Programming Languages, semantics, compilers

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

M1 Informatique - MPRI

2025–26

Semantics and interpretation of FUN

Organisation

- Course created by Thibaut Balabonski (I am using his material)
- Ecampus site :
  `https://ecampus.paris-saclay.fr/enrol/index.php?id=185060`
  (shared with the compiler's course of L3, guest access `COMPIL-25`)
- Evaluation : project + oral examination
- Exceptionnaly : next week course will be on monday 15th afternoon (13 :30 room C207)

Contents

This course explores programming languages, focusing on two main topics :

- their semantics, that is the formal description of the meaning of programs ;
- their compilation, that is the decomposition of high-level source language programs into simpler instructions whose execution can be performed by a computer.

We will define a functional programming language with a rich type system, and build an optimizing compiler and an execution environment for this language.

Plan

Semantics of a functional language

1. Semantics and interpretation
2. Types and safety

Assembly

1. The target : assembly code
2. Optimisations

Implementation of a functional language

1. Compilation
2. Automatic memory management

Semantics and interpretation of FUN

The FUN Language

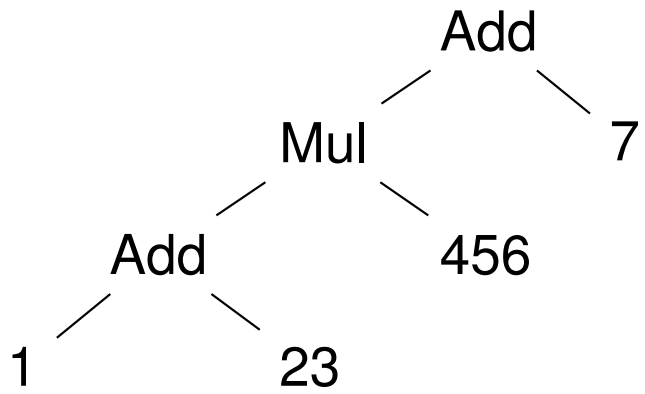
```
let rec fact = fun n ->
  if n = 0 then
    1
  else
    n * fact (n-1)
in
fact 6
```

Concrete syntax versus abstract syntax

- Concrete syntax of a programming language :
  - a sequence of characters,
  - what the programmer is allowed to write,
  - what is understood as a legitimate program by the computer

```
> (1+23)*456+7
> (1 + 23) * 456 + 7
> ( 1+23 ) *456   +7
```

- Abstract syntax :
  - a hierarchical representation of the input,
  - keeps only what is meaningful for computation



Syntactic sugar

- A construction available in the concrete syntax but not in the abstract syntax tree
- Examples :
  - x += 1 translates to x = x + 1
  - t[i] translates to *(t+i)
  - **let** f x = e translates to **let** f = **fun** x -> e

Semantics and interpretation of FUN

Inductive structures

- A mathematical way to describe tree structures and reason about them
- Given : a signature, a set of “constructors” each one comes with an arity
- Derived :
  - set of objects E build using a finite number of constructors
  - a way to define a function $f \in E \to A$ by case and using recursion
  - a awy to prove properties $\forall x \in E, P(x)$ by induction

Definition

We define a set of inductive objects with :

1. some base objects,
2. a set of constructors, that can combine already built objects to define new objects.

We then consider the set of all objects that can be built using the two previous points, only a finite number of times.

- Each construtor comes with an arity : the number of expected elements
- Base objects have an arity 0
- The signature of a set of inductive objects is the set comprising its base objects and its constructors.

Example : lists

Linked lists can be seen as a set of inductive objects defined by :

- a unique base object : the empty list, written [],
- a constructor, which builds a new list $e :: \ell$ by adding a new element e at the head of a list ℓ.

A small subtlety :

- From the theoretical point of view, we might consider we have a set of unary constructors $e ::$., one for each element e
- Alternatively, we might consider we have only one constructor which expects 2 arguments, but only the second one is recursive

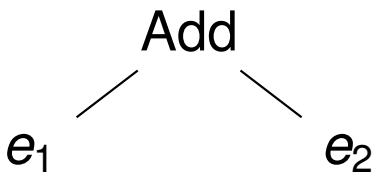
Example : arithmetic expressions.

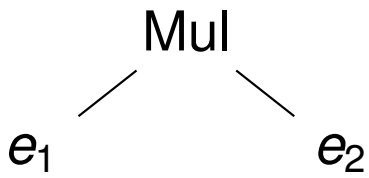
We define a minimal set of arithmetic expression $\mathbb{A}$ with :

- the integer constants as base objects,

$$0 \qquad 1 \qquad 2 \qquad 3 \qquad \cdots$$

- some binary constructors, each combining two already built expressions, such as addition or multiplication.





On can lighten the handling of such expression using mathematical notations such as n, $e_1 \oplus e_2$ or $e_1 \otimes e_2$.

Defining functions on a set of inductive objects

Each object of a set E of inductive objects can be built using only the base objects and the constructors (finitely).
A function f applicable to the elements of E can thus be defined in a very succinct way :

- for each base element e, give $f(e)$,
- for each n-ary constructor c, describe $f(c(e_1, \ldots, e_n))$ using the subelements e_i and their images $f(e_i)$ for each $e_i \in E$.

This define a unique image for each element of E.

Examples

$$\begin{cases} \text{nbCst}(\text{n}) & = & 1 \\ \text{nbCst}(e_1 \oplus e_2) & = & \text{nbCst}(e_1) + \text{nbCst}(e_2) \\ \text{nbCst}(e_1 \otimes e_2) & = & \text{nbCst}(e_1) + \text{nbCst}(e_2) \end{cases}$$

$$\begin{cases} \text{nbOp}(\text{n}) & = & 0 \\ \text{nbOp}(e_1 \oplus e_2) & = & 1 + \text{nbOp}(e_1) + \text{nbOp}(e_2) \\ \text{nbOp}(e_1 \otimes e_2) & = & 1 + \text{nbOp}(e_1) + \text{nbOp}(e_2) \end{cases}$$

$$\begin{cases} \text{eval}(\text{n}) & = & n \\ \text{eval}(e_1 \oplus e_2) & = & \text{eval}(e_1) + \text{eval}(e_2) \\ \text{eval}(e_1 \otimes e_2) & = & \text{eval}(e_1) \times \text{eval}(e_2) \end{cases}$$

Inductive objects in Ocaml

```
type 'a list =
  | []
  | (::) of 'a * 'a list
```

Expressions are defined similarly, using three constructors.

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Mul of expr * expr
```

$(1 \oplus 2) \oplus (3 \otimes 4)$ can be defined in caml by :

```
Add(Add(Cst 1, Cst 2), Mul(Cst 3, Cst 4))
```

Functions

```
let rec nb_cst = function
  | Cst n          -> 1
  | Add(e1, e2) -> nb_cst e1 + nb_cst e2
  | Mul(e1, e2) -> nb_cst e1 + nb_cst e2

let rec eval = function
  | Cst n          -> n
  | Add(e1, e2) -> eval e1 + eval e2
  | Mul(e1, e2) -> eval e1 * eval e2

let rec nb_op = function
  | Cst _                              -> 0
  | Add(e1, e2) | Mul(e1, e2) -> nb_op e1 + nb_op e2
```

Reasoning by structural induction

Proving that a given property E is valid for all elements in E reduces to :

1. proving that $P(e)$ is valid for each base element e,
2. proving, for each n-ary constructor c, and given n elements e_1, ..., e_n that the property P is valid for the combined element $c(e_1, \ldots, e_n)$ provided all $e_i \in E$ satisfy $P(e_i)$.
   In other words, for any constructor and any elements,
   $P(e_{i_1}) \wedge \ldots \wedge P(e_{i_k}) \implies P(c(e_1, \ldots, e_n))$.

Thus, we ensure that it is not possible to build an element e that does not satisfy the target property P.
In the second point hypotheses $P(e_{i_1})$ to $P(e_{i_n})$ that can be used for justifying $P(c(e_1, \ldots, e_n))$ are called induction hypotheses.

Examples of induction principles

- Proving that a property P is valid for all lists reduces to :
  1. proving that it is valid for the empty list [],
  2. for any list ℓ and any element e, proving that if P is valid for ℓ (induction hypothesis), then it is still valid for $e : \ell$.
- Proving that a property P is valid for all arithmetic expressions reduces to :
  1. proving that it is valid for all integer constants,
  2. for any expressions e_1 and e_2, proving that if P is valid for e_1 and e_2 (induction hypotheses), then it is still valid for $e_1 \oplus e_2$,
  3. for any expressions e_1 and e_2, proving that if P is valid for e_1 and e_2 (induction hypotheses), then it is still valid for $e_1 \otimes e_2$.

Example of proof by induction

$P(e)$ is the property $\text{nbCst}(e) = \text{nbOp}(e) + 1$,

- Case of a constant (base case) : for any constant n we have $\text{nbCst}(\text{n}) = 1$ and $\text{nbOp}(\text{n}) = 0$. Then the property P is satisfied by the term n.
- Case of an addition (inductive case) : let e_1 and e_2 be two expressions satisfying the property P. Then

$$
\begin{array}{lll}
& \text{nbCst}(e_1 \oplus e_2) & \\
= & \text{nbCst}(e_1) + \text{nbCst}(e_2) & \text{by definition of nbCst} \\
= & (\text{nbOp}(e_1) + 1) + (\text{nbOp}(e_2) + 1) & \text{by induction hypotheses} \\
= & (1 + \text{nbOp}(e_1) + \text{nbOp}(e_2)) + 1 & \text{(reorder)} \\
= & \text{nbOp}(e_1 \oplus e_2) + 1 & \text{by definition of nbOp}
\end{array}
$$

  Thus, the property P is still valid for the term $\text{Add}(e_1, e_2) = e_1 \oplus e_2$.
- Case of a multiplication (inductive case) : similar to the case of an addition.

Thus, using structural induction we proved that for any arithmetic expression e, we have $\text{nbCst}(e) = \text{nbOp}(e) + 1$.

Semantics and interpretation of FUN

Abstract syntax

A unique type of expressions

```
type bop = Add | Sub | Mul | Lt | Eq (* / ... *)
type expr =
  (* arithmetic *)
  | Int of int
  | Bop of bop * expr * expr
  (* variables *)
  | Var of string
  | Let of string * expr * expr
  (* conditional *)
  | If  of expr * expr * expr
  (* functions *)
  | Fun of string * expr
  | App of expr * expr
  (* recursion *)
  | Fix of string * expr
```

Abstract syntax : example

The expression **let** x = 41 **in** x+1 of FUN is represented in caml by
Let("x", Int 41, Bop(Add, Var "x", Int 1)).
The constructor Fix describes a recursive definition.
let rec f x = e1 **in** e2 will be represented in caml by
Let("f", Fix("f", Fun("x", e1)), e2).
Note that the identifier "f" of the function appears twice here.

Variables

- A name for a value that has been computed before
  - define what a value is
  - keep track of valid names
  - keep track of the link between names and values (association table)

Implementation (arithmetical part, without functions)

```
type value =
  | VInt  of int
  | VBool of bool
```

Association tables implemented usind balanced trees

```
module Env = Map.Make(String)
type env = value Env.t
```

Available objects from the module Env

- Env.empty for an empty table
- Env.find for fetching the value associated to a given key
- Env.add for adding or updating an association.

Eval function

```
let rec eval (e:expr) (env:env) : value = match e with
  | Int n -> VInt n
  | Bop(op, e1, e2) ->
     begin match op, eval e1 env, eval e2 env with
       | Add, VInt n1, VInt n2 -> VInt (n1 + n2)
       | Sub, VInt n1, VInt n2 -> VInt (n1 - n2)
       | Mul, VInt n1, VInt n2 -> VInt (n1 * n2)
       | Lt,  VInt n1, VInt n2 -> VBool (n1 < n2)
       | Eq,  v1,      v2      -> VBool (v1 = v2)
       | _ -> failwith "unauthorized operation"
     end
  | If(c, e1, e2) ->
     begin match eval c env with
     | VBool b -> if b then eval e1 env
                       else eval e2 env
     | _ -> failwith "unauthorized operation"
     end
```

Eval function (variables)

```
    | Var x -> Env.find x env
    | Let(x, e1, e2) ->
       let v1 = eval e1 env in
       let env' = Env.add x v1 env in
       eval e2 env'

let eval_top (e: expr): value = eval e Env.empty
```

Functions and functional closures

Functions are ordinary values, which can be passed to other functions as parameters, or returned as results.

```
let plus n =
  let f x = x + n in
  f
```

- a function plus defines and returns a local function f.
- The definition of f uses a variable n which is external to f (it is called a free variable).
- the variable n is a parameter of the function plus.
- Two calls plus 2 and plus 3 define two differents functions. Both correspond to the code **fun** x –> x + n, however we have $n = 2$ in the former case, and $n = 3$ in the latter.
- a function-value is a function together with an environment providing at least the values of all the external variables used in the function
- We call this function/environment pair a functional closure.

Implementation

```
type value = ...
| VClos of string * expr * env
```

The value corresponding to the function `fun x -> e` defined in the environment ρ is represented by `VClos("x", e, ρ)`.

```
let rec eval e env = match e with
  ...
  | Fun(x, e) -> VClos(x, e, env)
  | App(e1, e2) ->
      let x, e, env' = match eval e1 env with
        | VClos(x, e, env') -> x, e, env'
        | _ -> failwith "unauthorized operation"
      in
      let v2 = eval e2 env in
      eval e (Env.add x v2 env')
```

Recursion

- A recursive function f, like any ordinary function, evaluates to a functional closure $c = \texttt{VClos}(x, e, \rho)$.
- We need the environment ρ of the closure c to contain c itself (as the value of f)
- we would like

  ```
  eval (Fix(f, Fun(x, e))) env
  ```

  to produce a value `v` satisfying the recursive equation

  ```
  v = VClos(x, e, Env.add f v env)
  ```

  Not allowed in ocaml
- Solution : add a specific value for recursive functions with all needed information

Implementing recursive functions

```
type value =
  ...
  | VFix of expr * string * value * env
```

Given an expression *e*, an identifier *f*, and an environment ρ, the recursive value $v = \texttt{VFix(}e\texttt{, }f\texttt{, }v\texttt{, }\rho\texttt{)}$ is to be understood has the result of evaluating *e* in the environment `Env.add` f v ρ.

Evaluating recursive functions

- the evaluation rule for `Fix` produces a recursive value `VFix`
- an auxiliary function `force: value -> value` "opens" a recursive value during function application

Evaluation of fixpoints

```
let rec eval e env = match e with
...
  | Fix(f, Fun(x, e)) ->
     let rec v = VFix(Fun(x, e), f, v, env) in
     v
  | Fix _ -> failwith "unauthorized operation"

  | App(e1, e2) ->
     let x, e, env' = match force (eval e1 env) with
       | VClos(x, e, env') -> x, e, env'
       | _ -> failwith "unauthorized operation"
       in let v2 = eval e2 env
       in eval e (Env.add x v2 env')

and force v = match v with
| VFix(Fun(x,e), f, v, env) -> VClos(x,e,Env.add f v env)
| VClos(_,_,_) -> v
| _ -> failwith "unauthorized operation"
```

Semantics and interpretation of FUN

Natural semantics

- The evaluation function defined earlier in caml gives a meaning based on caml
- Semantics of programming languages is not always precisely defined

  > The Java programming language guarantees that the operands of operators appear to be evaluated in a specific order, namely, from left to right. It is recommended that code do not rely crucially on this specification.

- We want a formal semantics, a mathematical characterization of the computation described by a program.
- This more rigorous setting allows us to reason on the execution of programs.

Equational semantics

$$
\begin{aligned}
\text{eval}(n, \rho) &= n \\
\text{eval}(e_1 \oplus e_2, \rho) &= \text{eval}(e_1, \rho) + \text{eval}(e_2, \rho) \\
\text{eval}(x, \rho) &= \rho(x) \\
\text{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, \rho) &= \text{eval}(e_2, \rho \cup \{x \mapsto \text{eval}(e_1, \rho)\}) \\
\text{eval}(\texttt{fun } x \texttt{ -> } e, \rho) &= \text{Clos}(x, e, \rho) \\
\text{eval}(e_1\ e_2, \rho) &= \text{eval}(e, \rho' \cup \{x \mapsto \text{eval}(e_2, \rho)\}) \\
&\quad \text{if } \text{eval}(e_1, \rho) = \text{Clos}(x, e, \rho')
\end{aligned}
$$

Avoiding the environment

- Evaluate only closed values
- Substitution : $e[x := e']$ denotes the replacement of each occurrence of the variable x in the expression e by the other expression e'

$$
\begin{array}{rcl}
n[x := e'] & = & n \\
(e_1 \texttt{ + } e_2)[x := e'] & = & e_1[x := e'] \texttt{ + } e_2[x := e'] \\
y[x := e'] & = & \begin{cases} e' & \text{if } x = y \\ y & \text{otherwise} \end{cases} \\
(\texttt{let } y \texttt{ = } e_1 \texttt{ in } e_2)[x := e'] & = & \begin{cases} \texttt{let } y \texttt{ = } e_1[x := e'] \texttt{ in } e_2 \\ \texttt{let } y \texttt{ = } e_1[x := e'] \texttt{ in } e_2[x := \end{cases} \\
(\texttt{fun } y \texttt{ -> } e)[x := e'] & = & \begin{cases} \texttt{fun } y \texttt{ -> } e & \text{if } x = y \\ \texttt{fun } y \texttt{ -> } e[x := e'] & \text{otherwis} \end{cases} \\
(e_1\ e_2)[x := e'] & = & e_1[x := e']\ e_2[x := e']
\end{array}
$$

For the rules `let` and `fun` in the case $x \neq y$ are only applicable when $y \notin \text{fv}(e')$.
y being a bound (mute) variable, it can be replaced by a fresh name if needed.

Evaluation of closed expressions

$$
\begin{array}{rcl}
\text{eval}(n) & = & n \\
\text{eval}(e_1 \oplus e_2) & = & \text{eval}(e_1) + \text{eval}(e_2) \\
\text{eval}(x) & = & \text{indéfini} \\
\text{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) & = & \text{eval}(e_2[x := \text{eval}(e_1)]) \\
\text{eval}(\texttt{fun } x \texttt{ -> } e) & = & \texttt{fun } x \texttt{ -> } e \\
\text{eval}(e_1\ e_2) & = & \text{eval}(e[x := \text{eval}(e_2)]) \\
 & & \text{if eval}(e_1) = \texttt{fun } x \texttt{ -> } e
\end{array}
$$

Natural semantics

- Also called big step semantics
- Use a relation between expressions and values ($e \implies v$) instead of a function in order to define the semantics
- Use inference rules to define the relation
  - a proof of a concrete case $e \implies v$ is represented as a tree
  - associated induction principle to use an hypothesis $e \implies v$
- the set of values : integer numbers, and functions.

$$\begin{array}{rcl} v & ::= & n \\ & | & \texttt{fun}\ x\ \texttt{->}\ e \end{array}$$

Rules

$$\frac{}{n \implies n}$$

$$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1 \oplus e_2 \implies n_1 + n_2}$$

$$\frac{e_1 \implies v_1 \qquad e_2[x := v_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

$$\frac{}{\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e}$$

$$\frac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e_2 \implies v_2 \qquad e[x := v_2] \implies v}{e_1\ e_2 \implies v}$$

Example

$$\dfrac{\dfrac{}{\texttt{fun}\ x\texttt{->}x \oplus x \implies \texttt{fun}\ x\texttt{->}x \oplus x} \qquad \dfrac{\vdots}{(\texttt{fun}\ x\ \texttt{->}\ x \oplus x)\ (20 \oplus 1) \implies 42}}{\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x \oplus x\ \texttt{in}\ f\ (20 \oplus 1) \implies 42}$$

$$\dfrac{\dfrac{}{\texttt{fun}\ x\texttt{->}x \oplus x \implies \texttt{fun}\ x\texttt{->}x \oplus x} \qquad \dfrac{\dfrac{\dfrac{}{20 \implies 20} \qquad \dfrac{}{1 \implies 1}}{20 \oplus 1 \implies 21} \qquad \dfrac{\dfrac{}{21 \implies 21} \qquad \dfrac{}{21 \implies 21}}{21 \oplus 21 \implies 42}}{(\texttt{fun}\ x\ \texttt{->}\ x \oplus x)\ (20 \oplus 1) \implies 42}}{\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x \oplus x\ \texttt{in}\ f\ (20 \oplus 1) \implies 42}$$

Call by value versus call by name

- The argument of a function is evaluated before executing the body
- Alternatively

$$\frac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e[x := e_2] \implies v}{e_1\ e_2 \implies v}$$

$$\frac{e_2[x := e_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

Example

$$\dfrac{\dfrac{}{\texttt{fun}\ x\texttt{->}x \oplus x \implies \texttt{fun}\ x\texttt{->}x \oplus x} \qquad \dfrac{\dfrac{}{20 \implies 20} \quad \dfrac{}{1 \implies 1}}{20 \oplus 1 \implies 21} \quad \dfrac{\dfrac{}{20 \implies 20} \quad \dfrac{}{1 \implies 1}}{20 \oplus 1 \implies 21}}{(20 \oplus 1) \oplus (20 \oplus 1) \implies 42}}{(\texttt{fun}\ x\ \texttt{->}\ x \oplus x)\ (20 \oplus 1) \implies 42}}{\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x \oplus x\ \texttt{in}\ f\ (20 \oplus 1) \implies 42}$$

Reasoning on semantics

- Prove that an hypothesis $e \implies v$ implies a property $P(e, v)$
- One case by inference rule

Example

Let us consider the call by name semantics for FUN

$$\frac{}{n \implies n} \qquad \frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1 \oplus e_2 \implies n_1 + n_2} \qquad \frac{e_2[x := e_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

$$\frac{}{\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e}$$

$$\frac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e[x := e_2] \implies v}{e_1\ e_2 \implies v}$$

and prove that if $e \implies v$, then v is value such that $\mathrm{fv}(v) \subseteq \mathrm{fv}(e)$, by induction on the derivation of $e \implies v$.

Proof

- Case $n \Longrightarrow n$: immediate
- Case $\texttt{fun}\ x\ \texttt{->}\ e \Longrightarrow \texttt{fun}\ x\ \texttt{->}\ e$: immediate as well.
- Case $e_1 \oplus e_2 \Longrightarrow n_1 + n_2$ with $e_1 \Longrightarrow n_1$ and $e_2 \Longrightarrow n_2$. By definition $n_1 + n_2$ is an integer value. Moreover $\mathrm{fv}(n_1 + n_2) = \emptyset \subseteq \mathrm{fv}(e_1 \oplus e_2)$.
- Case $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Longrightarrow v$ with $e_2[x := e_1] \Longrightarrow v$.
  induction hypothesis : $\mathrm{fv}(v) \subseteq \mathrm{fv}(e_2[x := e_1])$ (and v is a value).
  Lemma :
  $$\mathrm{fv}(e[x := e']) \subseteq (\mathrm{fv}(e) \setminus \{ x \}) \cup \mathrm{fv}(e')$$
  Using the lemma, we get $\mathrm{fv}(v) \subseteq (\mathrm{fv}(e_2) \setminus \{ x \}) \cup \mathrm{fv}(e_1)$.
  We have
  $$\mathrm{fv}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus x)$$
  Then $\mathrm{fv}(v) \subseteq \mathrm{fv}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2)$.
- Case $e_1\ e_2 \Longrightarrow v$ with $e_1 \Longrightarrow \texttt{fun}\ x\ \texttt{->}\ e$ and $e[x := e_2] \Longrightarrow v$.
  induction hypotheses v is a value, and $\mathrm{fv}(\texttt{fun}\ x\ \texttt{->}\ e) \subseteq \mathrm{fv}(e_1)$ and $\mathrm{fv}(v) \subseteq \mathrm{fv}(e[x := e_2])$.

$$\begin{array}{rcl} \mathrm{fv}(v) & \subseteq & \mathrm{fv}(e[x := e_2]) \\ & = & (\mathrm{fv}(e) \setminus \{ x \}) \cup \mathrm{fv}(e_2) \\ & = & \mathrm{fv}(\texttt{fun}\ x\ \texttt{->}\ e) \cup \mathrm{fv}(e_2) \end{array}$$

Semantics and interpretation of FUN

Small step operational semantics

- Natural semantics speaks only about computations that succeed.
- Small step semantics, or reduction semantics, provides finer information by decomposing the evaluation $e \Longrightarrow v$ in a sequence of computation steps $e \to e_1 \to e_2 \to \ldots \to v$.
- Three possible behaviors
  - a computation reaches a result (value) :

$$e \to e_1 \to e_2 \to \ldots \to v$$

  - a computation which, stumbles on a failure state :

$$e \to e_1 \to e_2 \to \ldots \to e_n$$

    where e_n is not a value, but cannot be evaluated further,
  - a computation that never ends :

$$e \to e_1 \to e_2 \to \ldots$$

Computation rules

- a binary relation $e \to e'$ called reduction relation, describing a single step of computation.
- defined by a set of inference rules
- elementary computation rules, giving base cases
- inference rules that allow the application of a computation rule in a subexpression.

Rules

Basic computational rules

$$\frac{}{(\texttt{fun}\ \ x\ \ \texttt{->}\ \ e)\ v \to e[x := v]}$$

$$\frac{}{\texttt{let}\ \ x\ \ \texttt{=}\ \ v\ \ \texttt{in}\ \ e \to e[x := v]}$$

$$\frac{n_1 + n_2 = n}{n_1 \oplus n_2 \to n}$$

Dealing with subexpressions

$$\frac{e_1 \to e_1'}{e_1 \oplus e_2 \to e_1' \oplus e_2} \qquad \frac{e_2 \to e_2'}{e_1 \oplus e_2 \to e_1 \oplus e_2'}$$

Forcing the order of evaluation

$$\frac{e_2 \to e_2'}{v_1 \oplus e_2 \to v_1 \oplus e_2'}$$

Rules

Local variables and application

$$\frac{e_1 \to e_1'}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$$

Vocabulary

- $e \to e'$ when 1 computation step leads from e to e', and
- $e \to^* e'$ when a computation leads from e to e' using 0, 1 or more steps (this is called a computation sequence or a reduction sequence).
- An irreducible expression is an expression e from which no reduction step can take place, that is such that there is no expression e' such that $e \to e'$.
  - `let f = fun x -> x + x in f (20 + 1)`
  - `let f = fun x -> fun y -> x + y in 1 + f 2`

Semantics and interpretation of FUN

Equivalence between small step and big step

$$e \Longrightarrow v \qquad \text{if and only if} \qquad e \to^* v$$

- $e \Longrightarrow v$ implies $e \to^* v$: by induction on the derivation of $e \Longrightarrow v$
- $e \to^* v$ implies $e \Longrightarrow v$ (with v a value) : by induction on the length of the reduction sequence $e \to^* v$.
  - Lemma : if $e \to e'$ and $e' \Longrightarrow v$ then $e \Longrightarrow v$.
    Proof by induction on the derivation of $e \to e'$.

(see full proof in the course notes)

Rules : big step

$$\frac{}{n \Longrightarrow n} \qquad \frac{}{\texttt{fun } x \texttt{ -> } e \Longrightarrow \texttt{fun } x \texttt{ -> } e}$$

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{e_1 \oplus e_2 \Longrightarrow n_1 + n_2} \qquad \frac{e_1 \Longrightarrow v_1 \qquad e_2[x := v_1] \Longrightarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow v}$$

$$\frac{e_1 \Longrightarrow \texttt{fun } x \texttt{ -> } e \qquad e_2 \Longrightarrow v_2 \qquad e[x := v_2] \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$

Rules : small step

$$\frac{e_1 \to e_1'}{e_1 \oplus e_2 \to e_1' \oplus e_2} \qquad \frac{e_2 \to e_2'}{v_1 \oplus e_2 \to v_1 \oplus e_2'} \qquad \frac{n_1 + n_2 = n}{n_1 \oplus n_2 \to n}$$

$$\frac{e_1 \to e_1'}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \to \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2}$$

$$\frac{}{\texttt{let } x \texttt{ = } v \texttt{ in } e \to e[x := v]}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \frac{}{(\texttt{fun } x \texttt{ -> } e)\ v \to e[x := v]}$$

Exercises

- formalize the semantics of `if` and `fix`, both in big step and in small step style ;
- extend FUN with lazy operators such as `||` and `&&`, which do not evaluate their second operand when the first one is sufficent for the final result

Summary

- Abstract syntax
- Inductive sets (definition of functions, proofs by induction)
- Semantics of a functional language (dealing with (recursive) closure
- Mathematical definition of semantics
  - using inference rules
  - big step/small step semantics
  - example of proofs on semantics