Compilation and Program Analysis (#10) : Functions: semantics

Yannick Zakowski[1]

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025

[1] Slides borrowed from Ludovic Henrio

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).

OLD Type System (1/2)

From a declaration, we construct a typing context $\Gamma : Var \to Basetype$ with the two following rules:

$$\frac{}{var\ x : t \to_d [x \mapsto t]}$$

$$\frac{D_1 \to_d \Gamma_1 \quad D_2 \to_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D1; D_2 \to_d \Gamma_1 \cup \Gamma_2}$$

Then a typing judgment for expressions is $\Gamma \vdash e : \tau \in Basetype$. Statements have no type.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

OLD Type System (2/2)

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1; S_2}$$

$$\frac{D \to_d \Gamma \quad \Gamma \vdash S}{\emptyset \vdash D; S}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done}}$$

Function table (types)

First we extract the list of function signatures:

$$\frac{}{\tau f(x_1 : \tau_1, .., x_n : \tau_n)\ Body;\ return\ e \rightarrow_f [f \mapsto (\tau_1, .., \tau_n \rightarrow \tau)]}$$

$$\frac{Fundef_1 \rightarrow_f \Gamma_1 \quad Fundef_2 \rightarrow_f \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{Fundef_1\ Fundef_2 \rightarrow_f \Gamma_1 \cup \Gamma_2}$$

Type judgements and typing program

Typing of statements has now the form : $\Gamma, \Gamma_f \vdash S$
Where Γ: map that defines the variable types, Γ_f: function map, S statement. To type a program we type all function bodies:

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

Note: in $\Gamma_g + \Gamma_l$, bindings in Γ_g are shadowed by ones in Γ_l, i.e. $(\Gamma_g + \Gamma_l)(x)$ is $\Gamma_l(x)$ if $x \in dom(\Gamma_l)$.

Typing function call

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

Typing rules for other statement are unchanged (except for the threading of the additional Γ_f parameter)

An example

```
bool x
int f(int x, bool b) {
  int y;
  x:=1;
  y:=3;
  if b then x:=x+1 else x:=y;
  return x+1;
}
```

```
{
  int y;
  x:=true;
  y:=0;
  y:=f(3, False);
  y:=f(True); // Type error
}
```

Typing functions in Mini-C

- Typing of function calls have the right number of parameters, of the right type.
- There is function definition and function declaration without body: if both are present coherence should be checked (same types for parameter and return).
- Do not forget to check the return type.
- Typing MiniC is quite easy, producing meaningful error message is harder!
  Look at typing rule and suggest a few error messages

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 $\Phi \vdash (S, \sigma) \Downarrow \sigma'$ 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

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

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

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

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

Big step semantics (old) 2/2

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

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

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

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

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

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:

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

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:

$(S_m, \emptyset)$ 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:

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

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

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: $\Downarrow$: $Stm, Stack, Store \to Stack, Store$

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

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

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

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

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

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: $(x := e, \Sigma, sto) \Downarrow \Sigma, sto[\Sigma(x) \mapsto Val(e, sto \circ \Sigma)]$

- 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), they are easy to spot.
- 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 (in particular Σ)?**

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

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

$x := f(e'); S$ is (by definition) the whole current statement (body). $R(f)$ 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 like in the third solution for big step, with an empty Ctx.

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 (simplified with 1 parameter)

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.

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

$\Gamma_g + \Gamma_l$ 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

$$\frac{\Gamma_f(f) = \tau_1 \rightarrow \tau \qquad \Gamma \vdash e : \tau_1 \qquad \Gamma \vdash x : \tau}{\Gamma, \Gamma_f \vdash x := f(e)}$$

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 $x := R(f)$!

Conclusion

While typing for Mini-While with functions is straightforwards, we have seen that there is some margin for design of its operational semantics. Interestingly, a satisfying solution implements at a higher level of abstraction the exact same intuitions we have followed to generate code when compiling functions.
In particular, care had to be taken to:

- properly handling the scope of variables: pushing and popping a stack of environments (See FP);
- properly restoring the continuation when returning (See SP)