Compilation and Program Analysis (#9a) : Functions: semantics

Ludovic Henrio

Master 1, ENS de Lyon et Dpt Info, Lyon1

2022-2023

Approach and objective of the course

- Give a semantics to functions in an imperative setting.
  inspired by what can be found in the literature
- Explore the different ways to deal with variables, different environments of execution, ...
- Big step and small step operational semantics
- Observe that things are more complex but also more constructive than with a functional calculus like λ-calculus.

Note on organisation:

1: Course + **live proofs** ;

2: **exercises and proofs done after the course**

Mini-While Syntax 1/2

Expressions:

$$e \quad ::= \quad c \,|\, e + e \,|\, e \times e \,|\, ...$$

Mini-while:

$S(Smt)$	$::=$	$x := expr$	assign
	$\mid$	$x := \boldsymbol{f(e_1, .., e_n)}$	simple function call
	$\mid$	$skip$	do nothing
	$\mid$	$S_1; S_2$	sequence
	$\mid$	`if` b `then` S_1 `else` S_2	test
	$\mid$	`while` b `do` S `done`	loop

Mini-While Syntax 2/2

[NEW] Programs with function definitions and global variables

$$
\begin{array}{rcll}
Prog & ::= & D \;\; FunDef \;\; Body & \text{Program} \\
Body & ::= & D; S & \text{Function/main body} \\
D & ::= & var\ x : \tau | D; D & \text{Variable declaration} \\
FunDef & ::= & \tau\ f(x_1 : \tau_1, .., x_n : \tau_n)\ Body; return\ e & \\
 & | & FunDef \;\; FunDef & \text{Function def}
\end{array}
$$

Note/discussion: to simplify syntax and semantics:

1) function call is not an expression but a special statement.

2) `return` only appears at the end of the function definition (enforced by syntax).

Dealing with variable declaration and store management

Variable declaration: $Vars(D)$ is the set of variables declared by D.

Reminder: we could use it to initialize the local memory when needed and ensure progress (see typing course).

We define a global store update:

$$\sigma'[X \mapsto \sigma](x) = \begin{cases} \sigma(x) & \text{if } x \in X \\ \sigma'(x) & \text{else} \end{cases}$$

This will be used to restore part of the store to a previous value.

Function table

For each function declared with name f we have $params(f)$ the list of parameter variables, $ret(f)$ the expression in the return statement, and $body(f)$ the function body.

This could be formally defined as a function table Φ that is given as parameter of the semantics, i.e.: change the signature into $\Phi \vdash (S, \sigma) \longrightarrow \sigma'$ and use Φ to obtain $ret(f)$, $params(f)$, and $body(f)$.

Here we suppose that the functions ret, $params$ and $body$ are globally known.

Big step semantics (old) 1/2

$\longrightarrow : Stm \to (State \to State)$

$$(x := e, \sigma) \longrightarrow \sigma[x \mapsto Val(e, \sigma)]$$

$$(\texttt{skip}, \sigma) \longrightarrow \sigma$$

$$\frac{(S_1, \sigma) \longrightarrow \sigma' \qquad (S_2, \sigma') \longrightarrow \sigma''}{\big((S_1; S_2), \sigma\big) \longrightarrow \sigma''}$$

Big step semantics (old) 2/2

$$\frac{Val(b,\sigma) = tt \qquad (S_1,\sigma) \longrightarrow \sigma'}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\sigma) \longrightarrow \sigma'}$$

$$\frac{Val(b,\sigma) = ff \qquad (S_2,\sigma) \longrightarrow \sigma'}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2,\sigma) \longrightarrow \sigma'}$$

$$\frac{Val(b,\sigma) = tt \qquad (S,\sigma) \longrightarrow \sigma' \qquad (\texttt{while } b \texttt{ do } S \texttt{ done},\sigma') \longrightarrow \sigma''}{(\texttt{while } b \texttt{ do } S \texttt{ done},\sigma) \longrightarrow \sigma''}$$

$$\frac{Val(b,\sigma) = ff}{(\texttt{while } b \texttt{ do } S \texttt{ done},\sigma) \longrightarrow \sigma}$$

Evaluation of program: Let $D \;\; FunDef \;\; D'; S$ be a program, its evaluation is σ' s.t. $(S, \emptyset) \longrightarrow \sigma'$

Additional simplification

We only give semantics to functions that have a simple parameter for the formalisation of the semantics.
For example we use $f(e)$ instead of $f(e_1, .., e_n)$. Additionally $params(f)$ is an array of a single element $[x]$
Extending to multiple parameters raises no particular issue but you should be careful with the indices ...
In the examples we may use several parameters ...

Big step semantics (NEW) – First solution

Heavy manipulation of stores:

$$\frac{\begin{array}{c} body(f) = D_f; S_f \\ bind_1(f, e, \sigma) = (S, \sigma') \qquad (S, \sigma') \longrightarrow \sigma'' \qquad v = Val(ret(f), \sigma'') \end{array}}{(x := f(e), \sigma) \longrightarrow \sigma''[(Vars(D_f) \cup params(f)) \mapsto \sigma, x \mapsto v]}$$

Where:

$$bind_1(f, e, \sigma) = (S_f, \sigma[x' \mapsto v'])$$

with $body(f) = D_f; S_f \quad params(f) = [x'] \quad Val(e, \sigma) = v'$

Initial configuration:

$$(S_m, \emptyset) \text{ for program } D \;\; FunDef \;\; D'; S$$

An example

Evaluate the following program:

```
int x
int f(int x) {
  int y;
  x:=1;
  y:=2;
  return x+1;
}
{
  int y;
  x:=0;
  y:=0;
  y:=f(3);
}
```

Evaluation of this first solution

Notes:

- call-by-value,
- parameters in store,
- Non-trivial store restoration.

Variables: What happens with variables that have the same name? local vs. local? global vs. local? recursive invocations? **discussion live**

Problem: $\sigma'[(Vars(D_f) \cup params(f)) \mapsto \sigma]$ is used to restore the store as it was before the invocation.

This is really impractical: difficult to compute, difficult to reason about, far from any implementation.

But not much changes to the other rules, the initial configuration is unchanged.

Big step semantics (NEW) – Second solution

Renaming and fresh variables:

$$\frac{bind_2(f, e', \sigma) = (S', \sigma', e) \qquad (S', \sigma') \longrightarrow \sigma'' \qquad v = Val(e, \sigma'')}{(x := f(e'), \sigma) \longrightarrow \sigma''[x \mapsto v]}$$

$bind_2(f, e', \sigma) = (S'_f, \sigma[z \mapsto v'], e)$

where

$body(f) = D_f; S_f \qquad params(f) = [x'] \qquad Vars(D_f) = \{y_1..y_k\} \qquad z$ fresh

$\forall i \in [1..k].\, t_i$ fresh $\qquad Val(e', \sigma) = v' \qquad S'_f = S_f[z/x'][t_1/y_1]..[t_k/y_k]$

$$e = ret(f)[z/x'][t_1/y_1]..[t_k/y_k]$$

and fresh means not in σ (and not among the other fresh variables)

An example

Evaluate the following program:

```
int x
int f(int x) {
  int y;
  x:=1;
  y:=2;
  return x+1;
}
{
  int y;
  x:=0;
  y:=0;
  y:=f(3);
}
```

Evaluation of this Second solution

Note: We can see in the store the values of variables that are hidden by the current context but we cannot access them because of renaming.
There are also variables for function frames that are finished but are still visible, these variables cannot be accessed any more.

Variables: The store grows (unbounded) but we could easily remove useless variables, (detecting useless variables is not trivial but possible).

Problem: variable substitution difficult and inefficient (more difficult if recursive blocks). But no complex store manipulation.

Big step semantics (NEW) – Third solution (1/2)

A store and a stack: $\longrightarrow$: $Stm, Stack, Store \rightarrow Stack, Store$

Where $Stack : Var \rightarrow address$ and $Store : address \rightarrow Val$

$$\frac{bind_3(f, e', \Sigma, sto) = (S', \Sigma', sto') \qquad (S', \Sigma', sto') \longrightarrow (\Sigma'', sto'') \qquad v = Val(ret(f), sto'' \circ \Sigma'')}{(x := f(e'), \Sigma, sto) \longrightarrow (\Sigma, sto''[\Sigma(x) \mapsto v])}$$

$bind_3(f, e', \Sigma, sto) = (S_f, \Sigma', sto[\ell \mapsto v'])$ where

$$body(f) = D_f; S_f \qquad params(f) = [x'] \qquad Vars(D_f) = \{y_1..y_k\} \qquad \ell \text{ fresh}$$

$$\ell'_1..\ell'_k \text{ fresh} \quad Val(e', sto \circ \Sigma) = v' \qquad \Sigma' = \Sigma[x' \mapsto \ell][y_1..y_k \mapsto \ell'_1..\ell'_k]$$

where fresh means location not in sto (and not among the other fresh locations picked)

!! Access to store must be changed everywhere. See next slide!!

Big step semantics (NEW) – Third solution (2/2)

We now have to deal with two levels for memory addressing, this modifies the whole semantics: $Val(e_i, \Sigma)$ replaced by $Val(e_i, sto \circ \Sigma)$ everywhere.
And we have a new assign rule: $(x := e, \Sigma, sto) \longrightarrow \Sigma, sto[\Sigma(x) \mapsto Val(e, sto \circ \Sigma)]$

- the stack is "unstacked" after method invocation.
- in $\Sigma' = \mathbf{\Sigma}[x \mapsto \ell][y_1...]$, $\mathbf{\Sigma}$ is too big: only global variables are needed.
- Works with recursively defined blocks with local variables (**how?**)
- Memory grows unbounded but Σ is bounded; we could remove useless locations (gc or manually),
- Have to deal with two levels for memory addressing but this is closer to reality (e.g. statements are not modified upon function call).
- **What is a good initial configuration?**

An example

Evaluate the following program:

```
int x
int f(int x) {
  int y;
  x:=1;
  y:=2;
  return x+1;
}
{
  int y;
  x:=0;
  y:=0;
  y:=f(3);
}
```

Structural Op. Semantics (SOS = small step) for mini-while (OLD)

$$(x := a, \sigma) \Rightarrow \sigma[x \mapsto Val(a, \sigma)]$$

$$(\texttt{skip}, \sigma) \Rightarrow \sigma$$

$$\frac{(S_1, \sigma) \Rightarrow \sigma'}{\big((S_1; S_2), \sigma\big) \Rightarrow (S_2, \sigma')} \qquad \frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{\big((S_1; S_2), \sigma\big) \Rightarrow (S_1'; S_2, \sigma')}$$

$$\frac{Val(b, \sigma) = tt}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \Rightarrow (S_1, \sigma)}$$

$$\frac{Val(b, \sigma) = \mathit{ff}}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \Rightarrow (S_2, \sigma)}$$

the challenge with small step semantics

Writing a SOS is often desirable but with our functions it raises several challenges.
Indeed call and return cannot be done in the same rule, and thus:

- Previous state of the memory (before function call is not accessible in the rule for returning from the function),
- The point where the caller was before the call is also lost.

General solution: add thee information and define an extended syntax for "configurations" i.e. state of the program execution at runtime.
In the real life where we do not have

OLD SOS with new store structure – Principle

New configuration: $(Stm, ??, Stack, Store)$. $??$ defined later.

$$(x := e, ??, \Sigma, \boldsymbol{sto}) \Rightarrow (\Sigma, \boldsymbol{sto}[\boldsymbol{\Sigma}(x) \mapsto \boldsymbol{Val}(e, \boldsymbol{sto} \circ \boldsymbol{\Sigma})])$$

$$(\texttt{skip}, ??, \Sigma, \boldsymbol{sto}) \Rightarrow (\Sigma, \boldsymbol{sto})$$

$$\frac{(S_1, ??, \Sigma, \boldsymbol{sto}) \Rightarrow (\Sigma', \boldsymbol{sto'})}{\big((S_1; S_2), ??, \Sigma, \boldsymbol{sto}\big) \Rightarrow (S_2, ??, \Sigma', \boldsymbol{sto'})}$$

$$\frac{(S_1, ??, \Sigma, \boldsymbol{sto}) \Rightarrow (S_1', ??, \Sigma', \boldsymbol{sto'})}{\big((S_1; S_2), ??, \Sigma, \boldsymbol{sto}\big) \Rightarrow ((S_1'; S_2), ??, \Sigma', \boldsymbol{sto'})}$$

$$\frac{Val(b, \boldsymbol{sto} \circ \boldsymbol{\Sigma}) = tt}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, ??, \Sigma, \boldsymbol{sto}) \Rightarrow (S_1, ??, \Sigma, \boldsymbol{sto})}$$

$$\frac{Val(b, \boldsymbol{sto} \circ \boldsymbol{\Sigma}) = ff}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, ??, \Sigma, \boldsymbol{sto}) \Rightarrow (S_2, ??, \Sigma, \boldsymbol{sto})}$$

Small step semantics, based on third solution

$??$ is used to remember the execution contexts: it is a list of $(Stack, Stm)$. Let:: be the list constructor. Ctx is a list of $(Stack, Stm)$. In the big step semantics this is not needed because the inference is more complex (and remembers contexts)

CALL

$$\frac{bind_3(f, e', \Sigma, sto) = (S', \Sigma', sto')}{(x := f(e'); S, Ctx, \Sigma, sto) \Rightarrow (S', (\Sigma, x := R(f); S) :: Ctx, \Sigma', sto')}$$

$x := f(e'); S$ must be the whole current statement (imposed by the rule for sequence).
$R(f)$ is just a marker that remembers the name of the function called (and the calling point).
$bind_3$ is already defined.

SOS with new store structure and contexts

$$(x := e, Ctx, \Sigma, \boldsymbol{sto}) \Rightarrow (Ctx, \Sigma, \boldsymbol{sto}[\boldsymbol{\Sigma(x) \mapsto Val(e, sto \circ \Sigma)}])$$

$$\frac{(S_1, Ctx, \Sigma, \boldsymbol{sto}) \Rightarrow (Ctx, \Sigma', \boldsymbol{sto'})}{\big((S_1; S_2), Ctx, \Sigma, \boldsymbol{sto}\big) \Rightarrow (S_2, Ctx, \Sigma', \boldsymbol{sto'})}$$

$$\frac{(S_1, Ctx, \Sigma, \boldsymbol{sto}) \Rightarrow (S_1', Ctx, \Sigma', \boldsymbol{sto'})}{\big((S_1; S_2), Ctx, \Sigma, \boldsymbol{sto}\big) \Rightarrow (S_1'; S_2, Ctx, \Sigma', \boldsymbol{sto'})}$$

And a new **rule** for return (when current computation finished)

$$\frac{v = Val(ret(f), sto \circ \Sigma')}{((\Sigma, x := R(f); S) :: Ctx, \Sigma', sto) \Rightarrow (S, Ctx, \Sigma, sto[\Sigma(x) \mapsto v])}$$

if, skip, and while rules are trivially adapted

Initial configuration unchanged

What do we have at the end of the execution?

An example

Evaluate the following program:

```
int x
int f(int x) {
  int y;
  x:=1;
  y:=2;
  return x+1;
}
{
  int y;
  x:=0;
  y:=0;
  y:=f(3);
}
```

[REMINDER] Safety = well typed programs do not go wrong

In case of a small-step semantics safety relies on two lemmas:

Well-type programs run without error

Lemma (progression for mini-while)

If $\Gamma \vdash (S, \sigma)$*, then there exists* S', σ' *such that* $(S, \sigma) \Rightarrow (S', \sigma')$
OR there exists σ' *such that* $(S, \sigma) \Rightarrow \sigma'$*.*

... and remain well-typed

Lemma (preservation)

If $\Gamma \vdash (S, \sigma)$ *and* $(S, \sigma) \Rightarrow (S', \sigma')$ *then* $\Gamma \vdash (S', \sigma')$*.*

Note: (S, σ) cannot be a final configuration. Γ never changes (defined by declarations)

Recall the property for expression evaluation: if σ and Γ agree on all variables the valuation of the expression is of the right type.

[Reminder] Typing rules

Typing of statements has the form : $\Gamma, \Gamma_f \vdash S$ Where Γ: map that defines the variable types, Γ_f: function map, S statement.

$$\frac{\begin{array}{c} D \rightarrow_d \Gamma_g \qquad \mathit{Fundef} \rightarrow_f \Gamma_f \qquad D_m \rightarrow_d \Gamma_m \\ \Gamma_g + \Gamma_m, \Gamma_f \vdash S \qquad \forall(\tau\ f(x_1 : \tau_1, .., x_n : \tau_n)\ D_f; S_f; return\ e \in \mathit{Fundef}). \\ \Gamma_g + \Gamma_l \vdash e : \tau \wedge \Gamma_g + \Gamma_l, \Gamma_f \vdash S_f \text{ with } x_1 : \tau_1; ..; x_n : \tau_n; D_f \rightarrow_d \Gamma_l \end{array}}{\vdash D\ \mathit{Fundef}\ D_m; S}$$

$\Gamma_g + \Gamma_l$ overrides Γ_g with Γ_l, i.e. $(\Gamma_g + \Gamma_l)(x)$ is $\Gamma_l(x)$ if it is defined and $\Gamma_g(x)$ else.

CALL

$$\frac{\Gamma_f(f) = \tau_1..\tau_n \rightarrow \tau \qquad \forall i \in [1..n]. \Gamma \vdash e_i : \tau_i \qquad \Gamma \vdash x : \tau}{\Gamma, \Gamma_f \vdash x := f(e_1, .., e_n)}$$

State type safety and prove it for functions

Slight change in the correctness wrt store, it cannot be:

$$\Gamma \vdash (S, \sigma) \iff (\Gamma \vdash S \wedge \forall x. \emptyset \vdash \sigma(x) : \tau \iff \Gamma(x) = \tau)$$

any more!

Attempt 1:

$$\begin{aligned}\Gamma, \Gamma_f \vdash (S, Ctx, \Sigma, \sigma) \iff \big(&\Gamma, \Gamma_f \vdash S \wedge \forall x. (\emptyset \vdash \sigma(\Sigma(x)) : \tau \iff \Gamma(x) = \tau) \\ \wedge\, &\forall (\Sigma', y := R(f); S') \in Ctx. \Gamma, \Gamma_f \vdash S' \wedge \forall x. (\emptyset \vdash \sigma(\Sigma'(x)) : \tau \iff \mathbf{\Gamma}(x) = \tau) \\ \wedge\, &\exists \tau'. \Gamma_f(f) = \tau_1(..\tau_n) \to \tau' \wedge \mathbf{\Gamma}(y) = \tau' \\ \wedge\, &\text{ all function bodies are well-typed (cf rule)}\big)\end{aligned}$$

What is wrong?

Preservation and progress

On board: adaptation of the theorem + proof "sketch" for one or 2 cases

Recall we have to deal with variable initialisation.

First, we need also a typing rule for $x := R(f)$!

[Bonus 1] Dealing with non-terminal return

In big step semantics: One simple solution is to interrupt big step and get back to function call

$$\frac{v = Val(e, \Sigma)}{(return\ e, \Sigma) \longrightarrow (return\ v, \Sigma)}$$

Need to deal with sequence:

$$\frac{(S_1, \Sigma) \longrightarrow (return\ v, \Sigma')}{\big((S_1; S_2), \Sigma\big) \longrightarrow (return\ v, \Sigma')}$$

In small step, we can instead trigger the new rule for return at this point.

[Bonus 2] One semantics from the literature: imperative objects

From: Gordon A.D., Hankin P.D., Lassen S.B. (1997) Compilation and equivalence of imperative objects. FSTTCS 1997.

(Red Object) $(\mathcal{R}[o], \sigma) \to (\mathcal{R}[\iota], \sigma')$ if $\sigma' = (\iota \mapsto o) :: \sigma$ and $\iota \notin dom(\sigma)$.

(Red Select) $(\mathcal{R}[\iota.\ell_j], \sigma) \to (\mathcal{R}[b_j\{\!\{\iota/x_j\}\!\}], \sigma)$
if $\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$ and $j \in 1..n$.

(Red Update) $(\mathcal{R}[\iota.\ell_j \Leftarrow \varsigma(x)b], \sigma) \to (\mathcal{R}[\iota], \sigma')$
if $\sigma(\iota) = [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..n}]$, $j \in 1..n$, and
$\sigma' = \sigma + (\iota \mapsto [\ell_i = \varsigma(x_i)b_i \ ^{i\in 1..j-1}, \ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i \ ^{i\in j+1..n}])$.

(Red Clone) $(\mathcal{R}[clone(\iota)], \sigma) \to (\mathcal{R}[\iota'], \sigma')$
if $\sigma(\iota) = o$, $\sigma' = (\iota' \mapsto o) :: \sigma$ and $\iota' \notin dom(\sigma)$.

(Red Let) $(\mathcal{R}[let\ x = v\ in\ b], \sigma) \to (\mathcal{R}[b\{\!\{v/x\}\!\}], \sigma)$.

(Red Appl) $(\mathcal{R}[(\lambda(x)b)(v)], \sigma) \to (\mathcal{R}[b\{\!\{v/x\}\!\}], \sigma)$.

Figure: Small-step operational semantics

What kind of semantics for functions is this?

Conclusion

We had a look at many semantics to specify function behaviour; difficulty came from the imperative aspect + scopes + context switch.

Problems solved:

- stacking environment
- scope of variables
- restoring the right environment after invocation
- return a value

Each of the semantics has its advantages and corresponds to some of the semantics encountered in the literature.

We have not seen how to deal with imbricated scopes in general (it is often a simple extension along the same principles)