18. Operational Semantics

Goal: · Define operational semantics for while-programs
· Two styles:
Small step semantics specify the effect of every operation
Big step semantics capture the effect of the overall program execution.

Technically: Structural operational semantics (SOS, Gordon Plotkin 1981)
· The idea of SOS is that the configurations of a program have a syntactic structure.
↳ They compose basic commands via a set of operations.
· This means we can use proof systems (calculi) to define the behavior of configurations:
↳ A transition exists iff it can be derived in the proof system.
· Technically, the proof system implements an induction on the structure of configurations:
↳ Axioms define the transitions of basic commands
↳ Proof rules define the transitions of composed configurations via the transitions of the operands.

Benefits: · Simplicity and elegance
· Being able to establish properties of transitions via induction along the derivation.

Recapitulation (Syntax of while-programs):
$a ::= k \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2$
$b ::= true \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2$
$c ::= skip \mid x := a \mid c_1 ; c_2 \mid$ if b then c_1 else c_2 fi $\mid$ while b do c od,
with $k \in \mathbb{Z}$, $x \in Vars$, $true \in \mathbb{B}$.

- We use $Prog$ to refer to the set of all while-programs.
- Let $Sig = (Fun, Pred)$ with $Fun = \{+_2, -_2, *_2\} \cup \{k_0 \mid k \in \mathbb{Z}\}$ and $Pred = \{>_2\}$ be the (logical) <u>signature</u> of our programming language.

- The semantics assign to each syntactic expression a meaning. Formally, a meaning is an element from a <u>semantic domain</u>.
- A semantic domain is given as a (logical) <u>Sig-structure</u>

$$\mathcal{S} = (D, \mathcal{I})$$

with
- <u>data domain</u> $D \neq \emptyset$, a set of elements, and
- <u>interpretation</u> $\mathcal{I}$ that assigns to each function symbol $f_n \in Fun$ an <u>actual function</u>

$$\mathcal{I}(f): D^n \to D$$

and to each predicate symbol $p_n \in Pred$ an <u>actual predicate</u>

$$\mathcal{I}(p): D^n \to \mathbb{B}.$$

- For the while-programs with signature Sig defined above, we use $\mathcal{S} = (\mathbb{Z}, \mathcal{I})$ with $\mathcal{I}(k)$, $\mathcal{I}(+)$, $\mathcal{I}(-)$, $\mathcal{I}(*)$, and $\mathcal{I}(>)$ as expected.

- The behavior of programs depends on the <u>valuation of variables</u>, also called <u>state</u>:

$$\sigma \in State := \mathbb{Z}^{Vars}.$$

- Given a state σ, the semantics of arithmetic (a) and Boolean (b) expressions is defined as $\mathcal{S}[\![\begin{smallmatrix} a \\ b \end{smallmatrix}]\!](\sigma)$.

Like in Logic above we called the as <u>terms</u> and the bs <u>formulas</u>.

18.1 Small-Step Operational Semantics

Goal: Specify the operation of a program one command at a time.

Technically:
- There is a set of proof rules that we continue to apply to configurations (c, σ) until reaching a final configuration $(skip, \sigma)$.
- We write $(c,\sigma) \to (c',\sigma')$ to indicate that (c,σ) reduces to (c',σ') in one step.
  We use $(c,\sigma) \to^* (c',\sigma')$ to indicate that (c,σ) reduces to (c',σ') in zero or more steps:
  There are $(c_0,\sigma_0) \dots (c_k,\sigma_k)$ with $(c,\sigma) = (c_0,\sigma_0)$, $(c_k,\sigma_k) = (c',\sigma')$ and $(c_i,\sigma_i) \to (c_{i+1},\sigma_{i+1})$ for all $0 \le i < k$.

Definition (Small-step transition relation among configurations):
- A configuration is a pair $(c,\sigma) \in Prog \times State$ consisting of a program $c \in Prog$ and a state $\sigma \in State$.
- The small-step transition relation among configurations $\to \subseteq (Prog \times State) \times (Prog \times State)$ is the smallest relation satisfying the following rules:

$$\frac{}{(x := a, \sigma) \to (skip, \sigma[x := \mathcal{A}[\![a]\!](\sigma)])} \quad (assign)$$

$$\frac{(c_0,\sigma) \to (c_0',\sigma')}{(c_0;c_1,\sigma) \to (c_0';c_1,\sigma')} \quad (seq1) \qquad \frac{}{(skip;c_1,\sigma) \to (c_1,\sigma)} \quad (seq2)$$

$$\frac{}{(\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi},\sigma) \to (c_1,\sigma)} \text{ if } \mathcal{B}[\![b]\!](\sigma) = true \quad (iftrue)$$

$$\frac{}{(\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi},\sigma) \to (c_2,\sigma)} \text{ if } \mathcal{B}[\![b]\!](\sigma) = false \quad (iffalse)$$

$$\frac{}{(\text{while } b \text{ do } c \text{ od},\sigma) \to (\text{if } b \text{ then } (c; \text{while } b \text{ do } c \text{ od}) \text{ else } skip \text{ fi},\sigma)} \quad (while)$$

Note:

- The definition of the transition relation relies on rule schemata of the form

$$\frac{\text{Premise}}{\text{Conclusion}}.$$

If the premise can be instantiated, we can draw the corresponding instantiation of the conclusion.

- Rules without premise are called axioms.
- Repeated applications of the rules yield a derivation tree:

↳ axioms = leaves

↳ transition for the program + state of interest = root

Example:

Consider $c :=$

```
x := x+1;
while x>0 do
    y := y+1;
od
```

$\} =: c_1$

We have $(x := x+1, (x=0, y=0)) \to (skip, (x=1, y=0))$ by (assign).

Hence, by (seq 1):

$(x := x+1; c_1, (x=0, y=0)) \to (skip; c_1, (x=1, y=0))$.

We apply (seq 2) and obtain:

$(skip; c_1, (x=1, y=0)) \to (c_1, (x=1, y=0))$

Since c_1 is a while-program, we unwind it into

$c_2 :=$

```
if x>0 then
    y := y+1;
    c1
else
    skip
fi
```

Formally, with Rule (while) we obtain the transition

$$(c_1, (x=1, y=0)) \longrightarrow (c_2, (x=1, y=0)).$$

We have $[\![x>0]\!](x=1, y=0) = \text{true}$,

hence (iftrue) applies:

$$(c_2, (x=1 y=0)) \longrightarrow (y := y+1; c_1, (x=1, y=0)).$$

To derive a transition for the configuration $(y := y+1; c_1, (x=1, y=0))$, we first note that by (assign):

$$(y := y+1, (x=1, y=0)) \longrightarrow (\text{skip}, (x=1, y=1)).$$

Hence, with (seq1):

$$(y := y+1; c_1, (x=1, y=0)) \longrightarrow (\text{skip}; c_1, (x=1, y=1)).$$

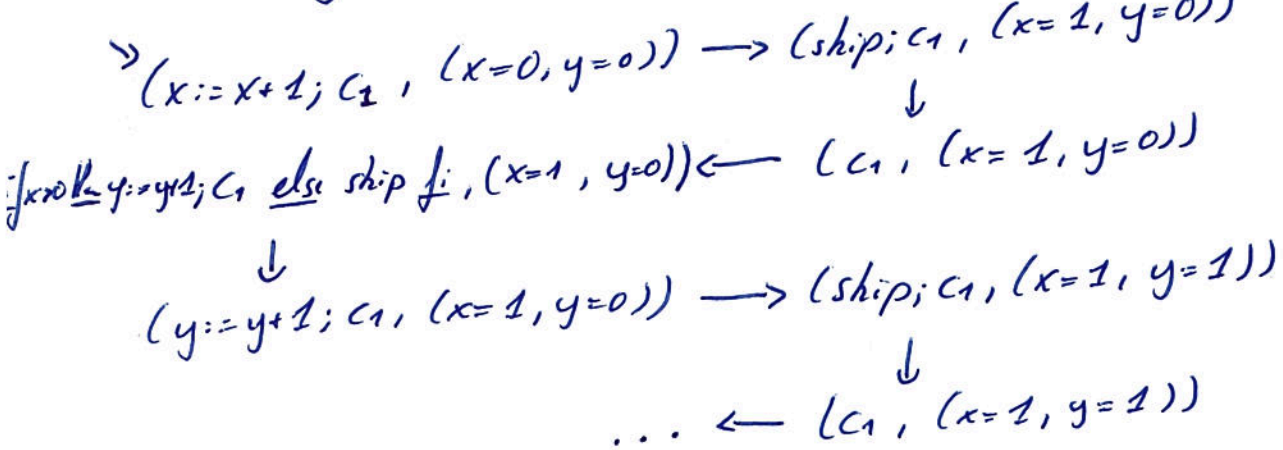
An application of (seq2) yields

$$(\text{skip}; c_1, (x=1, y=1)) \longrightarrow (c_1, (x=1, y=1)).$$

...

- Often, one represents the resulting transitions as a <u>transition system</u> $(\text{Prog} \times \text{State}, \longrightarrow, \text{init})$, where $\text{init} = (c_0, \sigma_0)$ is the initial configuration one starts from. The transition system is commonly restricted to the reachable configurations.

In our running example, this yields



The small step operational semantics is deterministic,
which means there is at most one successor configuration
for each configuration.

Lemma (Determinism):
If $(c, \sigma) \to (c', \sigma')$ and $(c, \sigma) \to (c'', \sigma'')$,
then $c' = c''$ and $\sigma' = \sigma''$.

For a composed program $c_1; c_2$, every computation

$(c_1; c_2, \sigma) \longrightarrow (skip, \sigma')$

can be decomposed into a computation of c_1
followed by a computation of c_2:

$(c_1, \sigma) \longrightarrow (skip, \sigma'') \longrightarrow (skip, \sigma')$
(c_2, σ'')

Lemma ((De)composing computations):

$(c_1; c_2, \sigma) \to^n (skip, \sigma')$ iff

$\exists \sigma'' \in State, n_1, n_2 \in \mathbb{N}$: $(c_1, \sigma) \to^{n_1} (skip, \sigma'')$
and $(c_2, \sigma'') \to^{n_2} (skip, \sigma')$
and $n = n_1 + n_2 + 1$.

18.2 Big-Step Operational Semantics

Goal:
- The small-step operational semantics specifies the operations of the program one step at a time.
- We now define a **big-step operational semantics** that specifies the entire computation from an initial to a final configuration.

Definition

The *big-step transition relation* $\Downarrow \subseteq (Prog \times Stack) \times Stack$ is the smallest relation satisfying the following rules:

$$(bs\_skip)\ \frac{}{(skip, \sigma) \Downarrow \sigma} \qquad (bs\_assign)\ \frac{}{(x := a, \sigma) \Downarrow \sigma[x := \mathcal{S}[\![a]\!](\sigma)]}$$

$$(bs\_seq)\ \frac{(c_0, \sigma) \Downarrow \sigma' \quad (c_1, \sigma') \Downarrow \sigma''}{(c_0 ; c_1, \sigma) \Downarrow \sigma''}$$

$$(bs\_if\_true)\ \frac{(c_0, \sigma) \Downarrow \sigma'}{(\underline{if}\ b\ \underline{then}\ c_0\ \underline{else}\ c_1\ \underline{fi}, \sigma) \Downarrow \sigma'}, \text{ if } \mathcal{S}[\![b]\!](\sigma) = true$$

$$(bs\_if\_false)\ \frac{(c_1, \sigma) \Downarrow \sigma'}{(\underline{if}\ b\ \underline{then}\ c_0\ \underline{else}\ c_1\ \underline{fi}, \sigma) \Downarrow \sigma'}, \text{ if } \mathcal{S}[\![b]\!](\sigma) = false$$

$$(bs\_while\_false)\ \frac{}{(\underline{while}\ b\ \underline{do}\ c\ \underline{od}, \sigma) \Downarrow \sigma}, \text{ if } \mathcal{S}[\![b]\!](\sigma) = false$$

$$(bs\_while\_true)\ \frac{(c, \sigma) \Downarrow \sigma' \quad (\underline{while}\ b\ \underline{do}\ c\ \underline{od}, \sigma') \Downarrow \sigma''}{(\underline{while}\ b\ \underline{do}\ c\ \underline{od}, \sigma) \Downarrow \sigma''}, \text{ if } \mathcal{S}[\![b]\!](\sigma) = true.$$

Remark:

The big-step semantics *cannot* talk about non-termination:
If (c, σ) does not terminate, *there is no* σ' with $(c, \sigma) \Downarrow \sigma'$.

Both, $\rightarrow^*$ and $\Downarrow$ capture the effect of a full execution of the program.
So for *terminating programs*, we expect that

- if (c, σ) leads to $(skip, \sigma')$ via a small-step computation,
  then σ' should be the result of a big-step evaluation;
- in turn, big steps should be justified by small-step computations.

This indeed holds.

Theorem (Correspondence of small-step and big-step semantics):

$(c,\sigma) \to^* (skip,\sigma') \iff (c,\sigma) \Downarrow \sigma'$.

Proof:

„⇒" We proceed by Noetherian induction on the length n of the computation and show that

$\forall c \in Prog\ \forall \sigma,\sigma' \in State: (c,\sigma) \to^n (skip,\sigma') \Rightarrow (c,\sigma) \Downarrow \sigma'$.

Base case $n \leq 1$:

- The only program that leads to skip in zero steps is skip itself:
  $(skip,\sigma) \to^0 (skip,\sigma)$.
  In the big-step semantics, $(skip,\sigma) \Downarrow \sigma$ by Axiom (bs skip).
- The only program that leads to skip in one step is $x := a$:
  $(x := a,\sigma) \to (skip,\sigma')$ with $\sigma' := \sigma[x := \mathcal{A}[\![a]\!](\sigma)]$.
  In the big-step semantics, $(x := a,\sigma) \Downarrow \sigma'$ by Axiom (bs assign).

Induction step: Assume

$(c,\sigma) \to^m (skip,\sigma')$ implies $(c,\sigma) \Downarrow \sigma'$

for all $m \leq n$ and

for all $c \in Prog$, $\sigma,\sigma' \in State$.

We again have to establish the universally quantified claim, and therefore consider arbitrary $c_{init} \in Prog$, $\sigma_{init}, \sigma_{fin} \in State$ with

$(c_{init},\sigma_{init}) \to^{n+1} (skip,\sigma_{fin})$.

To establish $(c_{init},\sigma_{init}) \Downarrow \sigma_{fin}$, we do a case distinction along the possible rules that were used to derive the first transition in the computation (the one from (c_{init},σ_{init})).

Case (seq 1):

If the first transition is derived with (seq 1),
the computation takes the shape:

$$(c_{init}, \sigma_{init}) = (c_0; c_1, \sigma_{init}) \to (c_0'; c_1, \sigma_1) \to^n (skip, \sigma_{final}).$$

The decomposition lemma for small-step computations above
now yields

$$\exists \sigma' \in State, n_1, n_2 \in \mathbb{N}:$$
$$(c_0, \sigma_{init}) \to^{n_1} (skip, \sigma') \text{ and } (c_1, \sigma') \to^{n_2} (skip, \sigma_{final})$$
$$\text{and } n_1 + n_2 + 1 = n + 1.$$

Since $n_1 \leq n$ and $n_2 \leq n$, we can apply the induction hypothesis
and obtain:

$$(c_0, \sigma_{init}) \Downarrow \sigma' \text{ and } (c_1, \sigma') \Downarrow \sigma_{final}.$$

An application of Rule (bsseq) yields

$$(\underbrace{c_0; c_1}_{c_{init}}, \sigma_{init}) \Downarrow \sigma_{final}.$$

Case (seq 2):

Along similar lines but does not need the decomposition lemma.

Case (if true):

If the first transition is derived with (if true),
the computation takes the shape:

$$(c_{init}, \sigma_{init}) = (\underline{\text{if}}\ b\ \underline{\text{then}}\ c_1\ \underline{\text{else}}\ c_0\ \underline{\text{fi}}, \sigma_{init}) \to (c_1, \sigma_{init}) \to^n (skip, \sigma_{final})$$

with $[\![b]\!](\sigma_{init}) = true$.

By the induction hypothesis,

$$(c_1, \sigma_{init}) \Downarrow \sigma_{final}.$$

Since we know that b evaluates to true,
we can apply Rule (bs_IfTrue) for the big-step semantics:

$$(\underbrace{\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}}_{c_{init}}, \sigma_{init}) \Downarrow \sigma_{final}.$$

Case (WhileFalse):

Similar.

Case (While):

Assume the first transition is derived with (While):

$$(c_{init}, \sigma_{init}) \to (c_1, \sigma_{init}) \to^n (skip, \sigma_{final})$$

with

$$c_{init} = \text{while } b \text{ do } c \text{ od}$$
$$c_1 := \text{if } b \text{ then } (c; c_{init}) \text{ else skip fi}.$$

There are two cases: The if evaluates to true or to false.
We consider the more complicated case that $\mathcal{S}[\![b]\!](\sigma_{init}) = true$.
Now the computation is

$$(c_{init}, \sigma_{init}) \to (c_1, \sigma_{init}) \to (c; c_{init}, \sigma_{init}) \to^{n-1} (skip, \sigma_{final}).$$

The decomposition lemma for small-steps computations gives

$$\exists \sigma' \in State, n_1, n_2 \in \mathbb{N}: \quad (c, \sigma_{init}) \to^{n_1} (skip, \sigma')$$
$$\text{and } (c_{init}, \sigma') \to^{n_2} (skip, \sigma_{final})$$
$$\text{and } n_1 + n_2 + 1 = n - 1.$$

Since $n_1 \leq n$ and $n_2 \leq n$, we can apply the induction hypothesis and obtain

$$(c, \sigma_{init}) \Downarrow \sigma' \quad \text{and} \quad (\underbrace{\text{while } b \text{ do } c \text{ od}}_{c_{init}}, \sigma') \Downarrow \sigma_{final}.$$

An application of Rule (bs_WhileTrue) yields

$$(\underbrace{\text{while } b \text{ do } c \text{ od}}_{c_{init}}, \sigma_{init}) \Downarrow \sigma_{final}.$$

Remark:

There is an alternative proof for case (while)
that does more reasoning on the big-step semantics.

- Consider

$$(C_{init}, \sigma_{init}) \to (C_1, \sigma_{init}) \to^* (skip, \sigma_{fin})$$

with

$$C_{init} = \underline{while}\ b\ \underline{do}\ c\ \underline{od}$$

$$C_1 := \underline{if}\ b\ \underline{then}\ (c; C_{init})\ \underline{else}\ skip\ \underline{fi}.$$

- The application of the induction hypothesis yields

$$(C_1, \sigma_{init}) \Downarrow \sigma_{fin}.$$

- Since the big-step semantics is the smallest set of transitions that satisfy the given rules, the only way to derive this big step for the if-program C_1 is

$$(c; C_{init}, \sigma_{init}) \Downarrow \sigma_{fin} \quad \text{or} \quad (skip, \sigma_{init}) \Downarrow \sigma_{fin},$$

depending on whether $\mathcal{B}[\![b]\!](\sigma_{init})$ is true or false.
Say $\mathcal{B}[\![b]\!](\sigma_{init}) = true$.

- With the same argument (smallest set of transitions),

$$(c; C_{init}, \sigma_{init}) \Downarrow \sigma_{fin}$$

allows us to apply Rule (bsseq) backwards and conclude

$$(c, \sigma_{init}) \Downarrow \sigma' \quad \text{and} \quad (C_{init}, \sigma') \Downarrow \sigma_{fin} \text{ for some } \sigma' \in Stat.$$

- To these two big steps, we can now apply (bswhiletrue):

$$(\underbrace{\underline{while}\ b\ \underline{do}\ c\ \underline{od}}_{C_{init}}, \sigma_{init}) \Downarrow \sigma_{fin}.$$

$\Leftarrow$: For the implication $(c,\sigma) \Downarrow \sigma' \Rightarrow (c,\sigma) \to^* (skip,\sigma')$ for all $c \in Prog$, $\sigma,\sigma' \in State$, we proceed by Noetherian induction on the height of the proof tree.

Base case:
- Axiom $\dfrac{}{(skip,\sigma) \Downarrow \sigma}$ is mimicked by $(skip,\sigma) \to^0 (skip,\sigma)$.
- Axiom $\dfrac{}{(x:=a,\sigma) \Downarrow \sigma'}$ with $\sigma' := \sigma[x := \mathcal{S}[\![a]\!](\sigma)]$ is reflected by (assign):
$$(x:=a,\sigma) \to (skip,\sigma').$$

Induction step: Assume
$(c,\sigma) \Downarrow \sigma'$ implies $(c,\sigma) \to^* (skip,\sigma')$
for all $c \in Prog$, $\sigma,\sigma' \in State$
where $(c,\sigma) \Downarrow \sigma'$ can be derived
with a proof tree of height $m \le n$.

To establish the universally quantified claim, we consider arbitrary $c_{init} \in Prog$, $\sigma_{init}, \sigma_{final} \in State$ where
$(c_{init},\sigma_{init}) \Downarrow \sigma_{final}$ holds due to a proof of height $n+1$.

We do a case distinction along the possible rules that were used to derive the final conclusion.

Case (Sseq):
If we have
$$\frac{(c_0,\sigma_{init}) \Downarrow \sigma' \qquad (c_1,\sigma') \Downarrow \sigma_{final}}{(\underbrace{c_0;c_1}_{c_{init}},\sigma_{init}) \Downarrow \sigma_{final}}$$
of height $n+1$,
the derivations $(c_0,\sigma_{init}) \Downarrow \sigma'$ and $(c_1,\sigma') \Downarrow \sigma_{final}$ have height at most n.
By the induction hypothesis, we get
$(c_0,\sigma_{init}) \to^* (skip,\sigma')$ and $(c_1,\sigma') \to^* (skip,\sigma_{final})$.
The decomposition lemma yields $(\underbrace{c_0;c_1}_{c_{init}},\sigma_{init}) \to^* (skip,\sigma_{final})$.

Cases (bs_if_true), (bs_if_false), and (bs_while_false):

Simple.

Case (bs_while_true):

If we have $\dfrac{(c, \sigma_{init}) \Downarrow \sigma' \quad (\underbrace{\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}}_{c_{init}}, \sigma') \Downarrow \sigma_{fin} \text{ of height } \leq n}{(\underbrace{\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}}_{c_{init}}, \sigma_{init}) \Downarrow \sigma_{fin}}$

The proof trees for $(c, \sigma_{init}) \Downarrow \sigma'$ and $(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma') \Downarrow \sigma_{fin}$ have height at most n.

The induction hypothesis thus gives

$(c, \sigma_{init}) \to^* (\texttt{skip}, \sigma')$ and

$(\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma') \to^* (\texttt{skip}, \sigma_{fin})$.

We apply the decomposition lemma and obtain

$(c; \underbrace{\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}}_{c_{init}}, \sigma_{init}) \to^* (\texttt{skip}, \sigma_{fin})$. (*)

Since we applied Rule (bs_while_true), we know that $\mathcal{S}[\![b]\!](\sigma_{init}) = \text{true}$. (**)

This gives the overall small-step computation:

$$(\underbrace{\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}}_{c_{init}}, \sigma_{init}) \to (\texttt{if}\ b\ \texttt{then}\ (c; c_{init})\ \texttt{else skip fi}, \sigma_{init})$$
$$\xrightarrow{\text{by (**)}} (c; c_{init}, \sigma_{init})$$
$$\xrightarrow{\text{by (*)}}{}^* (\texttt{skip}, \sigma_{fin}).$$

□

Benefits and drawbacks – a comparison:

Small-step semantics:
- Easy to model complex features of the programming language:
  - concurrency, heaps, divergence (non-terminating computations), stacks, runtime errors, exceptions.
- Much work when proving properties of programs.

Big-step semantics:
- Quicker to prove properties of programs because there are less rules.
- Since we drop all knowledge about intermediary configurations, all programs without final configurations look the same:
  Loops, errors, deadlocks.
  Sometimes one cannot prove properties about them in the big-step semantics.

Note:
- Structural operational semantics are also called natural semantics, because the proof system is in the style of natural deduction.
- The semantic brackets $\mathcal{S}[\![1+1]\!](G)$ put the inner term into quotes, and thus can be understood as $\mathcal{S}(\text{"}1+1\text{"})(G)$. The quotes indicate that the piece of syntax "1+1" is being mapped, and not the arithmetic expression 1+1 being evaluated.