Deductive Verification of Programs

6.0 VU Formal Methods in Computer Science

Gernot Salzer

AB Theoretische Informatik und Logik
Institut für Computersprachen

13 November 2013

Schedule Block 3

Lecture:

We, 13.11., 12:00–14:00, EI 9
Mo, 18.11., 13:00–15:00, EI 10
We, 20.11., 12:00–14:00, EI 9
Mo, 25.11., 13:00–15:00, EI 10

Exercises:

Tu, 26.11., 17:00–19:00, EI 9
So, 1.12., 24:00, TUWEL deadline for exercise uploads
Mo, 9.12., 13:00–15:00, EI 10

Topics today

1. Why formal methods?
2. Syntax of TPL (Toy Programming Language)
3. Operational Semantics of TPL

What can we do with this program?

/* This program multiplies x and y. */

$z := 0$;
while $y \neq 0$ **do**
  $z := z + x$;
  $y := y - 1$
od

Almost nothing!

If you are lucky, someone wrote a compiler/interpreter for your computer.

- You can run some test cases (manually).
- You can use the result of the program if you trust it.

Only works . . .

- if you and the compiler writer agree on the meaning of the program,
- if the compiler and the program have been implemented correctly.

What can we do with this program?

/* This program multiplies x and y. */

$z := 0$;
while $y \neq 0$ **do**
  $z := z + x$;
  $y := y - 1$
od

But you cannot . . .

- check specification for completeness (every case covered?)
- verify correctness of specification (does it specify what you mean?)
- let other people write the program (implicit assumptions)
- reuse the program safely (how to verify that it fits into new context?)
- generate test cases from specification or program automatically
- prove correctness of program and machine code
- . . .

Analysis and automation require formal methods!

A formal specification of the problem allows you to ...

- check for missing cases (incomplete spec),
- verify its correctness (e.g. by simulations),
- generate test cases automatically,
- generate (parts of) the program automatically,
- outsource the programming,
- safely reuse components,
- detect errors at run time (e.g. by assertions),
- write better software (abstraction, modularisation),
- improve documentation and maintenance,
- prove properties of programs.

Analysis and automation require formal methods!

Formal syntax&semantics of the programming language allows you to ...

- generate the compiler (correct by construction!),
- improve documentation,
- do static program analysis (e.g. to generate tests),
- prove properties of programs (correctness, termination, time/space bounds, fairness, liveness, ...).

A formal specification of the computer architecture allows you to ...

- generate the compiler (incl. code generation),
- prove correctness of machine program.

Formal methods increase quality and productivity.

"Do what I mean" does not work with computers.

If you want the computer to help you, you have to become formal.

Why aren't then formal methods everywhere?

Formal methods can be expensive.

- You need experts in formal methods AND the domain.
  "Knowledge acquisition bottleneck"
- You need time.
- Formal methods are difficult/impossible to apply to legacy systems.
  Formal methods are best used before and during development, not after the system is finished.

Therefore formal methods are mainly used for ...

- Safety-critical applications (e.g. railway switches)
- Security-critical applications (e.g. electronic banking)
- Financial reasons (e.g. smart cards)
- Legal reasons (e.g. EAL 6/7 in Common Criteria)

Formal methods ARE (almost) everywhere!

- Formal specification methods: VDM, Z, Object-Z, B, Perfect, ...
- Unified Modeling Language – UML:
  Graphical language for object-oriented modelling
  Standard of the Object Management Group (OMG)
- Object Constraint Language – OCL:
  Formal textual assertion language
  UML Substandard
- Consolidation and documentation of design knowledge:
  Patterns, idioms, architectures, frameworks, etc.
- "Design by contract" (B. Meyer, Eiffel)
- Specification in familiar languages: Java – JML, C# – Spec#, ...
- Model checking (more on that in the next block by Helmut Veith)
- Numerous verification systems: CBMC, eCv, Frama-C/Jessie, KeY, Perfect Developer, PVC, SLAM, VCC, Verifast, ...

Types of Requirements vs. Formal Methods

Requirements

- functional requirements
- communication protocols
- real-time requirements
- memory use
- security
- robustness
- ...

Formal methods

- deductive verification
- model checking
- static analysis
- run-time checks (of formal specification)
- ...

Inspired by Bernhard Beckert's slides, Karlsruhe, 2008

Limitations of Formal Methods

Possible reasons for errors

- Program is not correct (does not satisfy specification).
  Formal verification proves absence of this kind of error.
- Program is not adequate (error in specification).
  Formal methods help to find this kind of error.
- Error in operating system, compiler, hardware.
  Not covered by formal methods (unless OS, compiler etc. is formally specified/verified).

No full specification/verification

In general, it is neither useful nor feasible to fully specify and verify large software systems. Formal methods are restricted to:

- Important parts/modules
- Important properties/requirements

Inspired by Bernhard Beckert's slides, Karlsruhe, 2008

The Main Point of Formal Methods is Not ...

- to show the "correctness" of entire systems
  (What IS correctness? Always go for specific properties!)
- to replace testing entirely
- to replace good design practices

There is no silver bullet that lets you get away
without crystal clear requirements and good design.

In particular, formal methods can't do it.

But ...

- A formal proof replaces many test cases.
- Formal methods can be used in automatic test case generation.
- Formal methods improve the quality of specifications.

Inspired by Bernhard Beckert's slides, Karlsruhe, 2008

Toy Programming Language (TPL)

$$
\begin{array}{l}
z := 0; \\
\textbf{while } y \neq 0 \textbf{ do} \\
\quad z := z + x; \\
\quad y := y - 1 \\
\textbf{od}
\end{array}
$$

- Which character strings can be interpreted as programs?
  $\Longrightarrow$ syntax of TPL
- What do programs mean?
  $\Longrightarrow$ semantics of TPL
- What is the program supposed to do?
  $\Longrightarrow$ formal specification of intended behaviour
- Does the program do what it is supposed to do?
  $\Longrightarrow$ formal verification of program

Topics today

1. Why formal methods?
2. Syntax of TPL (Toy Programming Language)
3. Operational Semantics of TPL

TPL Syntax (1)

Programs

$\mathcal{P} ::=$	"**skip**"	no operation
	$\mid$ "**abort**"	error exit
	$\mid \mathcal{V}$ ":=" $\mathcal{E}$	assignment
	$\mid \mathcal{P}$ ";" $\mathcal{P}$	sequential composition
	$\mid$ "**if**" $\mathcal{E}$ "**then**" $\mathcal{P}$ "**else**" $\mathcal{P}$ "**fi**"	if-then-else
	$\mid$ "**while**" $\mathcal{E}$ "**do**" $\mathcal{P}$ "**od**"	loops

Expressions

$$\mathcal{E} ::= \mathcal{V} \mid \mathcal{N} \mid \mathcal{U}\,\mathcal{E} \mid \text{"("}\ \mathcal{E}\,\mathcal{B}\,\mathcal{E}\ \text{")"}$$

TPL Syntax (2)

Variables

$$\mathcal{V} ::= \text{“}x\text{”} \mid \text{“}y\text{”} \mid \cdots \mid \text{“}x_0\text{”} \mid \text{“}x_1\text{”} \mid \cdots \mid \text{any word except key words} \mid \cdots$$

Integer numerals

$$\mathcal{N} ::= \text{“}0\text{”} \mid \text{“}1\text{”} \mid \cdots \mid \text{“}9\text{”} \mid \text{“}10\text{”} \mid \text{“}11\text{”} \mid \cdots \mid \text{“}42\text{”} \mid \cdots$$

Unary and Binary Operators

$$\begin{aligned}
\mathcal{U} ::= &\ \text{“}+\text{”} \mid \text{“}-\text{”} \mid \text{“}\neg\text{”} \mid \cdots \\
\mathcal{B} ::= &\ \text{“}+\text{”} \mid \text{“}-\text{”} \mid \text{“}*\text{”} \mid \text{“}/\text{”} \\
&\mid \text{“}<\text{”} \mid \text{“}\leq\text{”} \mid \text{“}=\text{”} \mid \text{“}\geq\text{”} \mid \text{“}>\text{”} \\
&\mid \text{“}\wedge\text{”} \mid \text{“}\vee\text{”} \mid \text{“}\Rightarrow\text{”} \mid \cdots
\end{aligned}$$

TPL Syntax (3)

$$
\begin{array}{lcl}
\mathcal{P} & \Rightarrow & \mathcal{P}; \mathcal{P} \\
& \stackrel{*}{\Rightarrow} & \mathcal{V} := \mathcal{E};\ \textbf{while}\ \mathcal{E}\ \textbf{do}\ \mathcal{P}\ \textbf{od} \\
& \stackrel{*}{\Rightarrow} & z := \mathcal{N};\ \textbf{while}\ \mathcal{U}\,\mathcal{E}\ \textbf{do}\ \mathcal{P}; \mathcal{P}\ \textbf{od} \\
& \stackrel{*}{\Rightarrow} & z := 0;\ \textbf{while}\ \neg(\mathcal{E}\,\mathcal{B}\,\mathcal{E})\ \textbf{do}\ \mathcal{V} := \mathcal{E};\ \mathcal{V} := \mathcal{E}\ \textbf{od} \\
& \stackrel{*}{\Rightarrow} & z := 0;\ \textbf{while}\ \neg(\mathcal{V} = \mathcal{N})\ \textbf{do}\ z := (\mathcal{E}\,\mathcal{B}\,\mathcal{E});\ y := (\mathcal{E}\,\mathcal{B}\,\mathcal{E})\ \textbf{od} \\
& \stackrel{*}{\Rightarrow} & z := 0;\ \textbf{while}\ \neg(y = 0)\ \textbf{do}\ z := (\mathcal{V} + \mathcal{V});\ y := (\mathcal{V} - \mathcal{N})\ \textbf{od} \\
& \stackrel{*}{\Rightarrow} & z := 0;\ \textbf{while}\ \neg(y = 0)\ \textbf{do}\ z := (z + x);\ y := (y - 1)\ \textbf{od} \\
& \sim & z := 0;\ \textbf{while}\ y \neq 0\ \textbf{do}\ z := z + x;\ y := y - 1\ \textbf{od}
\end{array}
$$

Brush up your knowledge about context-free grammars (CFGs)!
(What is $\Rightarrow$ and $\stackrel{*}{\Rightarrow}$?)

TPL Syntax (4)

Notational conveniences

- Overloading: $\mathcal{P}$, $\mathcal{E}$, $\mathcal{V}$, ... denote
  - grammar variables (non-terminals)
    $\mathcal{E} ::= \mathcal{V} \mid \mathcal{N} \mid \mathcal{U}\,\mathcal{E} \mid (\mathcal{E}\,\mathcal{B}\,\mathcal{E})$
  - languages (sets of strings) generated by grammar variables
    $\mathcal{E} = \{\, x, y, \ldots, 0, 1, \ldots, -x, \neg 0, \ldots, ((x + 5) + (3 * y)), \ldots \}$
- Parentheses omitted when justified by operator precedence and associativity
  $x + 5 + 3 * y$ instead of $((x + 5) + (3 * y))$
- Simplified notations
  $x \neq 3y$ instead of $\neg(x = 3 * y)$

Topics today

1. Why formal methods?
2. Syntax of TPL (Toy Programming Language)
3. Operational Semantics of TPL

Syntax and Semantics

Syntax: rules specifying which strings of symbols are admissible

Semantics: assigns meaning to syntactically correct strings; needs meta-language (like mathematics or logic) to specify meaning.

- "Furiously sleep ideas green colorless."
  Syntactically incorrect, no semantics.
- "Colorless green ideas sleep furiously."
  Syntactically correct, but meaningless (no semantics).
- "Mount Everest is the highest mountain on earth."
  Syntactically correct, can be given meaning.

(Noam Chomsky used the first two sentences in 1957 to show that the syntax of natural languages cannot be explained by probabilistic models.)

Various forms of formal semantics

Operational semantics: Simulates program execution

- natural semantics (maps objects to final result)
- structural operational semantics (mimicks computation steps)

Denotational semantics: Constructs mathematical objects as meaning of programs; least fixed point semantics

- direct style semantics
- continuation style semantics

Axiomatic semantics: Describe the effect of programs on assertions about the program state

- Partial correctness
- Total correctness (= partial correctness + termination)

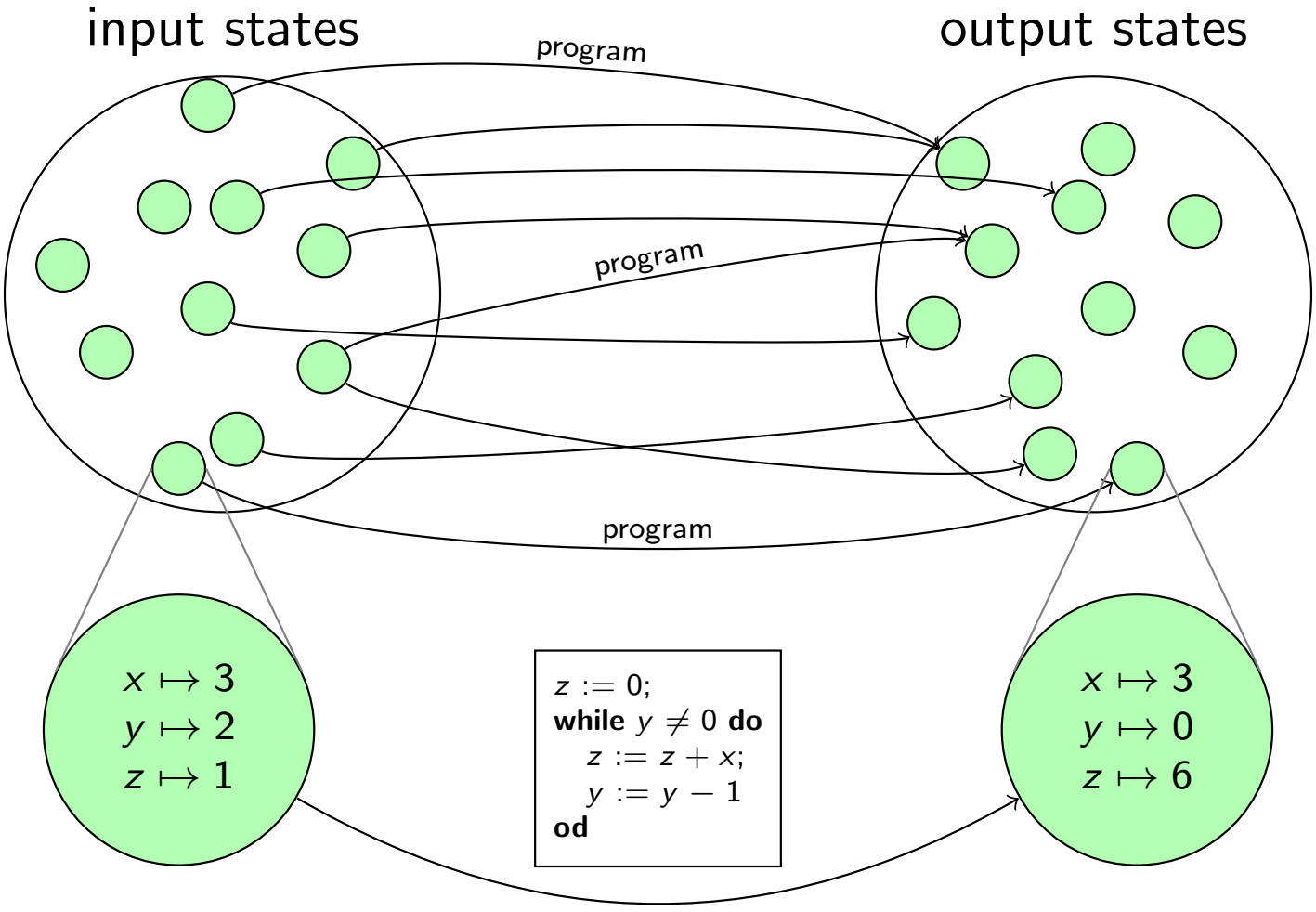
Which type of semantics to use?

Choice depends on ...

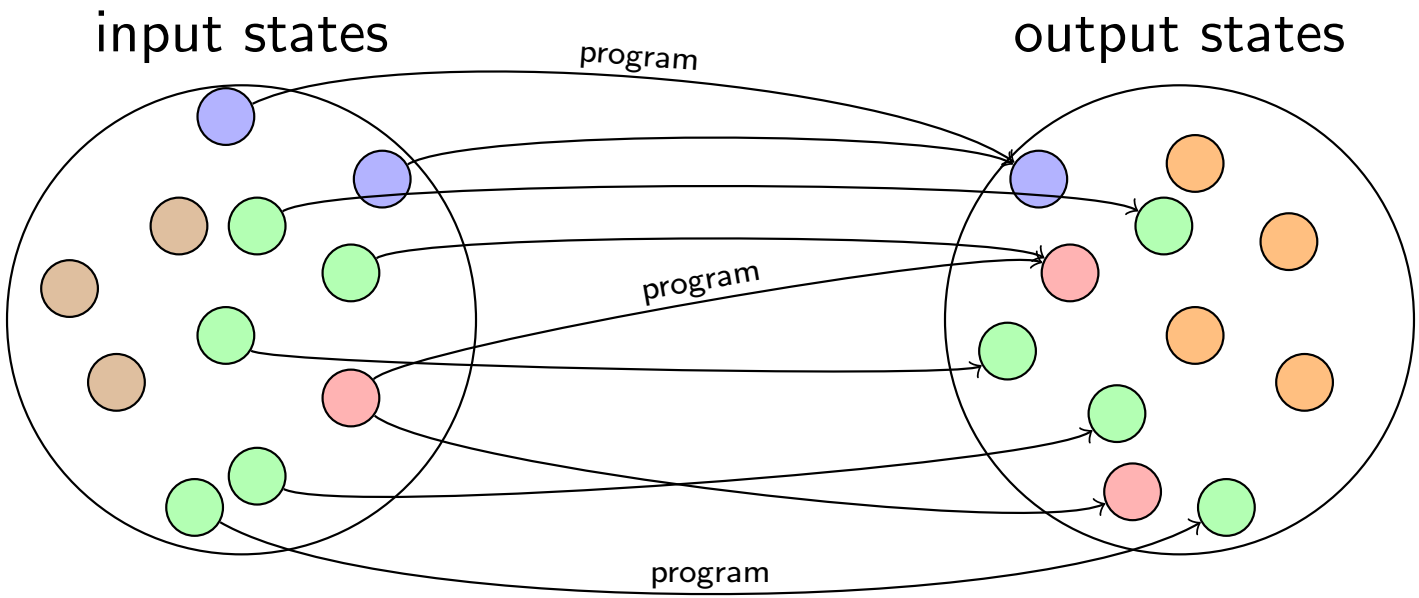
- constructs of the language:
  - imperative
  - functional/relational
  - concurrent/parallel
  - object-oriented
  - non-deterministic
  - ...
- what the semantics is used for:
  - understanding the language
  - verification of programs
  - prototyping
  - compiler construction
  - program analysis
  - ...

Understanding an imperative language $\Rightarrow$ operational semantics
Verifying imperative programs $\Rightarrow$ axiomatic semantics

Programs as State Transformers



Programs as State Transformers



- Several inputs may be mapped to the same output.
- Some inputs are not mapped to any output.
  (Program aborts or loops.)
- Some outputs are not reached from any input.
- One input may be mapped to several outputs (indeterminism).

Program States

Informally: memory snapshots

Formally: mappings from variables to values (integers in our case)

Set of all states: $\mathcal{S} \stackrel{\text{def}}{=} \{\, \sigma \mid \sigma : \mathcal{V} \mapsto \mathbb{Z} \,\}$

Our example program p maps state σ to state σ':

$$\begin{array}{l} \sigma(x) = 3 \\ \sigma(y) = 2 \\ \sigma(z) = 1 \end{array} \quad \xrightarrow{p} \quad \begin{array}{l} \sigma'(x) = 3 \\ \sigma'(y) = 0 \\ \sigma'(z) = 6 \end{array}$$

We write this as $[p](\sigma) = \sigma'$ or $[p]\,\sigma = \sigma'$.

$[p] : \mathcal{S} \mapsto \mathcal{S}$ ... (partial) function computed by program p
semantics (= meaning) of p

Configurations

Informally: system snapshots ("dumps"), consisting of

- the program that remains to be executed, and
- a memory snapshot (state).

Formally: pair (p, σ), where $p \in \mathcal{P}$ and $\sigma \in \mathcal{S}$,
or just a state σ (final configuration).

Set of configurations:

$$\mathcal{C} \stackrel{\text{def}}{=} (\mathcal{P} \times \mathcal{S}) \cup \mathcal{S}$$

Transitions

Transition relation $\Rightarrow$: describes a single computation step,
relates a non-final configuration to the successor configuration:

$$\Rightarrow \subseteq (\mathcal{P} \times \mathcal{S}) \times \mathcal{C}$$

Typical situations:

- $(p, \sigma) \Rightarrow (p', \sigma')$: intermediate step of computation
- $(p, \sigma) \Rightarrow \sigma'$: last step of computation

A transition relation is

- deterministic if for every configuration γ, there is at most one configuration γ' such that $\gamma \Rightarrow \gamma'$.
- indeterministic if for some configuration γ, there are configurations γ', γ'' such that $\gamma \Rightarrow \gamma'$, $\gamma \Rightarrow \gamma''$, and $\gamma' \neq \gamma''$.

Transition Relation for TPL (1)

skip: leave state unchanged

$$(\textbf{skip}, \sigma) \Rightarrow \sigma$$

abort: abort execution, no transition defined

$$(\textbf{abort}, \sigma) \not\Rightarrow \ldots$$

(Minimal exception handling)

p; q: execute from left to right, starting with first statement of p

$$\frac{(p, \sigma) \Rightarrow (p', \sigma')}{(p; q, \sigma) \Rightarrow (p'; q, \sigma')} \qquad \frac{(p, \sigma) \Rightarrow \sigma'}{(p; q, \sigma) \Rightarrow (q, \sigma')}$$

Read: If (p, σ) transforms to (p', σ')
then $(p; q, \sigma)$ transforms to $(p'; q, \sigma')$.

Transition Relation for TPL (2)

if e then p else q fi: execute p if the expression e evaluates to true, otherwise execute q.

$$\frac{[e]\,\sigma \neq 0}{(\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}, \sigma) \Rightarrow (p, \sigma)} \qquad \frac{[e]\,\sigma = 0}{(\textbf{if } e \textbf{ then } p \textbf{ else } q \textbf{ fi}, \sigma) \Rightarrow (q, \sigma)}$$

- $[e] : \mathcal{S} \mapsto \mathbb{Z}$ denotes the function computed by expression e. Takes current state as argument and returns value of expression in this state. (See below.)
- Design decision: 0 represents false, any other value (in particular 1) represents true.
  Advantage: we don't need "boolean" as second data type.
- If without else: **if** e **then** p **else skip fi**

Transition Relation for TPL (3)

while e do p od: do nothing if e evaluates to false, otherwise execute p and then redo the while-statement.

$$\frac{[e]\,\sigma = 0}{(\textbf{while } e \textbf{ do } p \textbf{ od}, \sigma) \Rightarrow \sigma}$$

$$\frac{[e]\,\sigma \neq 0}{(\textbf{while } e \textbf{ do } p \textbf{ od}, \sigma) \Rightarrow (p; \textbf{while } e \textbf{ do } p \textbf{ od}, \sigma)}$$

v := e: the new state is like the old one, except that variable v equals the value of expression e (in the old state).

$$(v := e, \sigma) \Rightarrow \sigma' \qquad \text{where} \quad \begin{array}{l} \sigma'(v) = [e]\,\sigma \\ \sigma'(x) = \sigma(x) \quad \text{for } x \neq v \end{array}$$

Semantics of Expressions

Assignments, **if**- and **while**-statements evaluate expressions e.

$[e]\colon \mathcal{S} \mapsto \mathbb{Z}$ ... (partial) function computed by expression e

$$
\begin{array}{rll}
[v]\,\sigma = \sigma(v) & & \text{for } v \in \mathcal{V} \\
[n]\,\sigma = [n] & & [n] \in \mathbb{Z} \text{ ... number for numeral } n \in \mathcal{N} \\
[u\,e]\,\sigma = [u]([e]\,\sigma) & & [u]\colon \mathbb{Z} \mapsto \mathbb{Z} \text{ ... func. for operator } u \in \mathcal{U} \\
[e\,b\,e']\,\sigma = [b]([e]\,\sigma, [e']\,\sigma) & & [b]\colon \mathbb{Z}^2 \mapsto \mathbb{Z} \text{ ... func. for operator } b \in \mathcal{B}
\end{array}
$$

Overloading of [.]!

- $[p]\colon \mathcal{S} \mapsto \mathcal{S}$ ... evaluation of programs (see below)
- $[e]\colon \mathcal{S} \mapsto \mathbb{Z}$ ... evaluation of expressions
- $[n]$, $[u]$, $[b]$ ... semantic entities corresponding to operators

Semantics of Operators

Boolean functions:

$[\neg]$	
0	1
$\neq 0$	0

$[\wedge]$	0	$\neq 0$
0	0	0
$\neq 0$	0	1

$[\vee]$	0	$\neq 0$
0	0	1
$\neq 0$	1	1

$[\Rightarrow]$	0	$\neq 0$
0	1	1
$\neq 0$	0	1

Integer functions:

$[+]$ ... Integer addition or positive sign $(+)$
$[-]$ ... Integer subtraction or negative sign $(-)$
$[*]$ ... Integer multiplication $(\cdot)$
$[/]$ ... Integer division with truncation

Integer predicates:

$[<]$, $[\leq]$, $[=]$, $[\geq]$, $[>]$ ... comparison of integers

Expressions: Example

σ: $x \mapsto 1$, $y \mapsto 2$

$$
\begin{aligned}
&[x < y \wedge y < 2 * x + 1]\,\sigma \\
&= [\wedge]([x < y]\,\sigma,\ [y < 2 * x + 1]\,\sigma) \\
&= [\wedge]([<]([x]\,\sigma, [y]\,\sigma),\ [<]([y]\,\sigma, [2 * x + 1]\,\sigma)) \\
&= [\wedge]([<](\sigma(x), \sigma(y)),\ [<](\sigma(y), [+]([2 * x]\,\sigma, [1]\,\sigma))) \\
&= [\wedge]([<](1, 2),\ [<](2, [+]([*]([2]\,\sigma, [x]\,\sigma), 1))) \\
&= [\wedge](1,\ [<](2, [+]([*](2, \sigma(x)), 1))) \\
&= [\wedge](1,\ [<](2, [+]([*](2, 1), 1))) \\
&= [\wedge](1,\ [<](2, [+](2, 1))) \\
&= [\wedge](1,\ [<](2, 3)) = [\wedge](1,\ 1) = 1
\end{aligned}
$$

Program Runs

Program run: Sequence $\langle \gamma_0, \gamma_1, \gamma_2, \gamma_3, \ldots \rangle$ of configurations such that $\gamma_0 \Rightarrow \gamma_1$, $\gamma_1 \Rightarrow \gamma_2$, $\gamma_2 \Rightarrow \gamma_3$, $\ldots$
written as $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \gamma_3 \Rightarrow \cdots$.

Finite program run of length k: $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_k$

A finite program run is complete if γ_k has no successor, i.e., if there is no $\gamma \in \mathcal{C}$ such that $\gamma_k \Rightarrow \gamma$.

The last configuration γ_k is called

- final if $\gamma_k \in \mathcal{S}$: program terminates.
- stuck if $\gamma_k \in (\mathcal{P} \times \mathcal{S})$: program aborts.

Infinite program run: program loops.

Program Runs: Examples

$$
\begin{aligned}
(p,\sigma) &= (z := 0; \textbf{while } \ldots, \sigma) \\
&\quad (z := 0, \sigma) \Rightarrow \sigma_1 \\
&\Rightarrow (\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}, \sigma_1) \\
&\Rightarrow (z := z + x; y := y - 1; \textbf{while } \ldots, \sigma_1) \\
&\quad (z := z + x; y := y - 1, \sigma_1) \\
&\qquad (z := z + x, \sigma_1) \Rightarrow \sigma_2 \\
&\quad \Rightarrow (y := y - 1, \sigma_2) \\
&\Rightarrow (y := y - 1; \textbf{while } \ldots, \sigma_2) \\
&\quad (y := y - 1, \sigma_2) \Rightarrow \sigma_3 \\
&\Rightarrow (\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}, \sigma_3) \\
&\Rightarrow (z := z + x; y := y - 1; \textbf{while } \ldots, \sigma_3) \\
&\quad (z := z + x; y := y - 1, \sigma_3) \\
&\qquad (z := z + x, \sigma_3) \Rightarrow \sigma_4 \\
&\quad \Rightarrow (y := y - 1, \sigma_4) \\
&\Rightarrow (y := y - 1; \textbf{while } \ldots, \sigma_4) \\
&\quad (y := y - 1, \sigma_4) \Rightarrow \sigma_5 \\
&\Rightarrow (\textbf{while } y \neq 0 \textbf{ do } \ldots \textbf{ od}, \sigma_5) \\
&\Rightarrow \sigma_5
\end{aligned}
$$

p: $z := 0;$
while $y \neq 0$ **do**
$z := z + x;$
$y := y - 1$
od

σ: $x \mapsto 3, y \mapsto 2, z \mapsto 1$

σ_1: $z \mapsto [0]\,\sigma = 0$
$x \mapsto 3, y \mapsto 2$

$[y \neq 0]\,\sigma_1 = 1$ (true)

σ_2: $z \mapsto [z + x]\,\sigma_1 = 3$
$x \mapsto 3, y \mapsto 2$

σ_3: $y \mapsto [y - 1]\,\sigma_2 = 1$
$x \mapsto 3, z \mapsto 3$

$[y \neq 0]\,\sigma_3 = 1$ (true)

σ_4: $z \mapsto [z + x]\,\sigma_3 = 6$
$x \mapsto 3, y \mapsto 1$

σ_5: $y \mapsto [y - 1]\,\sigma_4 = 0$
$x \mapsto 3, z \mapsto 6$

$[y \neq 0]\,\sigma_5 = 0$ (false)

Program Runs: Examples

Let $\sigma = \{ x \mapsto 1 \}$.

$(\textbf{if } x > 0 \textbf{ then abort else skip fi}; x := 2x, \sigma)$

$(\textbf{if } x > 0 \textbf{ then abort else skip fi}, \sigma)$
$[x > 0]\,\sigma = 1$ (true)
$\Rightarrow (\textbf{abort}, \sigma)$

$\Rightarrow (\textbf{abort}; x := 2x, \sigma)$

The program run is complete, the last configuration is stuck, the program aborts.

Program Runs: Examples

$(\textbf{while } 1 \textbf{ do skip od}, \sigma)$

$\quad [1]\,\sigma = 1$ (true)

$\Rightarrow (\textbf{skip}; \textbf{while } 1 \textbf{ do skip od}, \sigma)$

$\quad (\textbf{skip}, \sigma) \Rightarrow \sigma$

$\Rightarrow (\textbf{while } 1 \textbf{ do skip od}, \sigma)$

$\quad [1]\,\sigma = 1$ (true)

$\Rightarrow \ldots$

Infinite program run, the program loops.

Structural Operational Semantics (SOS) of TPL

The function $[p]\colon \mathcal{S} \mapsto \mathcal{S}$ computed by a program p is defined by

$[p]\sigma = \sigma'$ if and only if $(p, \sigma) \stackrel{*}{\Rightarrow} \sigma'$ for all states $\sigma, \sigma' \in \mathcal{S}$.

($\stackrel{*}{\Rightarrow}$ ... reflexive and transitive closure of $\Rightarrow$)

Semantic equivalence

Two programs p and q are semantically equivalent if $[p] = [q]$.

This means:

- If $(p, \sigma) \stackrel{*}{\Rightarrow} \sigma'$, then $(q, \sigma) \stackrel{*}{\Rightarrow} \sigma'$, and vice versa.
- If (p, σ) loops or aborts, then so does (q, σ), and vice versa.

Note: The semantic function $[p]$ does not distinguish between endless loops and abortion, even though the transition relation does.