Compiler

Abstract syntax, semantics

Jyun-Ao Lin

iFIRST & CSIE, NTUT

Credits

A large part of this course is based on the Compilation Course of J.-C. Filliâtre at ENS Ulm.

Overview of the course

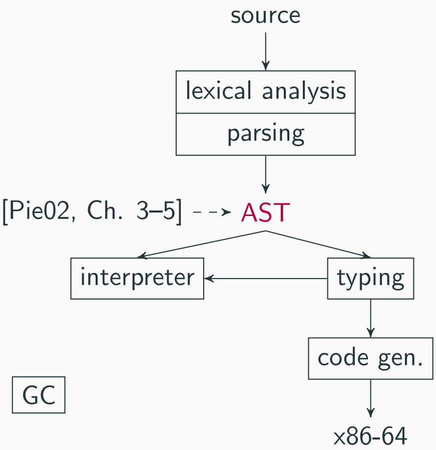


Table of contents

Abstract Syntax

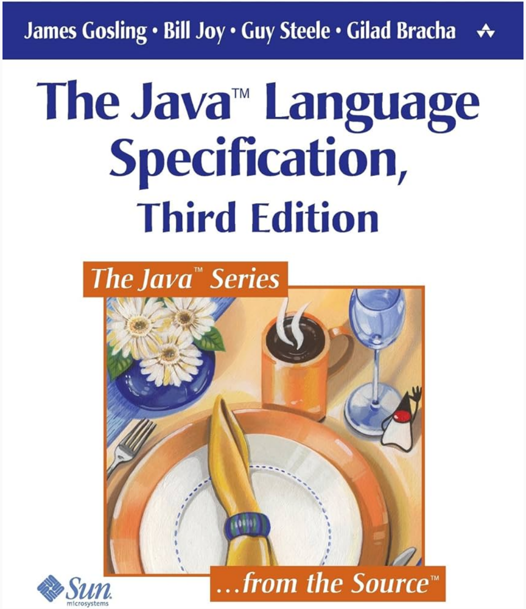
Meaning

How to define the meaning of programs?

Most of the time, we are satisfied with an informal specification, in natural language (e.g. ISO norm, standard, reference book)

Yet it is imprecise, sometimes even ambiguous

Informal Semantics



The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

It is recommended that code not rely crucially on this specification.

Formal Semantics

The formal semantics gives a mathematical characterization of the computations defined by a program.

Useful to make tools (interpreters, compilers, etc.)

Necessary to reason about programs.

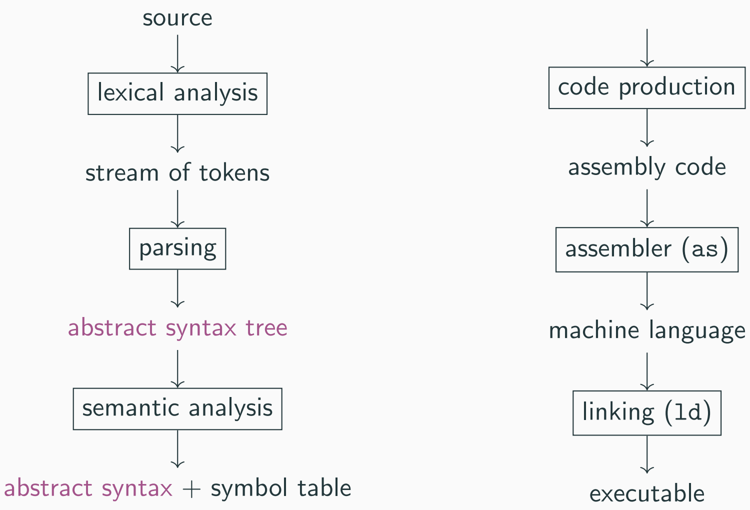
Raise Another Question

What is a program?

As a syntactic object (sequence of characters),
it is too complex to apprehend

That's why we switch to abstract syntax.

Abstract Syntax



Abstract Syntax

The text

```
2*(x+1)
```

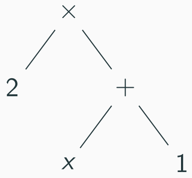
and

```
(2 * ((x)+1))
```

and

```
2 * (* I double it *) ( x + 1 )
```

all map to the same abstract syntax tree



Notation

We define the abstract syntax using a grammar

$$\begin{array}{rcll} e & ::= & c & \text{constant} \\ & | & x & \text{variable} \\ & | & e + e & \text{addition} \\ & | & e \times e & \text{multiplication} \\ & | & \dots & \end{array}$$

It reads "an expression, denoted by e, is

- either a constant c,
- either a variable x,
- either the addition of two expressions,
- etc."

Here $c, x, e, \dots$ etc are called meta-variables, that is, c denotes a whatever constant, ...etc

Notation

The notation $e_1 + e_2$ of the abstract syntax borrows the symbol of the concrete syntax.

But we could have picked something else, e.g. $Add(e_1, e_2), +(e_1, e_2)$, etc.

Abstract Syntax in Java

We use classes to build abstract syntax trees, as follows:

```
enum Binop { Add, Mul, ... }

abstract class Expr {}
class Cte extends Expr { int n; }
class Var extends Expr { string x; }
class Bin extends Expr { Binop op; Expr e1, e2; }
...
```

(constructors are omitted)

expression `2 * (x + 1)` is then represented as

```
new Bin(Mul, new Cte(2), new Bin(Add, new Var("x"), new Cte(1)))
```

Abstract Syntax in OCaml

We use algebraic data types to abstract syntax trees, as follows:

```
type binop = Add | Mul | ...

type expr =
    | Cte of int
    | Var of string
    | Bin of binop * expr * expr
    | ...
```

Expression `2 * (x + 1)` is then represented as

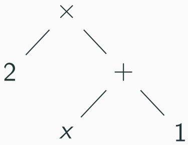
```
Bin (Mul, Cte 2, Bin (Add, Var "x", Cte 1))
```

Parentheses

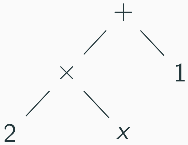
There is no construct for parentheses in abstract syntax

In concrete syntax `2 * (x + 1)`,

parentheses are used to build this tree



rather than this one



(the lecture on parsing will explain how)

Syntactic Sugar

We call syntactic sugar a construct of concrete syntax that does not exist in abstract syntax.

It is thus translated in terms of other constructs of abstract syntax
(typically during parsing)

Example

- In `C`, expression `a[i]` is syntactic sugar for `*(a+i)`.
- In `Java`, expression `x -> ...` is syntactic sugar for the construction of an object in some anonymous class that implements `Function`.
- In `OCaml`, expression `[`e_1`; `e_2`; ...; `e_n`]` is sugar for e_1 `::` e_2 `::` `...` `::` e_n `::[]`.

Operational Semantics

Semantics

Formal semantics is defined over abstract syntax

There are many approaches

- Axiomatic semantics
- Denotational semantics
- Semantics by translation
- Operational semantics

Axiomatic Semantics

Axiomatic semantics is also called Floyd-Hoare logic
(Robert Floyd, *Assigning meanings to programs*, 1967
Tony Hoare, *An axiomatic basis for computer programming*, 1969)

It defines programs by means of their properties; we introduce a triple

$$\{P\}\ i\ \{Q\}$$

meaning "if formula P holds before the execution of statement i, then formula Q holds after the execution".

Example

- Example

$$\{x \geq 0\}\ x := x + 1\ \{x > 0\}$$

- Example of rule:

$$\{P[x \leftarrow E]\} x := E \{P(x)\}$$

Denotational Semantics

Denotational semantics maps each program expression e to its denotation $[\![e]\!]$, a mathematical object that represents the computation denoted by e.

Example

Arithmetic expressions with a single variable $\mathtt{x}$

$$e ::= \mathtt{x} \mid n \mid e + e \mid e * e \mid \dots$$

the denotation is a function that maps the value of $\mathtt{x}$ to the value of the expression

$$\begin{aligned}
[\![\mathtt{x}]\!] &= x \mapsto x \\
[\![n]\!] &= x \mapsto n \\
[\![e_1 + e_2]\!] &= x \mapsto [\![e_1]\!](x) + [\![e_2]\!](x) \\
[\![e_1 * e_2]\!] &= x \mapsto [\![e_1]\!](x) \times [\![e_2]\!](x)
\end{aligned}$$

Semantics by Translation

(also called Strachey semantics)

We can define the semantics of a language by means of its translation to another language for which the semantics is already defined.

Example of Semantics by Translation

An esoteric language whose syntax consists of 8 characters and whose semantics is defined by translation to the C language

command	translation to C
(prelude)	`char array[3000] = 0` `char *ptr = array;`
>	`++ptr;`
<	`--ptr;`
+	`++*ptr;`
-	`--*ptr;`
.	`putchar(*ptr);`
,	`*ptr = getchar();`
[	`while (*ptr) {`
]	`}`

Operational Semantics

Operational semantics describes the sequence of elementary computations from the expression to its outcome (its value)

It operates directly over abstract syntax

Two kinds of operational semantics

- "natural semantics" or "big-steps"

$$e \twoheadrightarrow v$$

- "reduction semantics" or "small-steps"

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$

mini-ML

Let us illustrate big-steps operational semantics on the language Mini-ML

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
  & | & c & \text{constant } (1, 2, \ldots, \texttt{true}, \ldots) \\
  & | & op & \text{primitive operator } (+, \times, \texttt{fst}, \ldots) \\
  & | & \texttt{fun } x \rightarrow e & \text{function} \\
  & | & e\ e & \text{application} \\
  & | & (e, e) & \text{pair} \\
  & | & \texttt{let } x = e \texttt{ in } e & \text{local let}
\end{array}
$$

Example

```
let compose = fun f -> fun g -> fun x -> f (g x) in
let plus = fun x -> fun y -> + (x,y) in
compose (plus 2) (plus 4) 36
```

```
let distr_pair = fun f -> fun p -> (f (fst p), f (snd p)) in
let p = distr_pair (fun x -> x) (40,2) in
+ (fst p,snd p)
```

Another construct : conditional

The conditional can be defined as

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \quad \stackrel{\text{def}}{=} \quad \mathit{opif}(e_1, ((\texttt{fun}\ \_ \rightarrow e_2), (\texttt{fun}\ \_ \rightarrow e_3)))$$

where *opif* is a primitive.

The branches are frozen using function.

Another construct : recursive

Similarly, the recursive can be defined as

$$\texttt{rec}\ f\ x = e \quad \stackrel{\text{def}}{=} \quad \mathit{opfix}(\texttt{fun}\ f \to \texttt{fun}\ x \to e)$$

where *opfix* is an operator of fixed point, satisfying

$$\mathit{opfix}\ f = f\ (\mathit{opfix}\ f)$$

Example

$$\begin{aligned}\mathit{opfix}\ (\texttt{fun}\ \mathit{fact} \to \texttt{fun}\ n \to \texttt{if}\ n = 0\ \texttt{then}\ 1 \\ \texttt{else}\ \times\ (n, \mathit{fact}(-(n, 1))))\end{aligned}$$

Big-steps operational semantics of mini-ML

We seek to define a relation between some expression e and a value v

$$e \twoheadrightarrow v$$

Here, values are defined as

$$\begin{array}{lcll} v & ::= & c & \text{constant} \\ & | & op & \text{not applied primitive} \\ & | & \texttt{fun}\ x \rightarrow e & \text{function} \\ & | & (v, v) & \text{pair} \end{array}$$

To define $e \twoheadrightarrow v$, one needs the notions of inference rules and of substitution.

Inference rules

A relation may be defined as the smallest relation satisfying a set of rules with no premises (axioms) written

$$\frac{}{P}$$

and a set of rules with premises written

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_n}{P}$$

This is called inference rules.

Example

We can define the relation $\texttt{Even}(n)$ with two rules

$$\frac{}{\texttt{Even}(0)} \qquad \textit{and} \qquad \frac{\texttt{Even}(n)}{\texttt{Even}(n+2)}$$

that reads as follows

$$\begin{array}{rl} \text{on the one hand} & \texttt{Even}(0) \\ \text{on the other hand} & \forall n.\ \texttt{Even}(n) \implies \texttt{Even}(n+2) \end{array}$$

The smallest relation satisfying these two properties coincide with the property "n is an even natural number":

- even natural numbers are included, by induction
- if odd numbers were included, we could remove the smallest

Derivation Tree

A derivation is a tree whose internal nodes are rules with premises and whose leaves are axioms.

Example

$$\cfrac{\cfrac{\cfrac{}{\mathtt{Even}(0)}}{\mathtt{Even}(2)}}{\mathtt{Even}(4)}$$

The set of derivations characterizes the smallest relation satisfying the inference rules.

Free Variables

Defn. (Free Variables)
The set of free variables of an expression e, denoted by $fv(e)$, is defined recursively over e as follows:

$$
\begin{array}{rcl}
fv(x) & = & \{x\} \\
fv(c) & = & \emptyset \\
fv(op) & = & \emptyset \\
fv(\texttt{fun}\ x \rightarrow e) & = & fv(e) \setminus \{x\} \\
fv(e_1\ e_2) & = & fv(e_1) \cup fv(e_2) \\
fv((e_1, e_2)) & = & fv(e_1) \cup fv(e_2) \\
fv(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & fv(e_1) \cup (fv(e_2) \setminus \{x\})
\end{array}
$$

An expression without free variables is called close. □

Examples

Example

- $fv(\texttt{let}\ x = +(20,1)\ \texttt{in}\ (\texttt{fun}\ y \to +(y,y))x) = \emptyset$
- $fv(\texttt{let}\ x = z\ \texttt{in}\ (\texttt{fun}\ y \to (x\ y)t)) = \{z, t\}$

Substitution

Defn. (Substitution)
Let e be an expression, x a variable and v a value. We denote by $e[x \leftarrow v]$ the substitution of any free occurrence of x in e by v and is defined by

$$
\begin{array}{rcl}
x[x \leftarrow v] & = & v \\
y[x \leftarrow v] & = & y \text{ if } y \neq x \\
c[x \leftarrow v] & = & c \\
op[x \leftarrow v] & = & op \\
(\texttt{fun } x \rightarrow e)[x \leftarrow v] & = & \texttt{fun } x \rightarrow e \\
(\texttt{fun } y \rightarrow e)[x \leftarrow v] & = & \texttt{fun } y \rightarrow e[x \leftarrow v] \text{ if } y \neq x \\
(e_1\ e_2)[x \leftarrow v] & = & (e_1[x \leftarrow v]\ e_2[x \leftarrow v]) \\
(e_1, e_2)[x \leftarrow v] & = & (e_1[x \leftarrow v], e_2[x \leftarrow v]) \\
(\texttt{let } x = e_1 \texttt{ in } e_2)[x \leftarrow v] & = & (\texttt{let } x = e_1[x \leftarrow v] \texttt{ in } e_2) \\
(\texttt{let } y = e_1 \texttt{ in } e_2)[x \leftarrow v] & = & (\texttt{let } y = e_1[x \leftarrow v] \texttt{ in } e_2[x \leftarrow v]) \text{ if } y \neq x
\end{array}
$$

□

Examples

Example

- $((\texttt{fun}\ x \to +(x,x))x)[x \leftarrow 21] = (\texttt{fun}\ x \to +(x,x))21$
- $(+(x, \texttt{let}\ x = 17\ \texttt{in}\ x))[x \leftarrow 3] = +(3, \texttt{let}\ x = 17\ \texttt{in}\ x)$
- $(\texttt{fun}\ y \to y\ y)[y \leftarrow 17] = \texttt{fun}\ y \to y\ y$

Natural semantics of mini-ML

$$\frac{}{c \twoheadrightarrow c} \qquad \frac{}{op \twoheadrightarrow op} \qquad \frac{}{(\texttt{fun}\ \ x \to e) \twoheadrightarrow (\texttt{fun}\ \ x \to e)}$$

$$\frac{e_1 \twoheadrightarrow v_1 \qquad e_2 \twoheadrightarrow v_2}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)} \qquad \frac{e_1 \twoheadrightarrow v_1 \qquad e_2[x \leftarrow v_1] \twoheadrightarrow v}{\texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2 \twoheadrightarrow v}$$

$$\frac{e_1 \twoheadrightarrow (\texttt{fun}\ \ x \to e) \qquad e_2 \twoheadrightarrow v_2 \qquad e[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

Note: we have chosen a strategy of call by value. i.e. the argument is completely evaluated before the call.

Primitive

We have to add the rules for the primitives, for example,

$$\frac{e_1 \twoheadrightarrow + \qquad e_2 \twoheadrightarrow (n_1, n_2) \qquad n = n_1 + n_2}{e_1\ e_2 \twoheadrightarrow n}$$

$$\frac{e_1 \twoheadrightarrow opif \qquad e_2 \twoheadrightarrow (\texttt{true}, ((\texttt{fun}\ \_ \rightarrow e_3), (\texttt{fun}\ \_ \rightarrow e_4))) \qquad e_3 \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

$$\frac{e_1 \twoheadrightarrow opfix \qquad e_2 \twoheadrightarrow (\texttt{fun}\ f \rightarrow e) \qquad e[f \leftarrow opfix(\texttt{fun}\ f \rightarrow e)] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

$$\frac{e_1 \twoheadrightarrow fst \qquad e_2 \twoheadrightarrow (v_1, v_2)}{e_1\ e_2 \twoheadrightarrow v_1}$$

Example of Derivation

$$
\dfrac{
  \dfrac{
    \dfrac{}{+ \twoheadrightarrow +}
    \qquad
    \dfrac{\dfrac{}{20 \twoheadrightarrow 20} \quad \dfrac{}{1 \twoheadrightarrow 1}}{(20,1) \twoheadrightarrow (20,1)}
  }{+(20,1) \twoheadrightarrow 21}
  \qquad
  \dfrac{
    \dfrac{}{\texttt{fun}\ \ldots \twoheadrightarrow \texttt{fun}\ \ldots}
    \quad
    \dfrac{}{21 \twoheadrightarrow 21}
    \quad
    \dfrac{\vdots}{+(21,21) \twoheadrightarrow 42}
  }{(\texttt{fun}\ y \rightarrow +(y,y))21 \twoheadrightarrow 42}
}{\texttt{let}\ x = +(20,1)\ \texttt{in}\ (\texttt{fun}\ y \rightarrow +(y,y))x \twoheadrightarrow 42}
$$

Exercise

Give the derivation of

$$(opfix\ F)\ 2$$

where F is defined as

$$\texttt{fun}\ fact \rightarrow \texttt{fun}\ n \rightarrow\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ \times(n, fact(-(n, 1)))$$

Expression without value

There are expressions e for which there is no value v such that $e \twoheadrightarrow v$.

Example

- $e = 1\ 2$
- $e = (\texttt{fun}\ \ x \to x\ x)\ (\texttt{fun}\ \ x \to x\ x)$

Induction over the derivation

To establish a property of a relation defined by a set of inference rules, we can reason by induction on the derivation.

That is, by structural induction, i.e. we can apply the induction hypothesis to any subderivation. (Equivalently, we may say that we reason by induction on the height of derivation)

In practice, we proceed by induction on the derivation and by case on the last rule used.

Properties of the Natural Semantics of mini-ML

Prop.
If $e \twoheadrightarrow v$ then v is a value.

Moreover, if e is close, then so is v. □

Proof.
By induction on the derivation $e \twoheadrightarrow v$. Consider the case of an application,

$$\frac{\begin{array}{c}(D_1)\\ \vdots \\ e_1 \twoheadrightarrow (\texttt{fun}\ x \rightarrow e_3)\end{array} \qquad \begin{array}{c}(D_2)\\ \vdots \\ e_2 \twoheadrightarrow v_2\end{array} \qquad \begin{array}{c}(D_3)\\ \vdots \\ e_3[x \leftarrow v_2] \twoheadrightarrow v\end{array}}{e_1\ e_2 \twoheadrightarrow v}$$

If $e = e_1\ e_2$ is close, then e_1 and e_2 are close. By induction hypothesis, we have

- $(\texttt{fun}\ x \rightarrow e_3)$ is a close value.
- v_2 is a close value.

Since $(\texttt{fun}\ x \rightarrow e_3)$ is close, the only free variable in e_3 is x. Thus $e_3[x \leftarrow v_2]$ is close. By induction hypothesis, v is close. The other cases are left as exercises. ■

Properties of natural semantics of mini-ML

Prop. (evaluation is deterministic)
If $e \twoheadrightarrow v$ and $e \twoheadrightarrow v'$ then $v = v'$. □

Proof.
We proceed by induction on the derivations of $e \twoheadrightarrow v$ and $e \twoheadrightarrow v'$. Let us examine the case of a pair $e = (e_1, e_2)$.

$$\frac{\begin{array}{c}(D_1)\\ \vdots \\ e_1 \twoheadrightarrow v_1\end{array} \qquad \begin{array}{c}(D_2)\\ \vdots \\ e_2 \twoheadrightarrow v_2\end{array}}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)} \qquad\qquad \frac{\begin{array}{c}(D'_1)\\ \vdots \\ e_1 \twoheadrightarrow v'_1\end{array} \qquad \begin{array}{c}(D'_2)\\ \vdots \\ e_2 \twoheadrightarrow v'_2\end{array}}{(e_1, e_2) \twoheadrightarrow (v'_1, v'_2)}$$

By induction hypothesis, we have $v_1 = v'_1$ and $v_2 = v'_2$. Hence $v = (v_1, v_2) = (v'_1, v'_2) = v'$.

The other cases are left as exercises. ■

Determinism

Remark.
An evaluation relation is not necessarily deterministic. □

Example
We add a primitive *random* and the rule

$$\frac{e_1 \twoheadrightarrow random \qquad e_2 \twoheadrightarrow n_1 \qquad 0 \leq n < n_1}{e_1\ e_2 \twoheadrightarrow n}$$

then we can have *random* $2 \twoheadrightarrow 0$ but also *random* $2 \twoheadrightarrow 1$.

Interpreter

Interpreter

We can code an interpreter following the rules of natural semantics.

We use a type for the abstract syntax of expressions

```
type expression = ...
```

and we will define a function

```
val eval: expression -> expression
```

corresponding to the relation $\twoheadrightarrow$ (because of the determinism of natural semantics)

Interpreter for mini-ML

```
type expression =
  | Var of string
  | Const of int
  | Op of string
  | Fun of string * expression
  | App of expression * expression
  | Pair of expression * expression
  | Let of string * expression * expression
```

Interpreter for mini-ML

We have to code the substitution operation $e[x \leftarrow v]$

```
val subst: expression -> string -> expression -> expression
```

We suppose that v is closed.

```
let rec subst e x v =
  match e with
  | Var y -> if y = x then v else e
  | Const _ | Op _ -> e
  | Fun (y, e1) -> if y = x then e else Fun (y, subst e1 x v)
  | App (e1, e2) -> App (subst e1 x v, subst e2 x v)
  | Pair (e1, e2) -> Pair (subst e1 x v, subst e2 x v)
  | Let (y, e1, e2) ->
      Let (y, subst e1 x v, if y = x then e2 else subst e2 x v)
```

Interpreter for mini-ML

The natural semantics is realized by the function

```
val eval: expression -> expression
```

```
let rec eval = function
  | Const _ | Op _ | Fun _ as v ->   v
  | Pair (e1, e2) -> Pair (eval e1, eval e2)
  | Let(x, e1, e2) -> eval (subst e2 x (eval e1))
  | App (e1, e2) ->
    begin match eval e1 with
      | Fun (x, e) -> eval (subst e x (eval e2))
      | Op "+" ->
        let (Pair (Const n1, Const n2)) = eval e2 in
        Const (n1 + n2)
      | Op "fst" ->
        let (Pair(v1, v2)) = eval e2 in v1
      | Op "snd" ->
        let (Pair(v1, v2)) = eval e2 in v2
    end
```

Example of evaluation

```
# eval
    (Let
      ("x",
       App (Op "+", Pair (Const 1, Const 20)),
       App (Fun ("y", App (Op "+", Pair (Var "y", Var "y"))),
            Var "x")));;
```

```
- : expression = Const 42
```

Interpreter for mini-ML

the pattern matching is intentionally non-exhaustive

```
# eval (Var "x");;
```

```
# eval (App (Const 1, Const 2));;
```

```
Exception: Match_failure ("", 87, 6).
```

we might prefer an `option` type, an explicit exception, etc.

Interpreter for mini-ML

The evaluation may not terminate

For example

$$(\texttt{fun}\ x \rightarrow x\ x)\ (\texttt{fun}\ x \rightarrow x\ x)$$

```
# let b = Fun ("x", App (Var "x", Var "x")) in
    eval (App (b, b));;
```

```
Interrupted.
```

Exercise

Add the operators *opif* and *opfix* into this interpreter.

Little interlude

Our mini-ML interpreter is not very efficient because it spends its time making substitutions and therefore reconstructing expressions.

Question: Can we avoid the substitution operation?

Idea: we interpret the expression `e` using an environment providing the current value of each value (i.e. a dictionary)

```
val eval: environment -> expression -> value
```

Difficulty: the result of

$$\texttt{let}\ x = 1\ \texttt{in}\ \texttt{fun}\ y \rightarrow +(x, y)$$

is a function which has to "remember" that $x = 1$.

Response: we have to use a closure

Little interlude

We use the module `Map` for the environment

```
module Smap = Map.Make(String)
```

We define a new type for the values

```
type value =
  | Vconst of int
  | Vop    of string
  | Vpair  of value * value
  | Vfun   of string * environment * expression

and environment = value Smap.t
```

```
val eval: environment -> expression -> value
```

Little interlude

```
let rec eval env = function
  | Const n -> Vconst n
  | Op op -> Vop op
  | Pair (e1, e2) -> Vpair (eval env e1, eval env e2)
  | Var x -> Smap.find x env
  | Let (x, e1, e2) -> eval (Smap.add x (eval env e1) env) e2
  | Fun (x, e) -> Vfun (x, env, e)
  | App (e1, e2) ->
      begin match eval env e1 with
        | Vfun (x, clos, e) -> eval (Smap.add x (eval env e2) clos) e
        | Vop "+" ->
          let Vpair (Vconst n1, Vconst n2) = eval env e2 in
          Vconst (n1 + n2)
        | Vop "fst" -> let Vpair (v1, _) = eval env e2 in v1
        | Vop "snd" -> let Vpair (_, v2) = eval env e2 in v2
      end
```

Note: this is how we will do when we compile ML in the later lecture

Exercise

Add the operator *opif* into this interpreter.

note: adding the operator *opfix* is much more complicated

Weaknesses of natural semantics

Natural semantics makes no distinction between programs that crash, such as

$$1\ 2$$

and programs whose evaluation does not terminate, such as

$$(\texttt{fun}\ x \rightarrow x\ x)\ (\texttt{fun}\ x \rightarrow x\ x)$$

Small-steps operational semantics

Small-step operational semantics remedies this by introducing a notion of elementary computation step $e_1 \rightarrow e_2$, which we will iterate

then we can distinguish

1. successful termination

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$

2. evaluation stuck on an irreducible e_n which is not a value

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n$$

3. non-terminating evaluation

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots$$

Small-steps operational semantics for mini-ML

We start by defining a simpler relation $\stackrel{\epsilon}{\rightarrow}$, corresponding to a "head" reduction, i.e., involving the most external construction of the expression.

Two rules:

$$(\texttt{fun}\ x \rightarrow e)\ v \stackrel{\epsilon}{\rightarrow} e[x \leftarrow v]$$
$$\texttt{let}\ x = v\ \texttt{in}\ e \stackrel{\epsilon}{\rightarrow} e[x \leftarrow v]$$

Note: These two rules reflect the choice of a strategy of call by value, as for semantics in big steps.

Small-steps operational semantics for mini-ML

We also have reduction rules in mind for the primitives:

$$
\begin{array}{rcll}
+\ (n_1,\ n_2) & \xrightarrow{\epsilon} & n & \text{with } n = n_1 + n_2 \\[1em]
\mathit{fst}\ (v_1,\ v_2) & \xrightarrow{\epsilon} & v_1 & \\
\mathit{snd}\ (v_1,\ v_2) & \xrightarrow{\epsilon} & v_2 & \\[1em]
\mathit{opfix}\ (\texttt{fun}\ f \rightarrow e) & \xrightarrow{\epsilon} & e[f \leftarrow \mathit{opfix}\ (\texttt{fun}\ f \rightarrow e)] & \\[1em]
\mathit{opif}\ (\mathit{true},\ ((\texttt{fun}\ \_ \rightarrow e_1),\ (\texttt{fun}\ \_ \rightarrow e_2))) & \xrightarrow{\epsilon} & e_1 & \\
\mathit{opif}\ (\mathit{false},\ ((\texttt{fun}\ \_ \rightarrow e_1),\ (\texttt{fun}\ \_ \rightarrow e_2))) & \xrightarrow{\epsilon} & e_2 &
\end{array}
$$

Small-steps operational semantics for mini-ML

To reduce in depth, we introduce the inference rule

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \rightarrow E(e_2)}$$

where E is a context, defined by the following grammar

$$\begin{array}{rcl} E & ::= & \square \\ & \mid & E\ e \\ & \mid & v\ E \\ & \mid & \texttt{let}\ x = E\ \texttt{in}\ e \\ & \mid & (E,\ e) \\ & \mid & (v,\ E) \end{array}$$

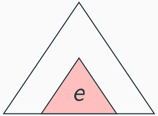
Context

A context is a "term with hole" which is represented by the symbol $\square$

Example

$$E \stackrel{\text{def}}{=} \texttt{let}\ x = +(2, \square)\ \texttt{in}\ \texttt{let}\ y = +(x, x)\ \texttt{in}\ y$$

$E(e)$ denotes the context E in which $\square$ has been replaced by e



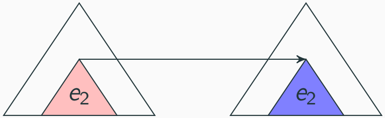
Example

$$E(+(10, 9)) = \texttt{let}\ x = +(2, +(10, 9))\ \texttt{in}\ \texttt{let}\ y = +(x, x)\ \texttt{in}\ y$$

Reduction in a context

The simple rule

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \to E(e_2)}$$



therefore allows us to evaluate a subexpression

Example

We have the reduction

$$\frac{+(1,2) \xrightarrow{\epsilon} 3}{\texttt{let } x = +(1,2) \texttt{ in } +(x,x) \to \texttt{let } x = 3 \texttt{ in } +(x,x)}$$

thanks to the context $E \stackrel{\text{def}}{=} \texttt{let } x = \square \texttt{ in } +(x,x)$.

Evaluation order

We could have made another choice, such as evaluating "from right to left", or even not setting an order of evaluation

In the case of mini-ML, this would not make a difference to the value of an expression, but in a more complex language with side effects or exceptional behaviors, evaluating in one order rather than another might make a difference.

Definition

Defn. (evaluation and normal form)
We denote by $\xrightarrow{*}$ the reflective and transitive closure of the relation $\rightarrow$.
(i.e. $e_1 \xrightarrow{*} e_2$ iff e_1 reduces to e_2 by zero or more steps)

We call an expression e normal form if there is no expression e_0 such that $e \xrightarrow{*} e_0$. □

Note: obviously, values are normal forms;
normal forms that are not values are bad expressions (e.g. `1 2`)

Programming the small-steps semantics

We are going to write the following functions:

```
val head_reduction: expression -> expression
```

corresponds to $\xrightarrow{\epsilon}$

```
val decompose: expression -> context * expression
```

decomposes an expression of the form $E(e)$
with e is head reducible

```
val reduce1: expression -> expression option
```

corresponds to $\rightarrow$

```
val reduce: expression -> expression
```

corresponds to $\xrightarrow{*}$

Programming the small-steps semantics

We start by characterize the values

```
let rec is_a_value = function
  | Const _ | Op _ | Fun _ ->
        true
  | Var _ | App _ | Let _ ->
        false
  | Pair (e1, e2) ->
        is_a_value e1 && is_a_value e2
```

Programming the small-steps semantics

Then we write the head reduction

```
let head_reduction = function
  | App (Fun (x, e1), e2) when is_a_value e2 ->
        subst e1 x e2
  | Let (x, e1, e2) when is_a_value e1 ->
        subst e2 x e1
  | App (Op "+", Pair (Const n1, Const n2)) ->
        Const (n1 + n2)
  | App (Op "fst", Pair (e1, e2))
      when is_a_value e1 && is_a_value e2 ->
        e1
  | App (Op "snd", Pair (e1, e2))
      when is_a_value e1 && is_a_value e2 ->
        e2
  | _ ->
        raise NoReduction
```

Programming the small-steps semantics

A context E can be represented directly by the function $e \mapsto E(e)$

```
type context = expression -> expression
```

```
let hole = fun e -> e
let app_left ctx e2 = fun e -> App (ctx e, e2)
let app_right v1 ctx = fun e -> App (v1, ctx e)
let pair_left ctx e2 = fun e -> Pair (ctx e, e2)
let pair_right v1 ctx = fun e -> Pair (v1, ctx e)
let let_left x ctx e2 = fun e -> Let (x, ctx e, e2)
```

Programming the small-steps semantics

```
let rec decompose e = match e with
(* what we can not decompose *)
  | Var _ | Const _ | Op _ | Fun _ -> raise NoReduction
(* case of one head reduction *)
  | App (Fun (x, e1), e2) when is_a_value e2 -> (hole, e)
  | Let (x, e1, e2) when is_a_value e1 -> (hole, e)
  | App (Op "+", Pair (Const n1, Const n2)) -> (hole, e)
  | App (Op ("fst" | "snd"), Pair (e1, e2))
    when is_a_value e1 && is_a_value e2 ->
      (hole, e)
```

Programming the small-steps semantics

```
(* case of in depth reduction *)
  | App (e1, e2) ->
    if is_a_value e1 then
      let (ctx, rd) = decompose e2 in
      (app_right e1 ctx, rd)
    else
      let (ctx, rd) = decompose e1 in
      (app_left ctx e2, rd)
  | Let (x, e1, e2) ->
    let (ctx, rd) = decompose e1 in
      (let_left x ctx e2, rd)
  | Pair (e1, e2) -> ...
```

Programming the small-steps semantics

```
let reduce1 e =
  try
    let ctx, e' = decompose e in
    Some (ctx (head_reduction e'))
  with NoReduction ->
    None
```

Finally

```
let rec reduce e =
  match reduce1 e with None -> e | Some e' -> reduce e'
```

Efficiency

Such an interpreter is not very efficient

It spent too much time to recalculate the context and then “forget” it

We can do better, for instance, by using a *zipper* [HUE97]

Equivalence between two operational semantics

We will show that the two operational semantics, big-steps and small-steps, are equivalent for expressions whose evaluation ends on a value. i.e.

$$e \twoheadrightarrow v \quad \text{if and only if} \quad e \xrightarrow{*} v$$

Equivalence between two operational semantics

Lem. (Transitions to the context reductions)
Suppose that $e \to e'$. Then for any expression e_2 and any value v, we have

- $e\ e_2 \to e'\ e_2$
- $v\ e \to v\ e'$
- $\texttt{let}\ x = e\ \texttt{in}\ e_2 \to \texttt{let}\ x = e'\ \texttt{in}\ e_2$

□

Proof.
From $e \to e'$ we know that there is an expression E such that

$$e = E(r), \quad e' = E(r'), \quad r \xrightarrow{\epsilon} r'$$

Consider the context $E \stackrel{\text{def}}{=} E\ e_2$, then

$$\frac{r \xrightarrow{\epsilon} r'}{E_1(r) \to E_1(r')} \qquad i.e. \qquad \frac{r \xrightarrow{\epsilon} r'}{e\ e_2 \to e'\ e_2}$$

The other cases can be proceeded similarly. ■

Equivalence between two operational semantics

Prop. (big-steps implies small-steps)
If $e \twoheadrightarrow v$, then $e \xrightarrow{*} v$. □

Proof.
We proceed by induction on the derivation of $e \twoheadrightarrow v$. Suppose that the last rule is that of a function application:

$$\frac{e_1 \twoheadrightarrow (\texttt{fun}\ x \to e_3) \qquad e_2 \twoheadrightarrow v_2 \qquad e_3[x \leftarrow v_2] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

By induction hypothesis for each three derivations in the premise and denote by v_1 the value of $\texttt{fun}\ x \to e_3$, we have the three evaluations in small-steps

$$e_1 \to \cdots \to v_1$$
$$e_2 \to \cdots \to v_2$$
$$e_3[x \leftarrow v_2] \to \cdots \to v$$

Equivalence between two operational semantics

By passing to the context, we also have the three evaluations

$$e_1\ e_2 \rightarrow \cdots \rightarrow v_1\ e_2$$

$$v_1\ e_2 \rightarrow \cdots \rightarrow v_1\ v_2$$

$$e_3[x \leftarrow v_2] \rightarrow \cdots \rightarrow v$$

By inserting the reduction

$$(\texttt{fun} \ \rightarrow e_3)\ v_2 \xrightarrow{\epsilon} e_3[x \leftarrow v_2]$$

between the second and third lines, we obtain the complete reduction

$$e_1\ e_2 \rightarrow \cdots \rightarrow v.$$

The other cases are left as exercises. ■

Equivalence between two operational semantics

To show the other implication, that is, "small-steps implies big-steps", we start by establishing two lemmas. The first is obvious.

Lem. (values are already evaluated)
$v \twoheadrightarrow v$ for any value v. □

Equivalence between two operational semantics

Lem. (reduction and evaluation)
If $e \rightarrow e'$ and $e' \twoheadrightarrow v$, then $e \twoheadrightarrow v$. □

Proof.
We start with the head reductions, i.e., $e \xrightarrow{\epsilon} e'$. Suppose for example that $e = (\texttt{fun}\ x \rightarrow e_1)\ v_2$ and $e' = e_1[x \leftarrow v_2]$. We can construct the reduction

$$\frac{\texttt{fun}\ x \rightarrow e_1) \twoheadrightarrow (\texttt{fun}\ x \rightarrow e_1) \qquad v_2 \twoheadrightarrow v_2 \qquad e_1[x \leftarrow v_2] \twoheadrightarrow v}{(\texttt{fun}\ x \rightarrow e_1)\ v_2 \twoheadrightarrow v}$$

using the previous lemma ($v_2 \twoheadrightarrow v_2$) and the hypothesis $e' \twoheadrightarrow v$. Other cases of head reduction are treated in a similar manner.

Let us now show that if $e \xrightarrow{\epsilon} e'$ and $E(e') \twoheadrightarrow v$, then $E(e) \twoheadrightarrow v$ for any expression E by the structural induction on E (or if one prefers, by induction on the height of the context). We have just done the base case $E = \square$.

Equivalence between two operational semantics

Let us now show that if $e \xrightarrow{\epsilon} e'$ and $E(e') \twoheadrightarrow v$, then $E(e) \twoheadrightarrow v$ for any expression E by the structural induction on E (or if one prefers, by induction on the height of the context). We have just done the base case $E = \square$.

Consider for instance the case $E = E'\ e_2$. We have $E(e') \twoheadrightarrow v$, i.e., $E'(e')\ e_2 \twoheadrightarrow v$. In the case of abstraction, this reduction is of the form

$$\frac{E'(e') \twoheadrightarrow (\texttt{fun}\ \ \to e_3) \qquad e_2 \twoheadrightarrow v_2 \qquad e_3[x \leftarrow v_2] \twoheadrightarrow v}{E'(e')\ e_2 \twoheadrightarrow v}$$

By induction hypothesis, we have $E'(e) \twoheadrightarrow (\texttt{fun}\ x \to e_3)$ and hence

$$\frac{E'(e) \twoheadrightarrow (\texttt{fun}\ x \to e_3) \qquad e_2 \twoheadrightarrow v_2 \qquad e_3[x \leftarrow v_2] \twoheadrightarrow v}{E'(e)\ e_2 \twoheadrightarrow v}$$

that is, $E(e) \twoheadrightarrow v$. The other cases are left as exercises. ■

Equivalence between two operational semantics

Prop. (small-steps implies big-steps)
If $e \xrightarrow{*} v$ then $e \twoheadrightarrow v$. □

Proof.
Let us write the reduction in small steps in the form

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n \rightarrow v$$

We have $v \twoheadrightarrow v$ and hence by previous lemma $e_n \twoheadrightarrow v$.
Similarly, $e_{n-1} \twoheadrightarrow v$ and so on until $e \twoheadrightarrow v$. ■

What about imperative languages?

We can define an operational semantics, in big-steps or in small-steps, for a language with imperative features.

$$
\begin{array}{lllll}
e & ::= & & & \textbf{expression} \\
 & & \mid & c & \text{constant } (\texttt{true}, 42, \ldots) \\
 & & \mid & x & \text{global variable} \\
 & & \mid & e\ op\ e & \text{binary operation } (+, <, \ldots) \\
 & & & & \\
c & ::= & & & \textbf{constant} \\
 & & \mid & n & \text{integer constant}(-17, 42, \ldots) \\
 & & \mid & b & \text{boolean constant}(\texttt{true}, \texttt{false})
\end{array}
$$

What about imperative languages?

s	$::=$		**statement**
	$\mid$	$x \leftarrow e$	assignment
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s$	conditional
	$\mid$	$\texttt{while}\ e\ \texttt{do}\ s$	loop
	$\mid$	$s;\ s$	sequence
	$\mid$	$\texttt{skip}$	do nothing

What about imperative languages?

Typically we associate an environment/state S to evaluate/reduce an expression:

$$S,\ e \twoheadrightarrow S',\ v \quad \text{or} \quad S,\ e \rightarrow S',\ e'$$

and, for instance, define inference rule like

$$\frac{S, e \twoheadrightarrow S', v}{S, x := e \twoheadrightarrow S' \oplus \{x \mapsto v\},\ \texttt{void}}$$

where S can be decomposed into many elements, for modeling a stack, a heap and others.

Correctness of a compiler

Correctness

Formal semantics is a powerful tool, which can notably be used to show the correctness of a compiler.

By this we mean that if the source language is provided with semantics $\rightarrow_s$ and the machine language of semantics $\rightarrow_m$, and if the expression e is compiled in $C(e)$ then we must have a "commuting diagram" of the form

$$\begin{tikzcd} e \arrow[r, "*"', phantom] & \overset{*}{\longrightarrow}_s & v \\ \downarrow & & \approx \\ C(e) & \overset{*}{\longrightarrow}_m & v' \end{tikzcd}$$

where $v \approx v'$ expresses that the values v and v' are coincident.

Minimalist example

Let us consider arithmetic expressions with no variables

$$e \quad ::= \quad n \mid e + e$$

and let us show the correctness of a very simple compiler to x86-64 that uses the stack to store intermediate computations

input language

We set a small-steps semantics for the input language

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

Target language

Similarly, we set a small-step semantics for the target language

$$\begin{array}{rcl} m & ::= & \texttt{movq} \quad \$n, r \\ & | & \texttt{addq} \quad \$n, r \mid \texttt{addq} \quad r, r \\ & | & \texttt{movq} \quad (r), r \mid \texttt{movq} \quad r, (r) \\ r & ::= & \texttt{\%rdi} \mid \texttt{\%rsi} \mid \texttt{\%rsp} \end{array}$$

A state gathers the value of register R,
and the contents of the memory, M

$$\begin{array}{rcl} R & ::= & \{\texttt{\%rdi} \mapsto n; \texttt{\%rsi} \mapsto n; \texttt{\%rsp} \mapsto n\} \\ M & ::= & \mathbb{N} \mapsto \mathbb{Z} \end{array}$$

We then define the semantics of an instruction m using a relation

$$R, M, m \rightarrow_m R', M'$$

Target language

the relation $R, M, m \rightarrow_m R', M'$ is defined as follows

$$
\begin{array}{rcll}
R, M, \texttt{movq} & \$n, r & \rightarrow_m & R\, r \mapsto n, M \\
R, M, \texttt{addq} & \$n, r & \rightarrow_m & R\, r \mapsto R(r) + n, M \\
R, M, \texttt{addq} & r_1, r_2 & \rightarrow_m & R\, r_2 \mapsto R(r_1) + R(r_2), M \\
R, M, \texttt{movq} & (r_1), r_2 & \rightarrow_m & R\, r_2 \mapsto M(R(r_1)), M \\
R, M, \texttt{movq} & r_1, (r_2) & \rightarrow_m & R, M\, R(r_2) \mapsto R(r_1)
\end{array}
$$

Compiler

The final value of an expression is stored in `%rdi`

$$
\begin{array}{rcll}
code(n) & = & \texttt{movq} & \$n, \%\texttt{rdi} \\
\\
code(e_1 + e_2) & = & code(e_1) & \\
 & & \texttt{addq} & \$-8, \%\texttt{rsp} \\
 & & \texttt{movq} & \%\texttt{rdi}, (\%\texttt{rsp}) \\
 & & code(e_2) & \\
 & & \texttt{movq} & (\%\texttt{rsp}), \%\texttt{rsi} \\
 & & \texttt{addq} & \$8, \%\texttt{rsp} \\
 & & \texttt{addq} & \%\texttt{rsi}, \%\texttt{rdi}
\end{array}
$$

Correctness of the compiler

We seek to prove that if

$$e \xrightarrow{*} n$$

and if

$$R, M, code(e) \xrightarrow{*}_m R', M'$$

then $R'(\%\texttt{rdi}) = n$.

We proceed by structural induction on e

Correctness of the compiler

We will show a stronger property (an invariant), namely

if $e \xrightarrow{*} n$ and $R, M, code(e) \xrightarrow{*}_m R', M'$, then

$$\begin{cases} R'(\%\mathtt{rdi}) = n \\ R'(\%\mathtt{rsp}) = R(\%\mathtt{rsp}) \\ \forall a \geq R(\%\mathtt{rsp}), M'(a) = M(a) \end{cases}$$

Correctness of the compiler

- case $e = n$:
  We have $e \xrightarrow{*} n$ and $code(e) = \texttt{movq}\quad \$n, \%\texttt{rdi}$ and the result is immediate

- case $e = e_1 + e_2$:
  We have $e \xrightarrow{*} n_1 + e_2 \xrightarrow{*} n_1 + n_2$ with $e_1 \xrightarrow{*} n_1$ and $e_2 \xrightarrow{*} n_2$
  thus we can invoke the induction hypothesis on e_1 and e_2

Correctness of the compiler

	R, M	
$code(e_1)$	R_1, M_1	by induction hypothesis $R_1(\texttt{\%rdi}) = n_1$ and $R_1(\texttt{\%rsp}) = R(\texttt{\%rsp})$ $\forall a \geq R(\texttt{\%rsp}), M_1(a) = M(a)$
`addq $-8, %rsp` `movq %rdi, (%rsp)`	R'_1, M'_1	$R'_1 = R_1\{\texttt{\%rsp} \mapsto R(\texttt{\%rsp}) - 8\}$ $M'_1 = M_1\{R(\texttt{\%rsp}) - 8 \mapsto n_1\}$
$code(e_2)$	R_2, M_2	by induction hypothesis $R_2(\texttt{\%rdi}) = n_2$ and $R_2(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8$ $\forall a \geq R(\texttt{\%rsp}) - 8, M_2(a) = M'_1(a)$

`movq (%rsp), %rsi`
`addq $8, %rsp`
`addq %rsi, %rdi`
	R', M_2	$R'(\texttt{\%rdi}) = n_1 + n_2$
		$R'(\texttt{\%rsp}) = R(\texttt{\%rsp}) - 8 + 8 = R(\texttt{\%rsp})$
		$\forall a \geq R(\texttt{\%rsp}), M_2(a) = M'_1(a) = M_1(a) = M(a)$

in the large

Such a proof can be done for a realistic compiler

Example: CompCert, an optimizing compiler from C to PowerPC, ARM, RISC-V, and x86, has been formally verified using the Coq proof assistant [Ler06]

see `https://compcert.org/`

Exercise

- handwritten assignment HW2
- project assignment: a mini-Python interpreter
  - in Java or OCaml
  - take your time to read and understand the code that is provide

```
> ./mini-python tests/good/pascal.py
*
**
***
****
*****
******
*******
*000000*
**00000**
***0000***
****000****
*****00*****
******0******
**************
*000000*000000*
**00000**00000**
***0000***0000***
****000****000****
*****00*****00*****
******0******0******
*********************
```

References i

GÉRARD HUET.
The zipper.
Journal of Functional Programming, 7(5):549–554, 1997.

Xavier Leroy.
Formal certification of a compiler back-end, or: programming a compiler with a proof assistant.
In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

Benjamin C. Pierce.
Types and Programming Languages.
The MIT Press, 1st edition, 2002.

Questions?