --> invert F(Patt')

given F(X) |- invert F(Env) --> Env

                      ~ Value = F(X)
-----------------------------------------------------
given Value |- invert F(Patt) -- not_in_domain true --> stuck

given F |- invert F --> map_empty

                     ~ Value = F
-------------------------------------------------
given Value |- invert F -- not_in_domain true --> stuck
```

Loose record pattern

```
Funcon loose_record(map(field,patt)) : patt

Glossary:

$loose_record(Map,Pattern)$ matches any $Record$ such
that whenever $map_select(Map,Field) = Patt$, the $Field
$ projection of $Record$ matches $Patt$.

Rules:

given Record |- loose_record(map_empty) --> map_empty

given Record |- loose_record(map_prefix(Field,Patt,Map1)) -->
accum(match(Record,record_item(Field,Patt)),match(Record,loose_record(Map1)
))

                    ~ Value : Record
-------------------------------------------------------
given Value |- loose_record(Map) -- not_in_domain true --> stuck
```

Pattern matching declaration

```
Funcon match(value,patt) : Decl

Glossary:

$match(Value,Patt)$ tries to match $Value$ against $Patt
$, computing the resulting bindings.

Rules:

                    given Value |- Pat --> Pat'
-------------------------------------------------------------------
given Value1 |- match(Value,Pat) --> match(Value,Pat')

match(Value,Env) --> Env
```

Constant pattern

```
Funcon only(value) : patt

Glossary:

$only(Groundval)$ matches only the data value $Value$.

Rules:

                Value : groundval
-------------------------------------------
given Value |- only(Value) --> map_empty

Value1 : groundval, Value2 : groundval,
~ Value1 = Value2
-------------------------------------------------------------
given Value1 |- only(Value2) -- not_in_domain true -->
stuck
```

Abstraction preference

```
Funcon prefer_over(depends(X,Y),depends(X,Y)) : depends(X,Y)

Glossary:

$prefer_over(Dep1,Dep2)$ evaluates $Dep1$ on the $given$ argument. If it's
undefined, $Dep2$ is applied to the $given$ argument.

Rules:

                Dep -- not_in_domain false --> Dep'
-------------------------------------------------------------------
prefer_over(Dep,Dep1) -- not_in_domain false --> prefer_over(Dep',Dep1)

prefer_over(Val,Dep) --> Val

        Dep -- not_in_domain true --> Dep'
-------------------------------------------------------
prefer_over(Dep,Dep1) -- not_in_domain false --> Dep1
```

Print

```
Funcon print(value) : comm

Glossary:

The computation $print(Value)$ prints $Value$ to output.

Rules:

print(Value) -- output list_prefix(Value,list_empty) -->
skip
```

Recursive declaration

Funcon **recursive**(decl) : decl

```
Glossary:

$recursive(Env)$ evaluates computation entries in $Env$ with respect to itself, and returns the resulting environment of
values. 1. Compute forwards and get resulting map. 2. Duplicate map. 3. From one copy: construct environment of forwards.
4. Evaluate entries of $Env$ within that environment of forwards. 5. From second copy: for each id, connect that forward to
the current bound value. NB: Declaration is evaluated twice in current model (once to find out which identifiers are bound,
again to evaluate in the environment with those identifiers bound to forwards.)

Rules:

recursive(Decl) = reclose(generate_forwards(Decl),Decl)


Funcon generate_forwards(env) : computes(map(id,fwd))

Rules:

generate_forwards(map_empty) = map_empty
                                                    next_forward(Fo) = Fwd
-------------------------------------------------------------------------------------------------------------------------
(generate_forwards(map_prefix(Id,Expr,Env)), forwards Fo) --> (map_prefix(Id,Fwd,generate_forwards(Env)), forwards map_update(Fo,Fwd,undefined))

Funcon reclose(map(id,fwd),decl) : decl

Rules:

                             fwd_env(MapIdFwd) = Env1
-------------------------------------------------------------------------
reclose(MapIdFwd,Decl) --> accum(Env1,accum(Decl,seq(tie(MapIdFwd),Env1)))


Operation fwd_env(map(id,fwd)) : env

Rules:

fwd_env(map_empty) = map_empty

fwd_env(map_prefix(Id,Fwd,Map)) = map_prefix(Id,forward(Fwd),fwd_env(Map))


Funcon tie(map(id,fwd)) : cmd

Rules:

tie(map_empty) = skip

env Env |- (tie(map_prefix(Id,Fwd,Map)), forwards Fo) --> (tie(Map), forwards map_update(Fo,Fwd,map_select(Env,Id)))
```

Scope restriction

```
Funcon scope(env,X): X { comp(X) }

Glossary:

The computation $scope(Env,X)$ runs $X$ with respect to
the bindings in $Env$.

Rules:

    env map_over(Env,Env1) |- X --> X'
-------------------------------------
env Env1 |- scope(Env,X) --> scope(Env,X')

scope(Env,Val) --> Val
```

Sequential execution

```
Funcon seq(comm,X) : X { comp(X) }

Glossary:

The computation $seq(Comm,X)$ first evaluates $Comm$,
and then $X$ is evaluated to a result.

Rules:
```

$$\frac{Comm \longrightarrow Comm'}{\texttt{seq}(Comm,X) \longrightarrow \texttt{seq}(Comm',X)}$$

$$\texttt{seq}(\texttt{skip},X) \longrightarrow X$$

Throw a value

```
Funcon throw(throwable) : X { comp(X) }

Uses: stuck

Glossary:

The computation $throw(Throwable)$ throws the exception
$Throwable$.

Rules:

throw(Throwable) -- exception some(Throwable) --> stuck
```

Loop

```
Funcon while_true(computes(boolean),comm) : comm

Uses: if_true, seq, skip

Glossary:

The computation $while_true(Expr,Comm)$ repeatedly runs
$Comm$ as long as the condition $Expr$ holds.
```

Rules:

```
while_true(Expr,Comm) -->
if_true(Expr,seq(Comm, while_true(Expr,Comm)),skip)
```

Ad hoc abbreviation

Funcon **ocaml_light_library : env**

Rules:

```
ocaml_light_library --> { id('+') |--> function(curry(abs int_plus)),
                          id('-') |--> function(curry(abs int_minus)),
                          id('*') |--> function(curry(abs int_times)),
                          id('/') |--> function(curry(abs int_divide)),
                          id('mod') |--> function(curry(abs int_modulo)),
                          id('land') |--> function(curry(abs int_and)),
                          id('lor') |--> function(curry(abs int_or)),
                          id('lxor') |--> function(curry(abs int_xor)),
                          id('lsl') |--> function(curry(abs int_lshift)),
                          id('lsr') |--> function(curry(abs int_rshift)),
                          id('asr') |--> function(curry(abs int_arshift)),
                          id('+.') |--> function(curry(abs float_plus)),
                          id('-.') |--> function(curry(abs float_minus)),
                          id('*.') |--> function(curry(abs float_times)),
                          id('/.') |--> function(curry(abs float_divide)),
                          id('**') |--> function(curry(abs float_exp)),
                          id('@') |--> ol_append,
                          id('^') |--> function(curry(abs string_append)),
                          id('!') |--> function(abs deref),
                          id(':=') |--> function(curry(abs assign)),
                          id('=') |--> function(curry(abs equal)),
                          id('<>') |--> function(curry(compose(abs not,abs equal))),
                          id('==') |--> function(curry(abs equal)),
                          id('!=') |--> function(curry(compose(abs not,abs equal))),
                          id('<') |--> function(curry(abs less)),
                          id('>') |--> function(curry(abs greater)),
                          id('<=') |--> function(curry(abs less_equal)),
                          id('>=') |--> function(curry(abs greater_equal)),
                          id('not') |--> function(abs not),
                          id('sin') |--> function(abs float_sin),
                          id('pi') |--> float_pi,
                          id('float') |--> function(abs int_to_float) }
```

Preliminary tool support

IDE :

- **Spoofax**

Parsing, translation to funcons :

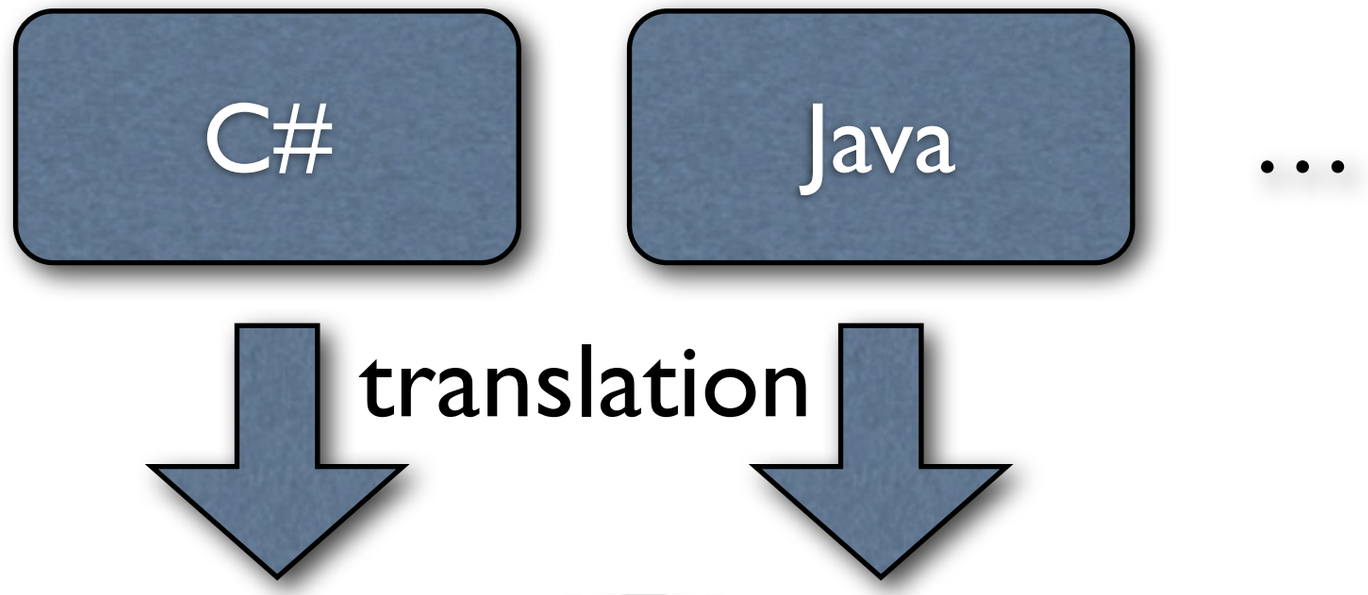
- **ASF+SDF**

Funcon interpretation :

- **Prolog**

PLANCOMPS

Programming Languages (incl. DSLs)



Components and their Specifications

reusable