Compilation and Program Analysis (#2a): Semantics

Ludovic Henrio

Master 1, ENS de Lyon et Dpt Info, Lyon1

2025-2026

Intro

Contact me:
web: lhenrio.github.io
email: ludovic.henrio@ens-lyon.fr

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

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**

Semantics

We will first define an abstract syntax for our language.

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.

On the abstract syntax we will define one or several semantics.

Different kinds of semantics:

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

Axiomatic Semantics (Hoare logic)

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

Characterisation by properties on variables, using triples of the form:

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

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

Example :

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

Example of generating rule:

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

- proving properties of programs.

Denotational Semantics

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

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

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

You must choose a domain for the mathematical meaning with adequate operations.
Trivial example for expressions with $\mathcal{D} = \texttt{env} \to \mathbb{N}$.

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

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.

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

- Inherit for free the meta-theory from the host language.
- Not always very illuminating in terms of behaviour
- ... but sometimes a good specification in terms of implementation or compilation.

Operational Semantics

Describes the computation as an evaluation from the program to its computed value. Operates directly on the abstract syntax. 2 kinds:

- "natural" or "*big-steps semantics*", evaluates the program in one step
$$e \Downarrow v$$
- "by reduction" or "*small-steps semantics*", repeat the evaluation until a result is obtained:
$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$
A relation describes an atomic reduction, and the semantics consider the transitive (reflexive) closure of this relation.

Results do not need to be a value.

Note: different notations (arrows) exist: $\Downarrow$ / $\Rightarrow$ / ... $\vdash$ ... $\rightarrow$ ...

▶ language specification and proving properties of languages.

mini-while

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

(abstract) grammar:

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

Semantics of expressions

We denote $State = Var \to \mathbf{Z}$.

This kind of state is sometimes called "store". We denote them by σ.

Access is denoted $\sigma(x)$. Update is denoted by $\sigma[y \mapsto n]$.

Semantics of arithmetic expressions – Val: $\mathcal{A} \to State \to \mathbf{Z}$ (in each state an integer value): **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: what kind of semantics is this? big step? denotational?

Semantics of boolean expressions

$Val : \mathcal{B} \rightarrow State \rightarrow \mathbf{Z}$ **Not now**

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

First properties and exercise

Semantics of arithmetic expressions

Substitution is denoted $e[e'/x]$.

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

1. Let $e \in \mathcal{A}$ a given arithmetic expression. Let σ, σ' be two states. Show that if $(\forall x \in Vars(e), \sigma(x) = \sigma'(x))$, then $Val(e, \sigma) = Val(e, \sigma')$. **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

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

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

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

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

Example

Compute 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 $\sigma' = \sigma''$.
- If $(S, \sigma) \Downarrow \sigma'$, there is no infinite derivation.

The Proof is by induction on the structure of the derivation tree.

WE do a proof sketch

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

Evaluating one statement at a time.
$\Rightarrow$: $Stm \times State \rightarrow Stm \times State$ OR $Stm \times State \rightarrow State$ (we could have a **done** statement to avoid the two cases).

$$(x := e, \sigma) \rightarrow \sigma[x \mapsto Val(e, \sigma)] \qquad (\texttt{skip}, \sigma) \rightarrow \sigma$$

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

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

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

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

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

Exercises

Compute the 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**

Comparison: divergence

In general a program diverges if it runs forever.

In mini-while, a program diverges in state σ iff:

- NAT: no successor to (S, σ).
- SOS: infinite sequence begining with (S, σ).

In other languages/semantics there might be other reasons to have no successor (see later in course), and you could have no successor in the SOS without reaching a final state.

Comparison: equivalence of programs

Semantics is also useful for defining program equivalence, in mini-while it is quite simple:
Two mini-while programs S_1 and S_2 are semantically equivalent if:

- NAT: $\forall \sigma, \sigma', (S_1, \sigma) \Downarrow \sigma'$ iff $(S_2, \sigma) \Downarrow \sigma'$
- SOS: $\forall \sigma$:
  - for all config (blocking or not): $(S_1, \sigma) \rightarrow^* \sigma'$ 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

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

Proof: see next slides ...

Equivalence of semantics 1/2

Proposition

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

Proof relies on:

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 $(S, \sigma) \Downarrow \sigma'$.

Proof relies on:

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 express interleaving, NAT cannot:

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

... more later in the course.

Mini-while is not exactly mini-C

variable initialisation!

- **variable declarations**
  - Main problem is scope of variables (x may not refer to the same variable depending on the point in the program)
  - see course on typing
- Expression **evaluation**
  restricted to expressions without side-effect, the val function has to be encoded as a set of instructions (a more precise semantics would define several reduction steps)
- **print-int and print-string** (operational semantics not much interesting)
- Mini-C will have **functions** ... defined later in the course

Conclusion

We have seen different kinds of semantics and compared them briefly.

We have shown how to define operational semantics.

- For expression evaluation
- On mini-while

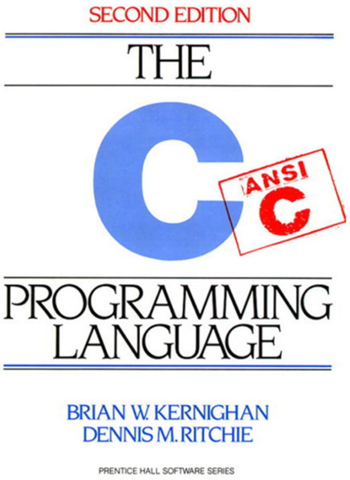
And how to reason on them to derive language properties (or at least properties of the semantics).

Next course on typing will illustrate more properties.

Additional exercise: **repeat**.

Final words: Different degrees of precision

Semi-formal specification in natural language



Final words: Different degrees of precision

Formal semantics

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

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

$$
\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}\ \ldots & \\
\mathbf{local}_n\{i; v_l^*\}\ v^n\ \mathbf{end} & \hookrightarrow & v^n & \mid\ \ldots \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

Final words: 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)
```