Semantics:

Goal:
- Reason about the runtime behavior of programs
  - ↳ Define it.
  - ↳ Understand the properties we could want.
  - ↳ Learn techniques to prove or disprove these properties.
- The runtime behavior is what we refer to as semantics.

Formally:
- There is a programming language PL that contains the programs that are syntactically valid.
- A semantics of this programming language is a function
  $$[\![-]\!] : PL \longrightarrow SemDom.$$
  It takes a program $p \in PL$ and assigns it an element in a semantic domain (that it also defines),
  $$[\![p]\!] \in SemDom.$$
- The following styles of defining semantics are common:

Operational semantics (aka small step):

↳ $[\![p]\!]$ is a transition system
whose nodes are configurations Γ and
whose transitions modify configurations
depending on the command that
has been executed.

Predicate transformer /
Axiomatic semantics (aka big step)

↳ $[\![p]\!] : \mathcal{P}(\Sigma) \longrightarrow \mathcal{P}(\Sigma)$
is a function that transforms
a set of input states
into a set of output states.
It does not reveal
what happens in-between.

$\Gamma = P(L \times \Sigma)$

↳ Sets of states are also called predicates,
and then the semantics of a program
is a predicate transformer (Dijkstra '75).

↳ Axiomatic refers to the fact that
$[\![p]\!]$ was not given explicitly in the past
but only specified through axioms
in Hoare logic.

Denotational semantics

↳ $[\![p]\!]$ is constructed compositionally,

and the semantic domain has predicates
that match the ones in the language.

⇒ Predicate transformers are
a denotational semantics,
but the concept is more general.

Language-theoretic semantics:

$[\![p]\!] \subseteq Act^*$

is a language of visible actions.
They circumvent the problem that Hoare logic
is so tightly linked to the syntax

1 while-Programs

Goal: Introduce a while-programming language PL
and illustrate the styles of semantics.
we are here ..... SemDom

Definition:
Let Com be a set of commands, including skip.
Then the programming language WL(Com)
contains the programs defined by the following grammar:

$$c ::= com \mid c_1 ; c_2 \mid c_1 + c_2 \mid c_1^*,$$

where $com \in Com$.

Remark:
if and while are syntactic sugar

if b then c_1 else c_2 fi $:=$ [assume b; c_1] + [assume $\neg b$; c_2]

(has to be in COM)

while b do c od $:=$ (assume b; c)*; assume $\neg b$

There is a set of (memory) states Σ

and every command $com \in COM$

is assigned a relation $[\![com]\!] \subseteq \Sigma \times \Sigma$.

We expect $[\![skip]\!] = id$.

Example: $\Sigma := Vars \to \mathbb{Z}$

$[\![x{+}{+}]\!] := \{(\sigma, \sigma[x \mapsto \sigma(x)+1]) \mid \sigma \in \Sigma\}$.

Structured operational semantics (Plotkin '81):

Since programs are defined inductively.

Also the transition relation should be defined inductively.

Use a proof system.

Recall: $T = W(COM) \times \Sigma$.

We associate with the programming language

a transition relation

$$\to \subseteq T \times T,$$

namely the smallest relation

that satisfies the following rules:

$$(com) \quad \frac{(\sigma, \sigma') \in [\![com]\!]}{(com, \sigma) \to (skip, \sigma')}$$

$$(skip)\ \frac{}{(skip;c,\sigma)\to(c,\sigma)} \qquad (seq)\ \frac{(c,\sigma)\to(c',\sigma')}{(c;\tilde{c},\sigma)\to(c';\tilde{c},\sigma')}$$

$$(choiceL)\ \frac{}{(c+\tilde{c},\sigma)\to(c,\sigma)} \qquad (choiceR)\ \frac{}{(\tilde{c}+c,\sigma)\to(c,\sigma)}$$

$$(loopend)\ \frac{}{(c^*,\sigma)\to(skip,\sigma)} \qquad (loop)\ \frac{}{(c^*,\sigma)\to(c;c^*,\sigma)}$$

The transition system semantics is then

$$TS[\![p]\!] := (T, \to, Init)$$

with $Init := \{p\} \times \Sigma$. // Start with the given program, the data can be arbitrary.

Why proof systems?
↳ Very flexible.

Predicate transformer semantics

(as an instance of denotational semantics)

Note that $[\![com]\!] \subseteq \Sigma \times \Sigma$

can be equivalently written as

$$[\![com]\!] : \Sigma \to \mathcal{P}(\Sigma).$$

We lift it to a predicate transformer

$$PT[\![com]\!] : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$$

$$[\![com]\!]\, A := \bigcup_{a \in A} [\![com]\!]\, a.$$

Then

$$PT[\![c_1 + c_2]\!] := PT[\![c_1]\!] \cup PT[\![c_2]\!]$$

$$PT[\![c_1; c_2]\!] := PT[\![c_2]\!] \circ PT[\![c_1]\!]$$

$$PT[\![c^*]\!] := \bigcup_{i \geq 0} PT[\![c^i]\!] \quad \text{with} \quad \begin{array}{l} c^0 := skip \\ c^{i+1} := c; c^i. \end{array}$$

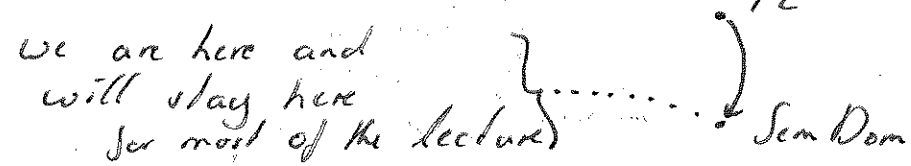
↳ Note that $\cup$, $\circ$, $\bigcup_{i \geq 0}$ are used to interpret $+$, $;$, $-^*$ on the semantic domain.

↳ While elegant, predicate transformer / denotational semantics are not as flexible as operational semantics.

Language-theoretic semantics:

$\mathcal{L}[\![p]\!]$ = build an automaton whose states and transitions are given by the small-steps semantics and whose alphabet is Com.

2. Safety



Safety:
- Nothing bad ever happens.
- Counterexamples are finite computations.

Old-school:
- Write down the safety property in a temporal logic
- Check that the program satisfies the property.

State-of-the-art:
- Assume that the program has been instrumented so that the property holds

iff

no bad configuration is reachable.

Throughout: Transition system semantics, $TS[\![p]\!] = (\Gamma, \to, \mathrm{init})$.
Let $\to^* := \bigcup_{i \geq 0} \to^i$ with $\to^0 := id$, $\to^{i+1} := \to^i ; \to$.

Given predicates $A, B \subseteq \Gamma$ // We also call sets of configurations predicates.

and a relation $R \subseteq \Gamma \times \Gamma$,

let

$$\forall R := \{ b \in \Gamma \mid \exists a \in A.\ a R b \}$$

$$R B := \{ a \in \Gamma \mid \exists b \in B.\ a R b \}.$$

Let $Reach(p) := Init \to^*$.

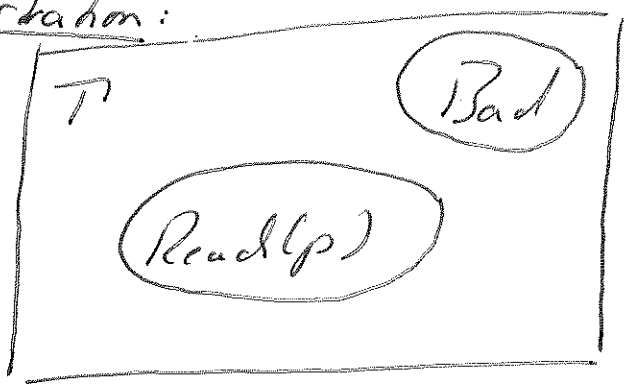
The safety verification problem is this.

SAFETY:

Given: Program p, bad configurations $Bad \subseteq \Gamma$.

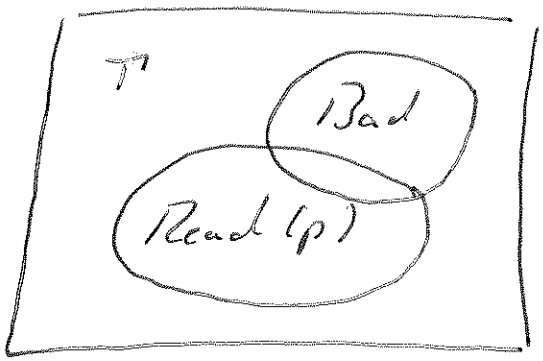
Question: $Reach(p) \cap Bad = \emptyset$.

Illustration:



Correct

vs.



Incorrect.

Goal (throughout the lecture):

↳ Understand which proof objects

are needed to prove or disprove

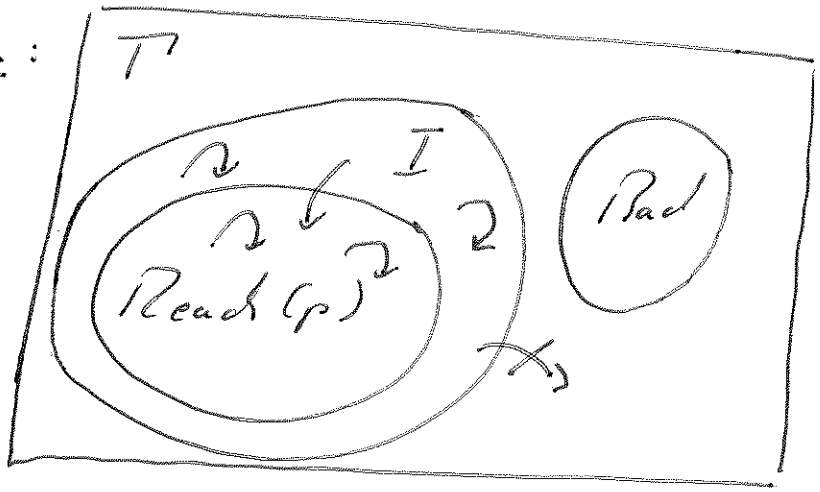
a property.

↳ For safety, the proof object are inductive invariants.

Definition:

- An inductive invariant for program P is a predicate $I \subseteq \Gamma$ with
  1) $Init \subseteq I$
  2) $\underbrace{I \rightarrow \subseteq I}$ closed under taking transitions.
- The invariant is save wrt. $Bad \subseteq \Gamma$, if $I \cap Bad = \emptyset$ holds.

Illustration:



Theorem (Soundness and completeness of inductive invariants for proving safety):

$Reach(P) \cap Bad = \emptyset$ iff $\exists$ ind. invariant for P that is save w.r.t. Bad.

Proof:

Soundness „⇐“:

Let I be an inductive invariant that is save wrt. Bad.

Since $I \to \subseteq I$,

we get $I \to^* \subseteq I$.

Since $init \subseteq I$,

we get $init \to^* \subseteq I$.

$$\| $$

$$Reach(p)$$

Since $I \cap Bad = \emptyset$,

we get $Reach(p) \cap Bad = \emptyset$.

(Completeness „⇒“:

Let $I := Reach(p)$.

We have $init \subseteq Reach(p)$ by definition.

Moreover, $Reach(p) \to = (init \to^*) \to$

$$\subseteq init \to^* = Reach(p).$$

This means $Reach(p)$ is an inductive invariant.

We have

$$Reach(p) \cap Bad = \emptyset$$

by assumption, so $Reach(p)$ is also save.

□

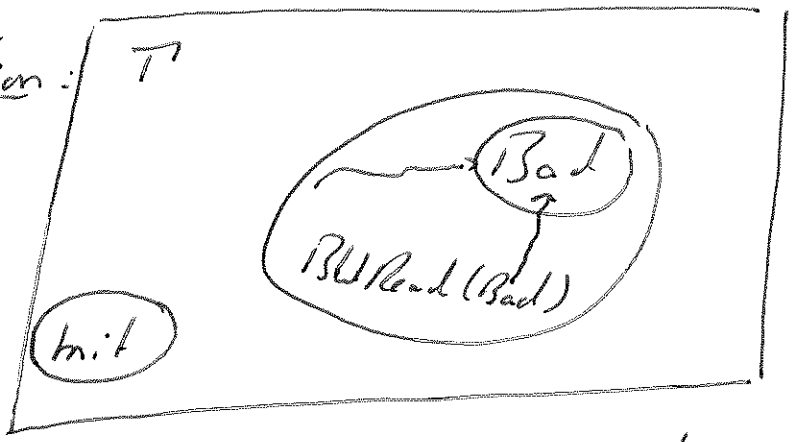
Remark:

- Completeness does not matter in practice.
  The whole point of using invariants
  is that we want to prove safety
  with predicates larger than Reach(p).
- Reach(p) is the smallest inductive invariant.
  There is also a largest, namely

$$\overline{BWReach(Bad)}$$

  with $BWReach(Bad) := \longrightarrow^* Bad$.

Illustration:



- One could also define backwards invariants
  as

$$\longrightarrow I \subseteq I.$$

- When a candidate for an invariant is given,
  checking invariance and safety is computationally easy.
  This is also called proof checking:

manual verification $\longleftarrow$ proof checking $\longrightarrow$ automated verification.

- There are also <u>invariants</u> that are not necessarily inductive, meaning

$$Reach(p) \subseteq I$$

holds but

$$I \rightarrow \subseteq I \quad \text{does not.}$$

But they do not give us a way to check that a candidate invariant is an invariant indeed.
So verification will always work with inductive invariants.