Compilation and Program Analysis (#2): Semantics

Yannick Zakowski

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025

Intro

Contact me:
web: https://perso.ens-lyon.fr/yannick.zakowski/
email: yannick.zakowski@ens-lyon.fr

Credits: JC Filliâtre / JC Fernandez / Nielson-Nielson-Hankin / Laure Gonnord / Ludovic Henrio

Note on organisation:

1: Course

2: **exercises and proofs during the course** ;

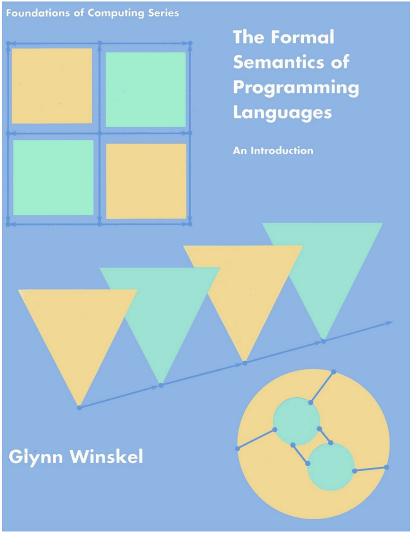
3: **exercises and proofs done at the end the course if we have the time**

1. Semantics: On the Meaning of Programs
2. Operational semantics for mini-while
3. Comparing the different semantics

An old story

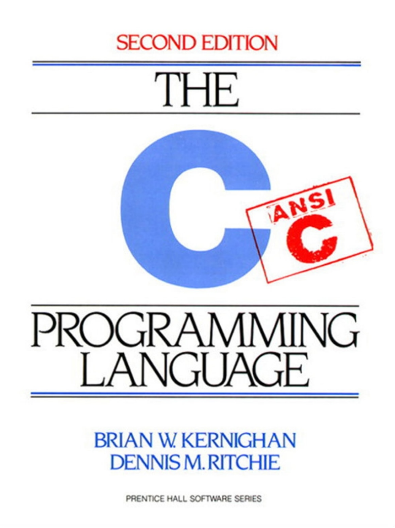
> *As the aim of a programming language is to describe processes, I regard the definition of its semantics as the design, the description of a machine that has as reaction to an arbitrary process description in this language the actual execution of this process. One could also give the semantic definition of the language by stating all the rules according to which one could execute a process, given its description in the language. Fundamentally, there is nothing against this, provided that nothing is left to my imagination as regards the way and the order in which these rules are to be applied. (...) In the design of a language this concept of the "defining machine" should help us to ensure the unambiguity of semantic interpretation of texts.*
>
> (Dijkstra, *On the Design of Machine Independent Programming Languages*, 1961)

Book of the week


Foundations of Computing Series
The Formal Semantics of Programming Languages
An Introduction
Glynn Winskel

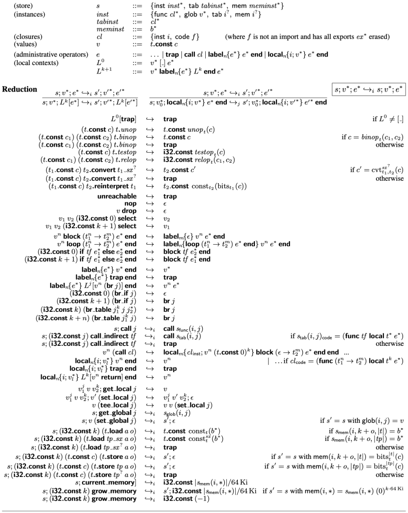
Different degrees of precision

Semi-formal specification in natural language



Different degrees of precision

Formal semantics



$$
\begin{array}{llcll}
\text{(store)} & s & ::= & \{\mathsf{inst}\ inst^*,\ \mathsf{tab}\ tabinst^*,\ \mathsf{mem}\ meminst^*\} & \\
\text{(instances)} & inst & ::= & \{\mathsf{func}\ cl^*,\ \mathsf{glob}\ v^*,\ \mathsf{tab}\ i^?,\ \mathsf{mem}\ i^?\} & \\
 & tabinst & ::= & cl^* & \\
 & meminst & ::= & b^* & \\
\text{(closures)} & cl & ::= & \{\mathsf{inst}\ i,\ \mathsf{code}\ f\} & \text{(where } f \text{ is not an import and has all exports } ex^* \text{ erased)} \\
\text{(values)} & v & ::= & t.\mathbf{const}\ c & \\
\text{(administrative operators)} & e & ::= & \dots \mid \mathbf{trap} \mid \mathbf{call}\ cl \mid \mathbf{label}_n\{e^*\}\ e^*\ \mathbf{end} \mid \mathbf{local}_n\{i; v^*\}\ e^*\ \mathbf{end} & \\
\text{(local contexts)} & L^0 & ::= & v^*\ [\_]\ e^* & \\
 & L^{k+1} & ::= & v^*\ \mathbf{label}_n\{e^*\}\ L^k\ \mathbf{end}\ e^* &
\end{array}
$$

Reduction

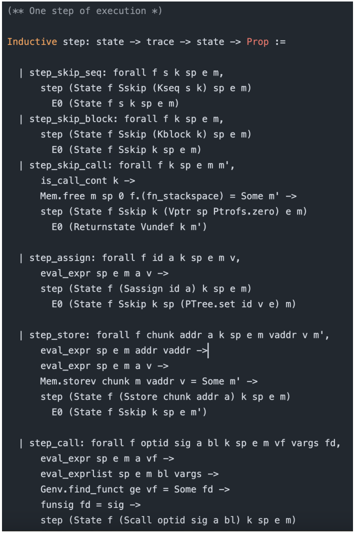
$$
\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v^*; L^k[e^*] \hookrightarrow_i s'; v'^*; L^k[e'^*]}
\qquad
\frac{s; v^*; e^* \hookrightarrow_i s'; v'^*; e'^*}{s; v_0^*; \mathbf{local}_n\{i; v^*\}\ e^*\ \mathbf{end} \hookrightarrow_j s'; v_0^*; \mathbf{local}_n\{i; v'^*\}\ e'^*\ \mathbf{end}}
\qquad
\boxed{s; v^*; e^* \hookrightarrow_i s; v^*; e^*}
$$

$$
\begin{array}{rcll}
L^0[\mathbf{trap}] & \hookrightarrow & \mathbf{trap} & \text{if } L^0 \neq [_] \\
(t.\mathbf{const}\ c)\ t.unop & \hookrightarrow & t.\mathbf{const}\ unop_t(c) & \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.binop & \hookrightarrow & t.\mathbf{const}\ c & \text{if } c = binop_t(c_1, c_2) \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.binop & \hookrightarrow & \mathbf{trap} & \text{otherwise} \\
(t.\mathbf{const}\ c)\ t.testop & \hookrightarrow & \mathbf{i32.const}\ testop_t(c) & \\
(t.\mathbf{const}\ c_1)\ (t.\mathbf{const}\ c_2)\ t.relop & \hookrightarrow & \mathbf{i32.const}\ relop_t(c_1, c_2) & \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{convert}\ t_1_sx^? & \hookrightarrow & t_2.\mathbf{const}\ c' & \text{if } c' = \mathrm{cvt}^{sx^?}_{t_1,t_2}(c) \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{convert}\ t_1_sx^? & \hookrightarrow & \mathbf{trap} & \text{otherwise} \\
(t_1.\mathbf{const}\ c)\ t_2.\mathbf{reinterpret}\ t_1 & \hookrightarrow & t_2.\mathbf{const}\ \mathrm{const}_{t_2}(\mathrm{bits}_{t_1}(c)) & \\
\mathbf{unreachable} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{nop} & \hookrightarrow & \epsilon & \\
v\ \mathbf{drop} & \hookrightarrow & \epsilon & \\
v_1\ v_2\ (\mathbf{i32.const}\ 0)\ \mathbf{select} & \hookrightarrow & v_2 & \\
v_1\ v_2\ (\mathbf{i32.const}\ k+1)\ \mathbf{select} & \hookrightarrow & v_1 & \\
v^n\ \mathbf{block}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end} & \hookrightarrow & \mathbf{label}_m\{\epsilon\}\ v^n\ e^*\ \mathbf{end} & \\
v^n\ \mathbf{loop}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end} & \hookrightarrow & \mathbf{label}_n\{\mathbf{loop}\ (t_1^n \to t_2^m)\ e^*\ \mathbf{end}\}\ v^n\ e^*\ \mathbf{end} & \\
(\mathbf{i32.const}\ 0)\ \mathbf{if}\ tf\ e_1^*\ \mathbf{else}\ e_2^*\ \mathbf{end} & \hookrightarrow & \mathbf{block}\ tf\ e_2^*\ \mathbf{end} & \\
(\mathbf{i32.const}\ k+1)\ \mathbf{if}\ tf\ e_1^*\ \mathbf{else}\ e_2^*\ \mathbf{end} & \hookrightarrow & \mathbf{block}\ tf\ e_1^*\ \mathbf{end} & \\
\mathbf{label}_n\{e^*\}\ v^*\ \mathbf{end} & \hookrightarrow & v^* & \\
\mathbf{label}_n\{e^*\}\ \mathbf{trap}\ \mathbf{end} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{label}_n\{e^*\}\ L^j[v^n\ (\mathbf{br}\ j)]\ \mathbf{end} & \hookrightarrow & v^n\ e^* & \\
(\mathbf{i32.const}\ 0)\ (\mathbf{br_if}\ j) & \hookrightarrow & \epsilon & \\
(\mathbf{i32.const}\ k+1)\ (\mathbf{br_if}\ j) & \hookrightarrow & \mathbf{br}\ j & \\
(\mathbf{i32.const}\ k)\ (\mathbf{br_table}\ j_1^k\ j\ j_2^*) & \hookrightarrow & \mathbf{br}\ j & \\
(\mathbf{i32.const}\ k+n)\ (\mathbf{br_table}\ j_1^k\ j) & \hookrightarrow & \mathbf{br}\ j & \\
s; \mathbf{call}\ j & \hookrightarrow_i & \mathbf{call}\ s_{\mathsf{func}}(i, j) & \\
s; (\mathbf{i32.const}\ j)\ \mathbf{call_indirect}\ tf & \hookrightarrow_i & \mathbf{call}\ s_{\mathsf{tab}}(i, j) & \text{if } s_{\mathsf{tab}}(i,j)_{\mathsf{code}} = (\mathbf{func}\ tf\ \mathbf{local}\ t^*\ e^*) \\
s; (\mathbf{i32.const}\ j)\ \mathbf{call_indirect}\ tf & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
v^n\ (\mathbf{call}\ cl) & \hookrightarrow & \mathbf{local}_m\{cl_{\mathsf{inst}}; v^n\ (t.\mathbf{const}\ 0)^k\}\ \mathbf{block}\ (\epsilon \to t_2^m)\ e^*\ \mathbf{end}\ \mathbf{end}\ \ \dots & \\
\mathbf{local}_n\{i; v_l^*\}\ v^n\ \mathbf{end} & \hookrightarrow & v^n & \mid\ \ \dots \text{if } cl_{\mathsf{code}} = (\mathbf{func}\ (t_1^n \to t_2^m)\ \mathbf{local}\ t^k\ e^*) \\
\mathbf{local}_n\{i; v_l^*\}\ \mathbf{trap}\ \mathbf{end} & \hookrightarrow & \mathbf{trap} & \\
\mathbf{local}_n\{i; v_l^*\}\ L^k[v^n\ \mathbf{return}]\ \mathbf{end} & \hookrightarrow & v^n & \\
v_1^j\ v\ v_2^k; \mathbf{get_local}\ j & \hookrightarrow & v & \\
v_1^j\ v\ v_2^k; v'\ (\mathbf{set_local}\ j) & \hookrightarrow & v_1^j\ v'\ v_2^k; \epsilon & \\
v\ (\mathbf{tee_local}\ j) & \hookrightarrow & v\ v\ (\mathbf{set_local}\ j) & \\
s; \mathbf{get_global}\ j & \hookrightarrow_i & s_{\mathsf{glob}}(i, j) & \\
s; v\ (\mathbf{set_global}\ j) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \mathsf{glob}(i,j) = v \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ a\ o) & \hookrightarrow_i & t.\mathbf{const}\ \mathrm{const}_t(b^*) & \text{if } s_{\mathsf{mem}}(i, k+o, |t|) = b^* \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ tp_sx\ a\ o) & \hookrightarrow_i & t.\mathbf{const}\ \mathrm{const}_t^{sx}(b^*) & \text{if } s_{\mathsf{mem}}(i, k+o, |tp|) = b^* \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{load}\ tp_sx^?\ a\ o) & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ a\ o) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \mathsf{mem}(i, k+o, |t|) = \mathrm{bits}_t^{|t|}(c) \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ tp\ a\ o) & \hookrightarrow_i & s'; \epsilon & \text{if } s' = s \text{ with } \mathsf{mem}(i, k+o, |tp|) = \mathrm{bits}_t^{|tp|}(c) \\
s; (\mathbf{i32.const}\ k)\ (t.\mathbf{const}\ c)\ (t.\mathbf{store}\ tp^?\ a\ o) & \hookrightarrow_i & \mathbf{trap} & \text{otherwise} \\
s; \mathbf{current_memory}] & \hookrightarrow_i & \mathbf{i32.const}\ |s_{\mathsf{mem}}(i,*)|/64\,\mathrm{Ki} & \\
s; (\mathbf{i32.const}\ k)\ \mathbf{grow_memory} & \hookrightarrow_i & s'; \mathbf{i32.const}\ |s_{\mathsf{mem}}(i,*)|/64\,\mathrm{Ki} & \text{if } s' = s \text{ with } \mathsf{mem}(i,*) = s_{\mathsf{mem}}(i,*)\ (0)^{k\cdot 64\,\mathrm{Ki}} \\
s; (\mathbf{i32.const}\ k)\ \mathbf{grow_memory} & \hookrightarrow_i & \mathbf{i32.const}\ (-1) &
\end{array}
$$

Figure 2. Small-step reduction rules

Different degrees of precision

Mechanized formal semantics in a proof assistant



```
(** One step of execution *)

Inductive step: state -> trace -> state -> Prop :=

  | step_skip_seq: forall f s k sp e m,
      step (State f Sskip (Kseq s k) sp e m)
        E0 (State f s k sp e m)
  | step_skip_block: forall f k sp e m,
      step (State f Sskip (Kblock k) sp e m)
        E0 (State f Sskip k sp e m)
  | step_skip_call: forall f k sp e m m',
      is_call_cont k ->
      Mem.free m sp 0 f.(fn_stackspace) = Some m' ->
      step (State f Sskip k (Vptr sp Ptrofs.zero) e m)
        E0 (Returnstate Vundef k m')

  | step_assign: forall f id a k sp e m v,
      eval_expr sp e m a v ->
      step (State f (Sassign id a) k sp e m)
        E0 (State f Sskip k sp (PTree.set id v e) m)

  | step_store: forall f chunk addr a k sp e m vaddr v m',
      eval_expr sp e m addr vaddr ->
      eval_expr sp e m a v ->
      Mem.storev chunk m vaddr v = Some m' ->
      step (State f (Sstore chunk addr a) k sp e m)
        E0 (State f Sskip k sp e m')

  | step_call: forall f optid sig a bl k sp e m vf vargs fd,
      eval_expr sp e m a vf ->
      eval_exprlist sp e m bl vargs ->
      Genv.find_funct ge vf = Some fd ->
      funsig fd = sig ->
      step (State f (Scall optid sig a bl) k sp e m)
```

Different kinds of semantics

Let us first define an abstract syntax for our language, via what is usually referred as **Backus–Naur form** (BNF).

Example : arithmetic expressions, $x \in V$ a set of variables

$$e ::= x \mid n \mid e + e \mid e * e \mid \ \ldots$$

This is just another view of the AST obtained after parsing.

Different kinds of semantics

Let us first define an abstract syntax for our language, via what is usually referred as **Backus–Naur form** (BNF).

Example : arithmetic expressions, $x \in V$ a set of variables

$$e ::= x \mid n \mid e + e \mid e * e \mid \ \dots$$

This is just another view of the AST obtained after parsing.

This abstract syntax typically forms the basis to define the semantics.

Semantics comes in different shapes:

- axiomatic
- denotational
- by translation
- **operational semantics (natural, structural)**

Axiomatic Semantics (ex: Floyd-Hoare logic)

(*An axiomatic basis for computer programming*, Hoare, 1969)

Axiomatic Semantics (ex: Floyd-Hoare logic)

(*An axiomatic basis for computer programming*, Hoare, 1969)

Hoare triples states invariants of the global state:

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

"if P is true before the instruction i, then Q is true afterwards"

Example of a valid triple:

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

Proved by application of the rule for assignment:

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

- A semantics of specifications.
- See also: separation logic

Denotational Semantics

Associates to an expression e its mathematical meaning $[\![e]\!]$ in a semantic domain $\mathcal{D}$.

Example : arithmetic expressions.

$$e ::= x \mid n \mid e + e \mid e * e \mid \ \ldots$$

For such a simple language, a simple domain does the job: $\mathcal{D} = \texttt{env} \to \mathbb{N}$.

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

Beyond arithmetic expressions, things get more involved: in what domain should we interpret the lambda calculus?

Semantics by translation

(*Definitional interpreters for higher-order programming languages*, Reynolds, 1972)

We can define the semantics of a language by translation into a language whose semantics is already known.

$$[\![x = v + v']\!] \;=\; \begin{array}{l} y = \textsf{get}\ v; \\ z = \textsf{get}\ v'; \\ x = y + z \end{array}$$

- ▶ Inherit for free the meta-theory from the host language.
- ▶ Not always very illuminating: to the extreme, R's language is defined by the result of its compiler...

Operational Semantics

We describe a process of evaluation for the computations. The approach is more syntactic: it operates directly on the abstract syntax.

- "natural" or "*big-steps semantics*", evaluates the program in one step (a derivation tree)

$$e \Downarrow v$$

- "by reduction" or "*small-steps semantics*": a relation describes an atomic reduction, and the semantics consider the transitive reflexive closure of this relation.

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

Note: although operational by nature, does not need be executable.

1. Semantics: On the Meaning of Programs
2. Operational semantics for mini-while
3. Comparing the different semantics

mini-while

$$e \in \mathcal{A} ::= x \mid n \mid e + e \mid e * e \mid \ldots$$

(abstract) grammar:

$$\begin{array}{rcll} S(Smt) & ::= & x := e & \text{assign} \\ & \mid & skip & \text{do nothing} \\ & \mid & S_1 ; S_2 & \text{sequence} \\ & \mid & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\ & \mid & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{loop} \end{array}$$

Semantics of expressions

We consider a very simple memory model: a store

$\sigma \in State = Var \to \mathbf{Z}$.

Access is written $\sigma(x)$, and update $\sigma[y \mapsto n]$.

Semantics of arithmetic expressions – Val: $\mathcal{A} \to State \to \mathbf{Z}$: **On board**

$$\begin{aligned} Val(n, \sigma) &= \mathcal{N}(n) \\ Val(x, \sigma) &= \\ Val(e + e', \sigma) &= \\ Val(e \times e', \sigma) &= \end{aligned}$$

Note: Denotational or natural semantics?

Semantics of boolean expressions

$Val : \mathcal{B} \rightarrow State \rightarrow \mathbf{Z}$ **Exercise at the end of course**

$(b ::= tt \mid ff \mid x \mid b \wedge b \mid ... \mid e < e \mid ...)$

Warm up: first properties

Semantics of arithmetic expressions

Show the two following properties (first one at the end of the course):

1. For any $e \in \mathcal{A}$, and σ, σ' two states. Show that if $(\forall x \in Vars(e), \sigma(x) = \sigma'(x))$, then $Val(e, \sigma) = Val(e, \sigma')$.
   Exercise at the end of course
2. Let $e, e' \in \mathcal{A}$, show that:

$$Val(e[e'/x], \sigma) = Val(e, \sigma[x \mapsto Val(e', \sigma)])$$

now

Natural semantics (big step) for mini-while 1/2

In one step from the source program to the final result.

$\Downarrow$: $Stm \times State \to State$

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

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

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

Natural semantics (big step) for mini-while 2/2

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

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

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

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

Natural semantics (big step) for mini-while 2/2

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

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

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

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

Example

Derive the semantics (leaves are axioms, nodes are rules) of:

- $x := 2; \texttt{while}\ x > 0\ \texttt{do}\ x := x - 1\ \texttt{done}$
- $x := 2; \texttt{while}\ x > 0\ \texttt{do}\ x := x + 1\ \texttt{done}$

Using the semantics to prove properties

Example: determinism

In mini-while there is a single way to evaluate a program.

Theorem: Determinism

For all S, for all $\sigma, \sigma', \sigma''$:

If $(S, \sigma) \Downarrow \sigma'$ and $(S, \sigma) \Downarrow \sigma''$ then $\sigma' = \sigma''$.

What should we induct on? **do the proof**

Structural Op. Semantics (SOS = small step) for mini-while 1/2

(*A Structural Approach to Operational Semantics*, Plotkin, late 70th)

We perform atomic reduction steps.

$\rightarrow$: $Stm \times State \rightarrow Stm \times State$

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

$$(\texttt{skip}, \sigma) \not\rightarrow$$

$$\frac{}{((\texttt{skip}; S_2), \sigma) \rightarrow (S_2, \sigma)} \quad \frac{(S_1, \sigma) \rightarrow (S_1', \sigma')}{((S_1; S_2), \sigma) \rightarrow (S_1'; S_2, \sigma')}$$

Structural Op. Semantics (SOS = small step) for mini-while 2/2

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

$$\frac{Val(b,\sigma) = f\!f}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \rightarrow (S_2, \sigma)}$$

$$(\texttt{while}\, b\, \texttt{do}\, S\, \texttt{done}, \sigma) \rightarrow$$

Structural Op. Semantics (SOS = small step) for mini-while 2/2

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

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

$$(\texttt{while}\, b\, \texttt{do}\, S\, \texttt{done}, \sigma) \rightarrow$$

$$(\texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\, b\, \texttt{do}\, S\, \texttt{done})\ \texttt{else}\ \texttt{skip}, \sigma)$$

Structural Op. Semantics (SOS = small step) for mini-while 2/2

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

$$\frac{Val(b,\sigma) = f\!f}{(\texttt{if}\ \ b\ \ \texttt{then}\ \ S_1\ \ \texttt{else}\ \ S_2, \sigma) \to (S_2, \sigma)}$$

$$(\texttt{while}\, b\, \texttt{do}\, S\, \texttt{done}, \sigma) \to$$

$$(\texttt{if}\ \ b\ \ \texttt{then}\ \ (S; \texttt{while}\, b\, \texttt{do}\, S\, \texttt{done})\ \ \texttt{else}\ \ \texttt{skip}, \sigma)$$

We write $(c, \sigma) \to^* \sigma'$ if $(c, \sigma) \to^* (\texttt{skip}, \sigma')$.

Exercises

Derive the small-step semantics of:

- $x := 2; \texttt{while}\ x > 0\ \texttt{do}\ x := x - 1\ \texttt{done}$
- $x := 2; \texttt{while}\ x > 0\ \texttt{do}\ x := x + 1\ \texttt{done}$

How to prove determinism for the SOS semantics? What is the structure of the proof? **do the proof**

1. Semantics: On the Meaning of Programs

2. Operational semantics for mini-while

3. Comparing the different semantics

Comparison: divergence

A program is said to diverge if its execution does not terminate (slightly ambiguous in presence of non-determinism). A formal meaning of this statement is tied to the semantics we consider. In mini-while, a program diverges in state σ iff:

- NAT: the pair (S, σ) admits no derivation for any σ'.
- SOS: the pair (S, σ) admits an infinite sequence of derivations.

Note:

▶ Assuming the existence of a derivation in NAT restricts the quantification to terminating programs.

▶ What if the language can get stuck?

Comparison: equivalence of programs

A central purpose of semantics is program equivalence.

Two mini-while programs S_1, S_2 are semantically equivalent if:

- NAT: $\forall \sigma, \sigma', (S_1, \sigma) \Downarrow \sigma'$ iff $(S_2, \sigma) \Downarrow \sigma'$
- SOS: $\forall \sigma$:
  - $(S_1, \sigma) \rightarrow^* \sigma'$ iff $(S_2, \sigma) \rightarrow^* \sigma'$
  - (S_1, σ) diverges iff (S_2, σ) diverges

Are the two semantics equivalent?

$$\mathcal{S}_{NS}[S]\sigma = \begin{cases} \sigma' & \text{If } (S,\sigma) \Downarrow \sigma' \\ undef & \text{else} \end{cases}$$

$$\mathcal{S}_{SOS}[S]\sigma = \begin{cases} \sigma' & \text{If } (S,\sigma) \rightarrow^{*} \sigma' \\ undef & \text{else} \end{cases}$$

Theorem

$$\mathcal{S}_{NS} = \mathcal{S}_{SOS}$$

Equivalence of semantics 1/2

Proposition

If $(S, \sigma) \Downarrow \sigma'$ then $(S, \sigma) \rightarrow^* \sigma'$.

Auxiliary lemma

If $(S_1, \sigma) \rightarrow^k \sigma'$ then $((S_1; S_2), \sigma) \rightarrow^k (S_2, \sigma')$

Proof: structural induction on the derivation tree for $(S, \sigma) \Downarrow$.

Equivalence of semantics 2/2

Proposition

If $(S, \sigma) \rightarrow^k \sigma'$ then $(S, \sigma) \Downarrow \sigma'$.

Auxiliary lemma

If $(S_1; S_2, \sigma) \rightarrow^k \sigma''$ then there exists σ', k_1 such that $(S_1, \sigma) \rightarrow^{k_1} \sigma'$ and $(S_2, \sigma') \rightarrow^{k-k_1} \sigma''$

Proof: induction on k.

Expressing parallelism

SOS can very naturally capture parallel execution as an interleaving.
For instance, for the parallel execution of two commands with no dynamic creation of threads:

$$\frac{(S_1,\sigma) \rightarrow (S_1',\sigma')}{\big((S_1||S_2),\sigma\big) \rightarrow (S_1'||S_2,\sigma')} \quad \frac{(S_2,\sigma) \rightarrow (S_2',\sigma')}{\big((S_1||S_2),\sigma\big) \rightarrow (S_1||S_2',\sigma')}$$

We will come back to parallelism later in this course.
Notice that expressing the same notion in NAT is not as straightforwards at all.

Correct compilation 1/3

What should we expect from a compiler?

It should preserves the meaning of programs.

$$\mathcal{T} : \mathcal{L}_1 \to \mathcal{L}_2$$

Correctness of $\mathcal{T}$

$$\forall p \in \mathcal{L}_1,\ [\![p]\!]_1 \equiv [\![\mathcal{T}(p)]\!]_2$$

Correct compilation 1/3

What should we expect from a compiler?

It should preserves the meaning of programs.

$$\mathcal{T} : \mathcal{L}_1 \rightarrow \mathcal{L}_2$$

Correctness of $\mathcal{T}$

$$\forall p \in \mathcal{L}_1,\ \llbracket p \rrbracket_1 \supseteq \llbracket \mathcal{T}(p) \rrbracket_2$$

Correct compilation 2/3

Terminating commands for `Mini_while` transformation

$$\mathcal{T} : \texttt{Mini\_while} \to \texttt{Mini\_while}$$

Correctness of $\mathcal{T}$

$$\forall c, \sigma, \sigma',\ (c, \sigma) \Downarrow \sigma' \to (\mathcal{T}(c), \sigma) \Downarrow \sigma'$$

Note:

- Induction on the source derivation gives us a very strong proof principle
- $\mathcal{T}(\texttt{while true do skip}) = \texttt{skip}$ is possible!

Correct compilation 3/3

But what of diverging commands?

For `Mini_while`, not very useful, but crucial when compiling a server, or a reactive program.

- We move to SOS and simulation diagrams. See on board.

Mini-while is not exactly mini-C

variable initialisation!

- **variable declarations**
  - Main problem is the scope of variables (x may not refer to the same variable depending on the point in the program)
  - See course on typing
- Expression **evaluation**
  Here we only had expressions without side-effects.
- **print-int and print-string** (operational semantics not so interesting, but introduces traces)
- Mini-C will have **functions.** We tackle them later on in this course.

Conclusion

Core ideas discussed today:

- Different flavors of semantics: focus on operational semantics
- Two sub-flavors: discussion on the difference between NAT and SOS
- Semantics as the basis to specify properties of programs and languages
- Reasoning by induction on the derivation, on the length of the reduction, by simulation diagrams

Next course: typing!

Additional exercise: make sure adding a construct such as **repeat** to the semantics is clear to you.