SYNTAX AND SEMANTICS OF MiniOOL

P. COUSOT

We consider the syntax and small-step operational semantics of an extremely simple object-oriented language `MiniOOL`.

1. Syntax

$\mathtt{p}, \ldots, \mathtt{x}, \ldots$	$\in$	Var	Variables
$\mathtt{f}, \ldots$	$\in$	Field	Fields ($\mathtt{val} \in \mathsf{Field}$ is reserved)[1]
e	$\in$	Exp	Expressions
e	$::=$	$\mathtt{f}$	Field expression
	$\mid$	$\mathtt{1} \mid e\ \mathtt{-}\ e \mid \ldots$	Arithmetic expression[2]
	$\mid$	$\mathtt{null} \mid \mathtt{x} \mid e\mathtt{.}e$	Location expression[3]
	$\mid$	$\mathtt{proc\ y}\mathtt{:}C$	Recursive procedure expression
b	$\in$	Bool	Boolean expressions[4]
b	$::=$	$\mathtt{true} \mid \mathtt{false} \mid e\ \mathtt{==}\ e \mid e\ \mathtt{<}\ e \mid \ldots$	
C	$\in$	Cmd	Commands
C	$::=$	$\mathtt{var\ x}; C$	Variable declaration
	$\mid$	$e(e)$	Recursive procedure call
	$\mid$	$\mathtt{malloc(x)}$	Dynamic object allocation
	$\mid$	$\mathtt{x}\ \mathtt{=}\ e$	Variable assignment[5]
	$\mid$	$e\mathtt{.}e\ \mathtt{=}\ e$	Field assignment
	$\mid$	$\mathtt{skip} \mid \{C; C\} \mid \mathtt{while}\ b\ C \mid \mathtt{if}\ b\ C\ \mathtt{else}\ C$	Sequential control
	$\mid$	$\{C \mid\mid\mid C\} \mid \mathtt{atom}(C)$	Parallelism

Date: Thursday 13th September, 2018, 17:12.

The language leaves out static types, inheritance, etc. Scoping rules ensuring that identifiers are used only in the lexical scope of their declaration bloc are left implicit in this context-free grammar (and will be specified by the static semantics of Sect. 3).

2. Examples

Example 1 (Static scoping)**.** *The program*

```
var r; var h; h=1; var p; p = proc y:  r = y+h; var h; h=2; p(4);
```

is equivalent to[6]

`var r; var h`$_1$`; h`$_1$`=1; var p; p = proc y:  r = y+h`$_1$`; var h`$_2$`; h`$_2$`=2;`
`p(4);`

and therefore returns `r` *= 5, not 6.*

Example 2 (Recursive procedure)**.** *A program example (omitting parentheses) is*

```
var p; p = proc y:if y < 1 then p = 1 else p(y - 1); p(1)
```

which creates a procedure `p` *taking an integer parameter and recursively terminating by assigning 1 to* `p` *so that the call* `p(1)` *assigns 1 to* `p`*.*

Example 3 (Object creation)**.** *In the program*

```
var x; malloc(x);
x.c = 0;
x.f = proc y:if y < 1 then x.r = x.c else x.f(y - 1);
x.f(2)
```

an object `x` *is created with fields* `r` *implicitly initialized to null,* `c` *initialized to 0, and* `f` *initialized to a procedure taking an integer parameter and recursively terminating by assigning* `c` *to* `r` *so that the call* `x.f(2)` *assigns 0 to* `r`*.*

[1] Fields must be distinguished from $\mathsf{Variable}$s. In the language this can be achieved in different ways. The simplest solution is to have a different syntax for Var (e.g. start with an upper case) and Field (e.g. start with a lower case) so that we can assume that $\mathsf{Var} \cap \mathsf{Field} = \emptyset$. Another solution is to have the same syntax for Var and Field with some way to make the distinction each time an identifier of $\mathsf{Var} \cup \mathsf{Field}$ is used. For example, a rule might be that all identifiers not appearing after a `var` or `proc` are Field identifiers. Another rule would be that Field identifiers are those identifiers appearing somewhere in the program after a dot, in which case they cannot be a variable or a procedure. Another solution is to allow the use of the same identifier for variables, fields, and procedures, which one is meant depending on the context of use. In case of ambiguity, one can decide on a priority, e.g. first procedure, then variable, else field.

[2] The substraction `-` is left associative so $e_1 - e_2 - e_3 = ((e_1 - e_2) - e_3)$. The only arithmetic operator that must be implemented is the difference of integer expressions $e_1 - e_2$. Optionally, arithmetic operators can be implemented.

[3] The field selection `.` is left associative so $e_1.e_2.e_3 = ((e_1.e_2).e_3)$.

[4] The only Boolean operator that must be implemented is $e_1 < e_2$ to compare arithmetic expressions, Optionally, other comparison and logic operators can be implemented.

[5] The field selection `.` has the highest priority, the substraction `-` has a medium priority and the assignment `=` has the lowest priority so `z = x.f - y` is `z = ((x.f) - y)`.

[6] We take some liberty with the language grammar and omit parentheses. Formally, we should write `var r; var h; {h = 1; {var p; p = proc y:  r = y + h; var h; {h = 2; p(4)}}}}`.

3. Static semantics

The static semantics defines the context conditions that cannot be defined by a context-free grammar (in the form of an attribute grammar).

The fields and variables are assumed to be distinguished by the lexer (e.g. variables are identifiers starting by an upper-case letter while fields are identifiers starting by a lower-case letter)[7].

The visible variables $e.V$ are the set of declared variables that can be used in an expression e.

The error $e.E$ is true if and only if a variable is used in expression e while not declared in its scope. Similarly for boolean expressions and commands.

3.1. Static semantics of expressions

$$\begin{array}{rcll}
e & ::= & \texttt{f} & e.E = \mathit{false} \\
 & \mid & \texttt{1} & e.E = \mathit{false} \\
 & \mid & e_1 \texttt{ - } e_2 & e.E = e_1.E \vee e_2.E \\
 & & & e_1.V = e.V \\
 & & & e_2.V = e.V \\
 & \mid & \texttt{null} & e.E = \mathit{false} \\
 & \mid & \texttt{x} & e.E = (\texttt{x} \notin e.V) \\
 & \mid & e_1\texttt{.}e_2 & e.E = e_1.E \vee e_2.E \\
 & & & e_1.V = e.V \\
 & & & e_2.V = e.V \\
 & \mid & \texttt{proc y:}C & e.E = C.E \\
 & & & C.V = e.V \cup \{\texttt{y}\}
\end{array}$$

3.2. Static semantics of Boolean expressions

$$\begin{array}{rcll}
b & ::= & \texttt{true} \mid \texttt{false} & b.E = \mathit{false} \\
 & \mid & e_1 \texttt{ == } e_2 \mid e \texttt{ < } e \mid \ldots & b.E = e_1.E \vee e_2.E \\
 & & & e_1.V = b.V \\
 & & & e_2.V = b.V
\end{array}$$

[7] The static semantics could also be changed to distinguish between variables and fields according to their use.

3.3. Static semantics of commands

$$
\begin{array}{rcll}
C & ::= & \texttt{var x;}\, C_1 & C.E = C_1.E \\
 & & & C_1.V = C.V \cup \{\texttt{x}\} \\
 & \mid & e_1(e_2) & C.E = e_1.E \vee e_2.E \\
 & & & e_1.V = e_2.V = C.V \\
 & \mid & \texttt{malloc(x)} & C.E = (\texttt{x} \notin C.V) \\
 & \mid & \texttt{x = } e & C.E = (\texttt{x} \notin C.V) \vee e.E \\
 & & & e.V = C.V \\
 & \mid & e_1 \texttt{.} e_2 \texttt{ = } e_3 & C.E = e_1.E \vee e_2.E \vee e_3.E \\
 & \mid & & e_1.V = e_2.V = e_3.V = C.V \\
 & \mid & \texttt{skip} & C.E = \mathit{false} \\
 & \mid & \{C_1; C_2\} & C.E = C_1.E \vee C_2.E \\
 & & & C_1.V = C_2.V = C.V \\
 & \mid & \texttt{while}\ b\ C_1 & C.E = b.E \vee C_1.E \\
 & & & b.V = C_1.V = C.V \\
 & \mid & \texttt{if}\ b\ C_1\ \texttt{else}\ C_2 & C.E = b.E \vee C_1.E \vee C_2.E \\
 & & & b.V = C_1.V = C_2.V = C.V \\
 & \mid & \{C_1 \mathbin{|||} C_2\} & C.E = C_1.E \vee C_2.E \\
 & & & C_1.V = C_2.V = C.V \\
 & \mid & \texttt{atom}(C_1) & C.E = C_1.E \\
 & & & C_1.V = C.V
\end{array}
$$

4. Transitional semantics

The *semantics* of a language defines for each (syntactically correct) program of the language a formal description of the possible executions of the program. The *transitional semantics* of a program defines a *transition relation* $\Rightarrow$ (corresponding to a possible program computation step) between *configurations* (describing the current state of the computation).

4.1. Semantic domains

In order to define the set *Conf* of configurations (describing the current state of computations), we need

i	$\in$	Int	(Machine) integers
b	$\in$	$\{\mathit{true}, \mathit{false}, \mathit{error}\}$	Booleans
l	$\in$	$\mathit{Objects}$ (where $\mathit{null} \notin \mathit{Objects}$)	Objects
ℓ	$\in$	$\mathit{Loc} \triangleq \mathit{Objects} \cup \{\mathit{null}\}$	Locations
$\nu = \underline{\mathit{clo}}\langle \mathtt{x},\ C,\ \xi\rangle$	$\in$	$\mathit{Clo} \triangleq \underline{\mathit{clo}}(\mathsf{Var} \times \mathsf{Cmd} \times \mathit{Stack})$	Closures
v	$\in$	$\mathit{Val} \triangleq \mathsf{Field} \cup \mathit{Int} \cup \mathit{Loc} \cup \mathit{Clo}$	Values
t	$\in$	$\mathit{Tva} \triangleq \mathit{Val} \cup \{\mathit{error}\}$	Tainted values[8]
ρ	$\in$	$\mathit{Env} \triangleq \mathsf{Var} \not\mapsto \mathit{Objects}$	Environments
$\varphi = \underline{\mathit{decl}}\langle\rho\rangle /$	$\in$	$\mathit{Frame} \triangleq \underline{\mathit{decl}}(\mathit{Env}) \cup$	Frames
$\underline{\mathit{call}}\langle\rho,\ \xi\rangle$		$\underline{\mathit{call}}(\mathit{Env} \times \mathit{Stack})$	
ξ	$\in$	$\mathit{Stack} \triangleq (\mathit{Frame})^\star$	Stacks
h	$\in$	$\mathit{Heap} \triangleq \mathit{Objects} \times \mathsf{Field} \not\mapsto \mathit{Tva}$	Heap
$\sigma = \langle\xi,\ h\rangle$	$\in$	$\mathit{State} \triangleq \mathit{Stack} \times \mathit{Heap}$	States
C	$\in$	Ctrl ::= `skip` $\mid \{\mathit{Ctrl}; \mathit{Ctrl}\} \mid$ `while` b Ctrl $\mid$	Control
		`if` b Ctrl `else` $\mathit{Ctrl} \mid \{\mathit{Ctrl} \parallel\mid \mathit{Ctrl}\} \mid$ `atom`$(\mathit{Ctrl}) \mid$	
		$\underline{\mathit{block}}(\mathit{Ctrl})$	
$\Gamma = \langle C,\ \sigma\rangle / \sigma$	$\in$	$\mathit{Conf} \triangleq (\mathit{Ctrl} \times \mathit{State}) \cup \mathit{State} \cup \{\mathit{error}\}$	Configurations

- An object l is a non-*null* location on the heap where the value and procedure fields of the object are stored.
- A location ℓ is a non-empty object location l or empty *null*.
- A closure $\nu = \underline{\mathit{clo}}\langle \mathtt{x},\ C,\ \xi\rangle$ records a procedure value that is the formal parameter `x`, the procedure body C, and the stack ξ providing, according to the lexical scoping rules, the object locations of the global variables appearing in the procedure body C, if any.
- The values are fields, integers, object locations, and procedure closures. Booleans are not values since they cannot be stored in variables or fields of objects.
- The tainted values are either a value or *error* resulting from a runtime error.
- The environments ρ are partial functions mapping the variables of a block that are visible according to the static scoping rules to the location of their value on the heap. The lifetime of the value of the variable can be longer than that of the variable (which is limited to the declaration bloc).
- A frame $\varphi = \underline{\mathit{decl}}\langle\rho\rangle$ records a variable location in the environment ρ. A frame $\varphi = \underline{\mathit{call}}\langle\rho,\ \xi\rangle$ records a formal parameter location in environment ρ and records the calling stack ξ (while the procedure body is being evaluated in the declaration stack for globals).

[8] The special value *error* is returned when computations go wrong.

- A stack ξ records the location of all visible variables in the lexical scope of enclosing blocks and formal parameters declared in called procedures. The notation $\mathit{Stack} \triangleq (\mathit{Frame})^\star$ defines the stack as a sequence of 0 or more *Frame*s.
- The heap h stores the value of the fields of objects as well as the value of variables (in a reserved field `val`).
- The state $\sigma = \langle \xi,\ h \rangle$ is made up of a stack ξ (recording the heap location of values of visible variables and parameters of procedures currently being called) and a heap h mapping these locations to values of variables, parameters, and allocated objects.
- The control state is a command C or $\underline{\mathit{block}}(C)$ to remember that the stack must be popped when execution of block C terminates.

  It follows from the definition of the small step semantics that the control state of a program is finite. It contains all the commands of the program plus $\underline{\mathit{block}}(C')$ for all possible control states C' of command C in variable declarations `var x`$;C$ or procedures `proc x:`C, and $\{C';$`while` $b\ C\}$ for all control states C' of command C in program loops `while` $b\ C$. Thus control states can easily be represented by labels/program points.
- A configuration is either $\Gamma = \langle C,\ \sigma \rangle$ recording that in state σ, the control C remains to be executed or $\Gamma = \sigma$ for a final state σ of the program where execution is terminated. So in $\Gamma = \langle C,\ \sigma \rangle$, C is the control state and σ is the memory state while in the second case $\Gamma = \sigma$, the control state is empty since execution is finished.

4.2. Operations on semantic domains

4.2.1. *Operations on sequences*

- A (finite) sequence $\xi \in S^*$ of elements of a set S is either
  - the empty sequence ε, or
  - the concatenation $\xi \cdot s$ of a sequence ξ with an element $s \in S$ (such that $\varepsilon \cdot s = s$ is a one element sequence)
- A sequence ξ of elements $\xi_1, \xi_2, \ldots, \xi_n$, $n \geqslant 0$ is written $\xi_1 \cdot \xi_2 \cdot \cdots \cdot \xi_n$, or $\xi = \xi_1\xi_2 \ldots \xi_n$ for brevity. When $n = 0$, $\xi = \varepsilon$ is the empty sequence.

4.2.2. *Operations on partial functions*

- For a partial function $f \in \mathcal{X} \not\mapsto \mathcal{Y}$[9], we let $\mathit{dom}(f) \in 2^{\mathcal{X}}$ be its domain[10].
- The empty function $\emptyset \in \mathcal{X} \not\mapsto \mathcal{Y}$ has an empty domain $\mathit{dom}(\emptyset) = \emptyset$. It is undefined for all its arguments.
- For $x, y \in \mathcal{X}$ and $v \in \mathcal{Y}$, we have

[9] $\mathcal{X} \not\mapsto \mathcal{Y} \triangleq \{f \in 2^{\mathcal{X}\times\mathcal{Y}} \mid \forall x \in \mathcal{X} : \forall y, y' \in \mathcal{Y} : (\langle x,\ y \rangle \in f \wedge \langle x,\ y' \rangle \in f) \Longrightarrow (y = y')\}$ is the set of all partial functions from $\mathcal{X}$ into $\mathcal{Y}$. We write $f \in \mathcal{X} \mapsto \mathcal{Y}$ when f is total, that is $f(x)$ is well-defined for all $x \in \mathcal{X}$, formally $\forall x \in \mathcal{X} : \exists y \in \mathcal{Y} : \langle x,\ y \rangle \in f$.

[10] $2^{\mathcal{X}}$ also denoted $\wp(\mathcal{X})$ is the set of all subsets of the set $\mathcal{X}$. It can be encoded as a map from $\mathcal{X}$ into the Booleans $2 = \{0.1\}$ hence the notation $2^{\mathcal{X}}$ when $\mathcal{Y}^{\mathcal{X}}$ denotes the set of total functions from $\mathcal{X}$ into $\mathcal{Y}$, which we write $\mathcal{X} \mapsto \mathcal{Y}$.

$$\begin{array}{rcll} \mathit{dom}(f[x \mapsto v]) & \triangleq & \mathit{dom}(f) \cup \{x\}, & \\ f[x \mapsto v](x) & \triangleq & v & \\ f[x \mapsto v](y) & \triangleq & f(y) & \text{when } y \in \mathit{dom}(f) \setminus \{x\}. \end{array}$$

- In particular $\emptyset[x \mapsto v] \triangleq \{\langle x,\, v\rangle\}$ has domain $\{x\}$.
- These notations are used both for environments $\rho \in \mathit{Env}$ and heaps $h \in \mathit{Heap}$. The semantics definition is designed in order to prevent using $f(x)$ when $x \notin \mathit{dom}(f)$.

4.2.3. *Operations on stacks*

- The stacks $\xi \in \mathit{Stack}$ have syntax $\xi ::= \epsilon \mid \xi \cdot \varphi$ that is $\varphi_1 \ldots \varphi_n$ where $n \geqslant 0$ and $n = 0$ means that the stack is empty. New frames φ are pushed on the right of the stack ξ to get $\xi \cdot \varphi$.
- The empty stack is $\epsilon \in \mathit{Stack}$ corresponds to an an empty initial environment (no variable/procedure has been declared).
- Otherwise a stack has the form $\xi \cdot \varphi$ where φ is the current frame and ξ corresponds to previous (recursive) procedure calls.
- The stack domain is $\mathit{dom} \in \mathit{Stack} \mapsto 2^{\mathsf{Var}}$ such that $\mathit{dom}(\epsilon) \triangleq \emptyset$ and $\mathit{dom}(\xi \cdot \varphi) \triangleq \mathit{dom}(\xi) \cup \mathit{dom}(\varphi)$ where $\mathit{dom}(\underline{\mathit{decl}}\langle\rho\rangle) \triangleq \mathit{dom}(\rho)$ and $\mathit{dom}(\underline{\mathit{call}}\langle\rho,\, \xi\rangle) \triangleq \mathit{dom}(\rho)$.

  So $\mathit{dom}(\xi)$ provides locations of all variables, procedures and parameter visible in the lexical scope.

4.2.4. *Location of a variable in a stack and a frame*

- The location of a variable $\mathtt{x}$ in a stack ξ is undefined whenever $\mathtt{x} \notin \mathit{dom}(\xi)$ (but this is impossible in the semantics because of the lexical scope rules).
- When $\mathtt{x} \in \mathit{dom}(\xi)$ then we recursively define the location $\xi(\mathtt{x})$ of a variable $\mathtt{x}$ in a stack ξ as $(\xi \cdot \varphi)(\mathtt{x}) = \varphi(\mathtt{x})$ when $\mathtt{x} \in \mathit{dom}(\varphi)$ and $\xi(\mathtt{x})$ otherwise.
- Moreover $\underline{\mathit{decl}}\langle\rho\rangle(\mathtt{x}) \triangleq \rho(\mathtt{x})$ and $\underline{\mathit{call}}\langle\rho,\, \xi\rangle(\mathtt{x}) \triangleq \rho(\mathtt{x})$.

4.2.5. *Value of a variable in a state*

- To obtain the value of a variable $\mathtt{x} \in \mathsf{Var}$ in a state $\langle\xi,\, h\rangle$,
  - the stack ξ provides its location $l = \xi(\mathtt{x}) \in \mathit{Loc}$ (which cannot be null[11]), and
  - the heap provides its value $h(\langle l,\, \mathtt{val}\rangle) = v$, $v \in \mathit{Val}$ (which is mutable).

4.2.6. *Allocated heap locations*

- It is easy to prove by structural induction on commands that for all states $\langle\xi,\, h\rangle$, the locations created on the heap are $\mathit{loc}(h)$ defined as

$$\begin{array}{rcl} \mathit{loc} & \in & \mathit{Heap} \mapsto 2^{\mathit{Objects}} \\ \mathit{loc}(h) & \triangleq & = \{l \mid \exists \mathtt{f} \in \mathsf{Field} : \langle l,\, \mathtt{f}\rangle \in \mathit{dom}(h)\} \end{array}$$

[11] since $\mathit{Env} \triangleq \mathsf{Var} \not\mapsto \mathit{Objects}$.

4.3. **Operational semantics**

- The small-step structural operational semantics [**?**] of commands specifies the transition relation $\Rightarrow \in 2^{\mathit{Conf} \times \mathit{Conf}}$.
- $\langle \Gamma,\ \Gamma' \rangle \in \Rightarrow$ (written $\Gamma \Rightarrow \Gamma'$ for brevity) means that if a program execution reaches configuration Γ then the next computation step *may* reach configuration Γ'.
- We say *may* since there might be several possible next configurations Γ'. For sure all possible ones are in $\{\Gamma' \mid \Gamma \Rightarrow \Gamma'\}$.
- This set of possible successors of configuration Γ may be empty, in which case Γ is a *blocking state*, without possible successor $\forall \Gamma' \in \mathit{Conf} : \Gamma \not\Rightarrow \Gamma'$. When reaching such a blocking state. if ever, execution stops.

4.3.1. ***Initial configurations***

- The initial configuration for a command C is
$$\langle C,\ \langle \epsilon,\ \emptyset \rangle \rangle$$
meaning that the command command C must be executed with an initial empty stack ϵ and an initial empty heap $\emptyset$.
- If libraries are used their global variables and procedures must be included in the initial stack and heap.

4.3.2. ***Variable declaration***

- The declaration of a new variable `x` in a block pushes a new environment $\emptyset[\mathtt{x} \mapsto l]$ with a fresh non-*null* location l for `x` and assigns the initial value *null* in the field `val` of this new location l on the heap h (which is extended with this new location l).
- Recall from Sect. 4.2.6, that $\mathit{loc}(h)$ is the set of locations already allocated on the heap h.

$$\begin{array}{rl} & \langle \mathtt{var\ x}; C,\ \langle \xi,\ h \rangle \rangle \\ \Rightarrow & \mathit{let}\ \ l \notin \mathit{loc}(h) \cup \{\mathit{null}\} \\ & \mathit{and}\ \xi' =\ \ \xi \cdot \underline{\mathit{decl}}\langle \emptyset[\mathtt{x} \mapsto l] \rangle \\ & \mathit{and}\ h' =\ \ h[\langle l,\ \mathtt{val} \rangle \mapsto \mathit{null}]\ \mathit{in} \\ & \cdot\ \ \ \langle \underline{\mathit{block}}(C),\ \langle \xi',\ h' \rangle \rangle \end{array}$$

- Whenever a variable `x` is used in the program in a state $\langle \xi,\ h \rangle$, it is assumed that scoping rules have been statically checked to ensure that an environment for this variable `x` does exist on the stack ξ providing a location $\xi(\mathtt{x})$ on the heap h.
- By induction on the program syntax, one can prove that the operational semantics is defined so that the value of the variable `x` does exist on the heap and is $h(\langle \xi(\mathtt{x}),\ \mathtt{val} \rangle)$.

4.3.3. ***Execution of a block***

- The execution of $\underline{\mathit{block}}(C)$ is similar to the execution of command C, except that the frame that was pulled on top of the stack before starting C must be pulled out at the end of the execution of C.

- At the end of the block, the frame φ created for that block is popped off the stack. The lifetime of the location of `x` on the heap is unlimited and so values assigned to `x` may live longer on the heap than the lexical scope of `x`.

$$\frac{\langle C,\ \sigma\rangle \Rightarrow \langle C',\ \sigma'\rangle}{\langle \underline{block}(C),\ \sigma\rangle \Rightarrow \langle \underline{block}(C'),\ \sigma'\rangle}$$

$$\frac{\langle C,\ \langle \xi \cdot \varphi,\ h\rangle\rangle \Rightarrow \langle \xi' \cdot \underline{decl}\langle \rho'\rangle,\ h'\rangle}{\langle \underline{block}(C),\ \langle \xi \cdot \varphi,\ h\rangle\rangle \Rightarrow \langle \xi',\ h'\rangle}$$

4.3.4. ***Expressions***

The evaluation of an expression $e \in \mathsf{Exp}$ in a state σ returns the tainted value of e so $eval[\![e]\!] \in State \mapsto Tva$.

$$eval[\![\texttt{f}]\!]\langle \xi,\ h\rangle \triangleq \texttt{f} \qquad \text{Fields}$$

$$\begin{array}{rcl}
eval[\![\texttt{1}]\!]\langle \xi,\ h\rangle & \triangleq & 1 \qquad \text{Tainted arithmetics}\\
eval[\![e_1\ \texttt{-}\ e_2]\!]\langle \xi,\ h\rangle & \triangleq & let\ \ v_1\ =\ eval[\![e_1]\!]\langle \xi,\ h\rangle\\
& & and\ v_2\ =\ eval[\![e_2]\!]\langle \xi,\ h\rangle\ in\\
& & \quad if\ \ v_1 \in Int \wedge v_2 \in Int\\
& & \qquad then\ v_1 - v_2\\
& & \qquad else\ \ error
\end{array}$$

$$\begin{array}{rcl}
eval[\![\texttt{null}]\!]\langle \xi,\ h\rangle & \triangleq & null \qquad \text{Tainted locations}\\
eval[\![\texttt{x}]\!]\langle \xi,\ h\rangle & \triangleq & h(\langle \xi(\texttt{x}),\ \texttt{val}\rangle)\\
eval[\![e\texttt{.}e']\!]\langle \xi,\ h\rangle & \triangleq & let\ \ l\ =\ eval[\![e]\!]\langle \xi,\ h\rangle\\
& & and\ \texttt{f}\ =\ eval[\![e']\!]\langle \xi,\ h\rangle\ in\\
& & \quad if\ \ l \in loc(h) \wedge \texttt{f} \in \mathsf{Field} \wedge \langle l,\ \texttt{f}\rangle \in dom(h)\ then\\
& & \qquad h(\langle l,\ \texttt{f}\rangle)\\
& & \quad else\ \ error
\end{array}$$

$$eval[\![\texttt{proc x:}C]\!]\langle \xi,\ h\rangle \triangleq \underline{clo}\langle \texttt{x},\ C,\ \xi\rangle \qquad \text{Procedure closures}$$

- In the above definition of $eval[\![\texttt{x}]\!]\langle \xi,\ h\rangle$, $h(\langle \xi(\texttt{x}),\ \texttt{val}\rangle)$, is well defined by lexical scope rules (so that the declaration of `x` has assigned a possibly *null* tainted value to the location $\xi(\texttt{x})$ of `x`).
- However typing `x.f` may not ensure that the tainted value of `x` is a valid heap location when accessing the field `f`, so a runtime check is necessary.
- The value of a procedure is a closure.
- The closure records the formal parameter `x`, the procedure body C, and the stack ξ at declaration time to record the lexicographic binding of the global variables in the procedure body.

- For recursive calls, the closure stack must contain a variable or field which value is the closure.

4.3.5. *Boolean expressions*

- The evaluation of a Boolean expression $b \in \mathsf{Bool}$ in a state σ returns a Boolean $\mathit{bool}[\![b]\!]\sigma \in \{\mathit{true}, \mathit{false}, \mathit{error}\}$ and has no side-effects (the state is not modified).
- Runtime type checking avoids errors (such as comparing an integer with a location or a procedure).

$$
\begin{aligned}
\mathit{bool}[\![\mathtt{true}]\!]\sigma &\triangleq \mathit{true}\\
\mathit{bool}[\![\mathtt{false}]\!]\sigma &\triangleq \mathit{false}\\
\mathit{bool}[\![e_1 \;\mathtt{==}\; e_2]\!]\langle\xi,\, h\rangle &\triangleq \mathit{let}\ v_1 \ =\ \mathit{eval}[\![e_1]\!]\langle\xi,\, h\rangle\ \mathit{and}\ v_2 \ =\ \mathit{eval}[\![e_2]\!]\langle\xi,\, h\rangle\\
&\ \mathit{in}\ \mathit{if}\ (v_1 \in \mathit{Int} \wedge v_2 \in \mathit{Int}) \vee (v_1 \in \mathit{Loc} \wedge v_2 \in \mathit{Loc})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \vee (v_1 \in \mathit{Clo} \wedge v_2 \in \mathit{Clo})\\
&\qquad \mathit{then}\ v_1 = v_2\\
&\qquad \mathit{else}\ \mathit{error}\\
\mathit{bool}[\![e_1 \;\mathtt{<}\; e_2]\!]\langle\xi,\, h\rangle &\triangleq \mathit{let}\ v_1 \ =\ \mathit{eval}[\![e_1]\!]\langle\xi,\, h\rangle\ \mathit{and}\ v_2 \ =\ \mathit{eval}[\![e_2]\!]\langle\xi,\, h\rangle\\
&\ \mathit{in}\ \mathit{if}\ (v_1 \in \mathit{Int} \wedge v_2 \in \mathit{Int})\\
&\qquad \mathit{then}\ v_1 < v_2\\
&\qquad \mathit{else}\ \mathit{error}
\end{aligned}
$$

4.3.6. *Recursive procedure call*

- In a recursive procedure call $e(e')$, the closure value $\underline{\mathit{clo}}\langle\mathtt{z},\, C,\, \xi'\rangle$ of e is recovered on the heap.
- The procedure body C is evaluated in the context of the declaration stack ξ' for global variables on which the formal parameter $\mathtt{z}$ is pushed and initialized at a new location l with the value of the actual parameter e'.

$$
\begin{array}{ll}
 & \langle e(e'),\, \langle\xi,\, h\rangle\rangle\\
\Rightarrow & \mathit{let}\ v \ =\ \mathit{eval}[\![e]\!]\langle\xi,\, h\rangle\ \mathit{in}\\
 & \mathit{match}\ v\ \mathit{with}\\
 & \quad \mid \underline{\mathit{clo}}\langle\mathtt{x},\, C,\, \xi'\rangle \ \rightarrow\ \mathit{let}\ l \notin \mathit{loc}(h) \cup \{\mathit{null}\}\\
 & \qquad\qquad\qquad\qquad\quad\ \mathit{and}\ \xi'' \ =\ \xi' \cdot \underline{\mathit{call}}\langle\emptyset[\mathtt{x} \mapsto l],\, \xi\rangle\\
 & \qquad\qquad\qquad\qquad\quad\ \mathit{and}\ h' \ =\ h[\langle l,\, \mathtt{val}\rangle \mapsto \mathit{eval}[\![e']\!]\langle\xi,\, h\rangle]\ \mathit{in}\\
 & \qquad\qquad\qquad\qquad\qquad \langle\underline{\mathit{block}}(C),\, \langle\xi'',\, h'\rangle\rangle\\
 & \quad \mid \_ \ \rightarrow\ \mathit{error}
\end{array}
$$

- After evaluation of the body with global variables evaluated in the declaration environment ξ', the calling environment ξ is restituted.

$$\frac{\langle C,\ \langle \xi' \cdot \varphi,\ h\rangle\rangle \Rightarrow \langle \xi'' \cdot \underline{call}\langle \rho',\ \xi\rangle,\ h'\rangle}{\langle \underline{block}(C),\ \langle \xi' \cdot \varphi,\ h\rangle\rangle \Rightarrow \langle \xi,\ h'\rangle}$$

4.3.7. ***Assignment***

In the following

- the fact that $\xi(\texttt{x})$ exists follows from the language scoping rule, while
- the fact that $\xi(\texttt{x})$ is an existing heap location so that $h(\langle \xi(\texttt{x}),\ \texttt{val}\rangle)$ is well-defined follows from the definition of the operational semantics (whenever the scope of a variable is entered, its new location is allocated on the heap).

$$\begin{array}{l}\langle \texttt{x = } e,\ \langle \xi,\ h\rangle\rangle \Rightarrow \textit{match eval}[\![e]\!]\langle \xi,\ h\rangle\ \textit{with}\\ \qquad\qquad\qquad\quad \mid \textit{error} \ \rightarrow\ \textit{error}\\ \qquad\qquad\qquad\quad \mid v \ \rightarrow\ \langle \xi,\ h[\langle \xi(\texttt{x}),\ \texttt{val}\rangle \mapsto v]\rangle\end{array}$$

$$\begin{array}{l}\langle e\texttt{.}e' \texttt{ = } e'',\ \langle \xi,\ h\rangle\rangle \Rightarrow \textit{let}\quad l\ =\ \textit{eval}[\![e]\!]\langle \xi,\ h\rangle\\ \qquad\qquad\qquad\qquad\quad \textit{and}\ \texttt{f}\ =\ \textit{eval}[\![e']\!]\langle \xi,\ h\rangle\ \textit{in}\\ \qquad\qquad\qquad\qquad\qquad \textit{if}\ l \notin \textit{loc}(h) \vee \texttt{f} \notin \mathsf{Field} \vee \langle l,\ \texttt{f}\rangle \notin \textit{dom}(h)\ \textit{then error}\\ \qquad\qquad\qquad\qquad\qquad \textit{else let}\ v''\ =\ \textit{eval}[\![e'']\!]\langle \xi,\ h\rangle\ \textit{in}\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \langle \xi,\ h[\langle l,\ \texttt{f}\rangle \mapsto v'']\rangle\end{array}$$

4.3.8. ***Dynamic allocation***

$$\begin{array}{rl} & \langle \texttt{malloc(x)},\ \langle \xi,\ h\rangle\rangle\\ \Rightarrow & \textit{let}\quad l \notin \textit{loc}(h) \cup \{\textit{null}\}\\ & \textit{and}\ h' =\ h[\langle \xi(\texttt{x}),\ \texttt{val}\rangle \mapsto l] \cup I(l)\ \textit{in}\\ & \quad \langle \xi,\ h'\rangle\\ & \text{where}\quad I(l) \triangleq \{\langle\langle l,\ \texttt{f}\rangle,\ \textit{null}\rangle \mid \texttt{f} \in \mathsf{Field}\}\end{array}$$

- A new heap location l is assigned to variable `x` with all fields `f` initially *null*.
- In practice only some fields will exist (which must all appear in the program text) and need to be initialized to *null*
- In the field assignment of Sect. 4.3.7, the existence of the field is checked before accessing it although this is redundant thanks to the above initialization.

4.3.9. ***Sequential control***

$$\langle \texttt{skip},\, \sigma\rangle \Rightarrow \sigma$$

$$\begin{array}{c}\langle \texttt{if}\ b\ C_1\ \texttt{else}\ C_2,\, \sigma\rangle \Rightarrow \langle C_1,\, \sigma\rangle, \quad \mathit{bool}[\![b]\!]\sigma = \mathit{true}\\ \langle \texttt{if}\ b\ C_1\ \texttt{else}\ C_2,\, \sigma\rangle \Rightarrow \langle C_2,\, \sigma\rangle, \quad \mathit{bool}[\![b]\!]\sigma = \mathit{false}\end{array}$$

$$\begin{array}{c}\langle \texttt{while}\ b\ C_1,\, \sigma\rangle \Rightarrow \langle C_1; \texttt{while}\ b\ C_1,\, \sigma\rangle, \quad \mathit{bool}[\![b]\!]\sigma = \mathit{true}\\ \langle \texttt{while}\ b\ C_1,\, \sigma\rangle \Rightarrow \sigma, \quad \mathit{bool}[\![b]\!]\sigma = \mathit{false}\end{array}$$

$$\frac{\langle C_1,\, \sigma\rangle \Rightarrow \langle C_1',\, \sigma'\rangle}{\langle C_1; C_2,\, \sigma\rangle \Rightarrow \langle C_1'; C_2,\, \sigma'\rangle}$$
$$\frac{\langle C_1,\, \sigma\rangle \Rightarrow \sigma'}{\langle C_1; C_2,\, \sigma\rangle \Rightarrow \langle C_2,\, \sigma'\rangle}$$

4.3.10. ***Parallelism***

A computation step in $C_1 \mathbin{|||} C_2$ is a computation step in C_1 or in C_2.

$$\frac{\langle C_1,\, \sigma\rangle \Rightarrow \langle C_1',\, \sigma'\rangle}{\langle C_1 \mathbin{|||} C_2,\, \sigma\rangle \Rightarrow \langle C_1' \mathbin{|||} C_2,\, \sigma'\rangle}$$
$$\frac{\langle C_1,\, \sigma\rangle \Rightarrow \sigma'}{\langle C_1 \mathbin{|||} C_2,\, \sigma\rangle \Rightarrow \langle C_2,\, \sigma'\rangle}$$
$$\frac{\langle C_2,\, \sigma\rangle \Rightarrow \langle C_2',\, \sigma'\rangle}{\langle C_1 \mathbin{|||} C_2,\, \sigma\rangle \Rightarrow \langle C_1 \mathbin{|||} C_2',\, \sigma'\rangle}$$
$$\frac{\langle C_2,\, \sigma\rangle \Rightarrow \sigma'}{\langle C_1 \mathbin{|||} C_2,\, \sigma\rangle \Rightarrow \langle C_1,\, \sigma'\rangle}$$

There is no fairness hypothesis, meaning that C_1 or C_2 may run for ever while C_2 or C_1 remains idle for ever.

4.3.11. ***Atomicity***

- We let $\Rightarrow^\star$ be the transitive closure of $\Rightarrow$ right-restricted to a state (that is a final configuration).
- Otherwise stated $\Gamma \Rightarrow^\star \sigma$ if and only if $\exists \Gamma_1, \Gamma_2, \ldots, \Gamma_n \in \mathit{Conf} : n \geqslant 1 \wedge \Gamma_1 = \Gamma \wedge \Gamma_1 \Rightarrow \Gamma_2 \wedge \ldots \wedge \Gamma_{n-1} \Rightarrow \Gamma_n \wedge \Gamma_n \Rightarrow \sigma$ (where $\Gamma_1 \Rightarrow \Gamma_2 \wedge \ldots \wedge \Gamma_{n-1} \Rightarrow \Gamma_n$ is true for $n = 1$).
- $\Rightarrow^\star$ can be defined by the rules

$$\frac{\langle C,\, \sigma\rangle \Rightarrow \sigma'}{\langle C,\, \sigma\rangle \Rightarrow^\star \sigma'}$$
$$\frac{\langle C,\, \sigma\rangle \Rightarrow \langle C',\, \sigma'\rangle, \quad \langle C',\, \sigma'\rangle \Rightarrow^\star \sigma''}{\langle C,\, \sigma\rangle \Rightarrow^\star \sigma''}$$

- Atomicity $\texttt{atom}(C)$ forces the command C to be executed up to termination (if ever).

- No other parallel process, if any, can interact with the execution of C (*e.g.* by simultaneously modifying the heap) while executing $\texttt{atom}(C)$.

$$\frac{\langle C,\ \sigma\rangle \Rightarrow^{\star} \sigma'}{\langle \texttt{atom}(C),\ \sigma\rangle \Rightarrow \sigma'}$$

For example `{x = 0; x = x+1; x= x+1` $|||$ `x = 0}` will terminate with $\texttt{x} = 0$, 1, or 2 while `{x = 0; atom(x = x+1; x= x+1)` $|||$ `x = 0}` will terminate with $\texttt{x} = 0$ or 2.

4.4. **Example**

The execution trace of the program

```
var p; p = proc y:if y < 1 then p = 1 else p(y - 1); p(1)
```

is

$\langle$`var p; p = proc y:if y < 1 then p = 1 else p(y - 1); p(1)`$, \langle \epsilon,\ \emptyset\rangle\rangle$

$\wr$Initial state, Sect. 4.3.1$\int$

$\Rightarrow \langle \underline{\mathit{block}}($`p = proc y:if y < 1 then p = 1 else p(y - 1); p(1)`$),\ \langle \xi_1,\ h_1\rangle\rangle$

$\wr$Variable declaration, Sect. 4.3.2$\int$

$\wr$where $\xi_1 \triangleq \epsilon \cdot \underline{\mathit{decl}}\langle \emptyset[\texttt{p} \mapsto l_1]\rangle$ and $h_1 \triangleq \emptyset[\langle l_1,\ \texttt{val}\rangle \mapsto \mathit{null}]$$\int$

$\Rightarrow \langle \underline{\mathit{block}}(\texttt{p(1)}),\ \langle \xi_1,\ h_2\rangle\rangle$ $\wr$Assignment, Sect. 4.3.7$\int$

$\wr$where $h_2 \triangleq \emptyset[\langle l_1,\ \texttt{val}\rangle \mapsto \langle \underline{\mathit{clo}}\langle$`y, if y < 1 then p = 1 else p(y - 1)`$,\ \xi_1\rangle,\ \emptyset\rangle]$$\int$

$\Rightarrow \langle \underline{\mathit{block}}(\underline{\mathit{block}}($`if y < 1 then p = 1 else p(y - 1)`$)),\ \langle \xi_2,\ h_3\rangle\rangle$

$\wr$Procedure call, Sect. 4.3.6$\int$

$\wr$where $\xi_2 \triangleq \xi_1 \cdot \underline{\mathit{call}}\langle \emptyset[\texttt{y} \mapsto l_2],\ \xi_1\rangle$ and $h_3 \triangleq h_2[\langle l_2,\ \texttt{val}\rangle \mapsto 1]$$\int$

$\Rightarrow \langle \underline{\mathit{block}}(\underline{\mathit{block}}(\texttt{p(y - 1)})),\ \langle \xi_2,\ h_3\rangle\rangle$ $\wr$Conditional, Sect. 4.3.9$\int$

$\Rightarrow \langle \underline{\mathit{block}}(\underline{\mathit{block}}(\underline{\mathit{block}}($`if y < 1 then p = 1 else p(y - 1)`$))),\ \langle \xi_3,\ h_4\rangle\rangle$

$\wr$Recursive procedure call, Sect. 4.3.6$\int$

$\wr$where $\xi_3 \triangleq \xi_2 \cdot \underline{\mathit{call}}\langle \emptyset[\texttt{y} \mapsto l_3],\ \xi_2\rangle$ and $h_4 \triangleq h_3[\langle l_3,\ \texttt{val}\rangle \mapsto 0]$$\int$

$\Rightarrow \langle \underline{\mathit{block}}(\underline{\mathit{block}}(\underline{\mathit{block}}(\texttt{p = 1}))),\ \langle \xi_3,\ h_4\rangle\rangle$ $\wr$Conditional, Sect. 4.3.9$\int$

$\Rightarrow \langle \epsilon,\ h_5\rangle$ $\wr$Assignment, Sect. 4.3.7 $\int$

$\wr$where $h_5 \triangleq h_4[\langle l_1,\ \texttt{val}\rangle \mapsto 1]$ since $\langle \texttt{p = 1},\ \langle \xi_3,\ h_4\rangle\rangle \Rightarrow \langle \xi_3,\ h_5\rangle$ so $\langle \underline{\mathit{block}}(\texttt{p = 1}),\ \langle \xi_3,\ h_4\rangle\rangle \Rightarrow \langle \xi_2,\ h_5\rangle$ hence $\langle \underline{\mathit{block}}(\underline{\mathit{block}}(\texttt{p = 1})),\ \langle \xi_3,\ h_4\rangle\rangle \Rightarrow \langle \xi_1,\ h_5\rangle$ proving $\langle \underline{\mathit{block}}(\underline{\mathit{block}}(\underline{\mathit{block}}(\texttt{p = 1}))),\ \langle \xi_3,\ h_4\rangle\rangle \Rightarrow \langle \epsilon,\ h_5\rangle$ by the stack popping of the block exit rule in Sect. 4.3.6 $\int$

5. Trace semantics

- A (partial) execution of a program C in initial state σ is a sequence of configurations $\Gamma_1\Gamma_2\ldots\Gamma_n$ such that $\Gamma_1 = \langle C,\ \sigma\rangle$ and for all $i = 1, \ldots, n-1$, $\Gamma_i \Rightarrow \Gamma_{i+1}$.

- The semantics of the program is the set $[\![C]\!]I$ of all such executions starting from initial states $\sigma \in I$, $I \in 2^{\mathit{State}}$.

Definition 1. *The trace semantics of a command C is*

$$\begin{array}{rcl} [\![C]\!] & \in & 2^{\mathit{State}} \mapsto 2^{\mathit{Conf}^\star} \\ [\![C]\!]I & \triangleq & \{\Gamma_1\Gamma_2\ldots\Gamma_n \mid \Gamma_1 = \langle C,\, \sigma\rangle \wedge \sigma \in I \wedge \forall i = 1, \ldots, n-1 : \Gamma_i \Rightarrow \Gamma_{i+1}\} \end{array}$$

Lemma 1.

$$\begin{array}{rcl} [\![C]\!]I & = & \mathit{lfp}^{\subseteq}_{\emptyset}\ F[\![C]\!]I \\ \textit{where}\quad F[\![C]\!]I & \triangleq & \boldsymbol{\lambda}\mathcal{X} \cdot \big\{\langle C,\, \sigma\rangle \bigm| \sigma \in I\big\} \cup \big\{\Gamma_1 \ldots \Gamma_{n-1}\Gamma_n \bigm| \Gamma_1 \ldots \Gamma_{n-1} \in \mathcal{X} \wedge \Gamma_{n-1} \Rightarrow \Gamma_n\big\} \end{array}$$

Proof. $F[\![C]\!]I$ is continuous so the result is proved by calculating the fixpoint iterates. □

This can be generalized to include infinite traces describing the non-terminating executions [**?**].

6. Absence of Runtime Errors

The objective is to prove statically or check dynamically the absence of errors at runtime.

Definition 2 (Runtime error)**.** *Execution of program C in initial states I leads to a runtime error if and only if there exists an execution of the form $\Gamma_1\Gamma_2\ldots\Gamma_n$ in $[\![C]\!]I$ such that for all $\Gamma' \in (\mathit{Ctrl} \times \mathit{State}) \cup \{\mathit{error}\} : \Gamma_n \not\Rightarrow \Gamma'$ (i.e. Γ_n is a non-terminal blocking state).*

- The problem of absence of runtime errors is undecidable
- Static analyzers like Astrée [**?**] based on abstract interpretation [**?**, **?**] can provide sound solutions (no error is ever forgotten) but incomplete (some potential errors may not be actual errors due to the imprecision of the analysis).