Semantics and Verification of Software

Lecture 16: Provably Correct Implementation I (Abstract Machine & Compiler)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH AACHEN UNIVERSITY

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/svsw11/

Winter Semester 2011/12

EINLADUNG

Zeit:	Mittwoch, 25. Januar 2012, 15:00 Uhr
Ort:	Hörsaal AH 3, Ahornstr. 55
Referent:	Dr. Thomas Noll RWTH Aachen
Thema:	Correctness, Safety and Fault Tolerance in Aerospace Systems: The ESA COMPASS Project

Building modern aerospace systems is highly demanding. They should be extremely dependable, offering service without failures for a very long time – typically years or decades. The need for an integrated system-software co-engineering framework to support the design of such systems is therefore pressing. However, current tools and formalisms tend to be tailored to specific analysis techniques and do not sufficiently cover the full spectrum of required system aspects such as safety, dependability and performability. Additionally, they cannot properly handle the intertwining of hardware and software operation. As such, current engineering practice lacks integration and coherence.

This talk gives an overview of the COMPASS project that was initiated by the European Space Agency to overcome this problem. It supports system-software co-engineering of real-time embedded systems by following a coherent and multidisciplinary approach. We show how such systems and their possible failures can be modeled in the Architecture and Analysis Design Language, how their behavior can be formalized, and how to analyze them by means of model checking and related techniques.

Es laden ein: Die Dozenten der Informatik

Wanted: HiWis

- To support Compiler Construction in Summer Semester
- Tasks:
  - evaluation of exercises
  - organizational support
- 12 hrs/week contract
- Previous CC lecture not a prerequisite (but of course helpful)

Outline

Procedure Environments

- Procedure environments now store semantic information:
  - So far: $PEnv := \{\pi \mid \pi : PVar \dashrightarrow Cmd \times VEnv \times PEnv\}$
  - Now: $PEnv := \{\pi \mid \pi : PVar \dashrightarrow (Sto \dashrightarrow Sto)\}$
- Procedure declarations ("$\texttt{proc}\ P\ \texttt{is}\ c$") update procedure environment:

$$\mathrm{upd}_p[\![.]\!] : PDec \times VEnv \times PEnv \to PEnv$$

  - non-recursive case: P not (indirectly) called within c $\Rightarrow \pi(P)$ immediately given by $\mathfrak{C}''[\![c]\!]\rho\,\pi$

$$\mathrm{upd}_p[\![\texttt{proc}\ P\ \texttt{is}\ c;p]\!](\rho,\pi) := \mathrm{upd}_p[\![p]\!](\rho,\pi[P \mapsto \mathfrak{C}''[\![c]\!]\rho\,\pi])$$

  - recursive case: $\pi(P)$ must be a solution of equation $P = \mathfrak{C}''[\![c]\!]\rho\,\pi$ (cf. fixpoint semantics of `while` loop – Slide 5.15)

$$\mathrm{upd}_p[\![\texttt{proc}\ P\ \texttt{is}\ c;p]\!](\rho,\pi) := \mathrm{upd}_p[\![p]\!](\rho,\pi[P \mapsto \mathrm{fix}(\Phi)])$$

    where $\Phi : (Sto \dashrightarrow Sto) \to (Sto \dashrightarrow Sto) : f \mapsto \mathfrak{C}''[\![c]\!]\rho\,\pi[P \mapsto f]$
  - $\mathrm{upd}_p[\![\varepsilon]\!](\rho,\pi) := \pi$

Statement Semantics Including Procedures

So far: $\mathfrak{C}'[\![.]\!] : \mathit{Cmd} \to \mathit{VEnv} \to (\mathit{Sto} \dashrightarrow \mathit{Sto})$

Definition (Denotational semantics with procedures)

$$\mathfrak{C}''[\![.]\!] : \mathit{Cmd} \to \mathit{VEnv} \to \mathit{PEnv} \to (\mathit{Sto} \dashrightarrow \mathit{Sto})$$

is given by:

$$\begin{aligned}
\mathfrak{C}''[\![\texttt{skip}]\!]\rho\,\pi &:= \mathrm{id}_{\mathit{Sto}} \\
\mathfrak{C}''[\![x\ \texttt{:=}\ a]\!]\rho\,\pi\,\sigma &:= \sigma[\rho(x) \mapsto \mathfrak{A}[\![a]\!](\mathrm{lookup}\ \rho\,\sigma)] \\
\mathfrak{C}''[\![c_1\,\texttt{;}\,c_2]\!]\rho\,\pi &:= (\mathfrak{C}''[\![c_2]\!]\rho\,\pi) \circ (\mathfrak{C}''[\![c_1]\!]\rho\,\pi) \\
\mathfrak{C}''[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]\rho\,\pi &:= \mathrm{cond}(\mathfrak{B}[\![b]\!] \circ (\mathrm{lookup}\ \rho), \\
&\qquad\qquad \mathfrak{C}''[\![c_1]\!]\rho\,\pi, \mathfrak{C}''[\![c_2]\!]\rho\,\pi) \\
\mathfrak{C}''[\![\texttt{while}\ b\ \texttt{do}\ c]\!]\rho\,\pi &:= \mathrm{fix}(\Phi) \\
\mathfrak{C}''[\![\texttt{call}\ P]\!]\rho\,\pi &:= \pi(P) \\
\mathfrak{C}''[\![\texttt{begin}\ v\ p\ c\ \texttt{end}]\!]\rho\,\pi\,\sigma &:= \mathfrak{C}''[\![c]\!]\rho'\,\pi'\,\sigma'
\end{aligned}$$

where

$$\begin{aligned}
\mathrm{upd}_v[\![v]\!](\rho, \sigma) &= (\rho', \sigma') \\
\mathrm{upd}_p[\![p]\!](\rho', \pi) &= \pi' \\
\mathrm{lookup}\ \rho\,\sigma &:= \sigma \circ \rho \\
\Phi(f) &:= \mathrm{cond}(\mathfrak{B}[\![b]\!] \circ (\mathrm{lookup}\ \rho), f \circ \mathfrak{C}''[\![c]\!]\rho\,\pi, \mathrm{id}_{\mathit{Sto}})
\end{aligned}$$

Summary: Blocks and Procedures

- Blocks allow to declare local variables and recursive procedures

Summary: Blocks and Procedures

- Blocks allow to declare local variables and recursive procedures
- Requires concept of locations to support instantiation of variables

Summary: Blocks and Procedures

- Blocks allow to declare local variables and recursive procedures
- Requires concept of locations to support instantiation of variables
- Static scoping: meaning of identifier determined by declaration context (rather than calling context)

Summary: Blocks and Procedures

- Blocks allow to declare local variables and recursive procedures
- Requires concept of locations to support instantiation of variables
- Static scoping: meaning of identifier determined by declaration context (rather than calling context)
- Meaning of variable declaration: storage allocation

Summary: Blocks and Procedures

- Blocks allow to declare local variables and recursive procedures
- Requires concept of locations to support instantiation of variables
- Static scoping: meaning of identifier determined by declaration context (rather than calling context)
- Meaning of variable declaration: storage allocation
- Meaning of procedure call:
  - operationally: execution of procedure body

  $\Rightarrow$ procedure environment records statement ("symbol table")
  - denotationally: application of procedure meaning

  $\Rightarrow$ procedure environment records (partial) store transformation
  - Recursive behavior again handled by fixpoint approach

Summary: Blocks and Procedures

- Blocks allow to declare local variables and recursive procedures
- Requires concept of locations to support instantiation of variables
- Static scoping: meaning of identifier determined by declaration context (rather than calling context)
- Meaning of variable declaration: storage allocation
- Meaning of procedure call:
  - operationally: execution of procedure body
  - ⇒ procedure environment records statement ("symbol table")
  - denotationally: application of procedure meaning
  - ⇒ procedure environment records (partial) store transformation
  - Recursive behavior again handled by fixpoint approach
- Further extensions:
  - axiomatic semantics (for $\texttt{proc}\ P\ \texttt{is}\ c$)
    - non-recursive: $(\text{call})\ \dfrac{\{A\}\ c\ \{B\}}{\{A\}\ \texttt{call}\ P\ \{B\}}$
    - recursive: $(\text{call})\ \dfrac{\{A\}\ \texttt{call}\ P\ \{B\} \vdash \{A\}\ c\ \{B\}}{\{A\}\ \texttt{call}\ P\ \{B\}}$
  - procedure parameters
  - higher-order procedures

Outline

Compiler Correctness

$$\begin{array}{rcl} \text{programming language} & \xrightarrow{\text{compiler}} & \text{machine code} \\ \text{semantics} \downarrow & & \downarrow \text{(simple) semantics} \\ \text{meaning} & \stackrel{?}{=} & \text{meaning} \end{array}$$

$$
\begin{array}{rcl}
\text{programming language} & \xrightarrow{\text{compiler}} & \text{machine code} \\
\text{semantics}\downarrow & & \downarrow\text{(simple) semantics} \\
\text{meaning} & \stackrel{?}{=} & \text{meaning}
\end{array}
$$

To do:

1. Definition of abstract machine
2. Definition (operational) semantics of machine instructions
3. Definition of translation WHILE $\rightarrow$ machine code ("compiler")
4. Proof: semantics of generated machine code = semantics of original source code

Outline

The Abstract Machine

Definition 16.1 (Abstract machine)

The abstract machine (AM) is given by

- configurations of the form $\langle d, e, \sigma \rangle \in \mathit{Cnf}$ where
  - $d \in \mathit{Code}$ is the sequence of instructions (code) to be executed
  - $e \in \mathit{Stk} := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top left)
  - $\sigma \in \Sigma := (\mathit{Var} \to \mathbb{Z})$ is the (storage) state

(thus $\mathit{Cnf} = \mathit{Code} \times \mathit{Stk} \times \Sigma$)

The Abstract Machine

Definition 16.1 (Abstract machine)

The abstract machine (AM) is given by

- configurations of the form $\langle d, e, \sigma \rangle \in Cnf$ where
  - $d \in Code$ is the sequence of instructions (code) to be executed
  - $e \in Stk := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top left)
  - $\sigma \in \Sigma := (Var \to \mathbb{Z})$ is the (storage) state

  (thus $Cnf = Code \times Stk \times \Sigma$)
- initial configurations of the form $\langle d, \varepsilon, \sigma \rangle$

The Abstract Machine

Definition 16.1 (Abstract machine)

The abstract machine (AM) is given by

- configurations of the form $\langle d, e, \sigma \rangle \in \mathit{Cnf}$ where
  - $d \in \mathit{Code}$ is the sequence of instructions (code) to be executed
  - $e \in \mathit{Stk} := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top left)
  - $\sigma \in \Sigma := (\mathit{Var} \to \mathbb{Z})$ is the (storage) state

  (thus $\mathit{Cnf} = \mathit{Code} \times \mathit{Stk} \times \Sigma$)
- initial configurations of the form $\langle d, \varepsilon, \sigma \rangle$
- final configurations of the form $\langle \varepsilon, e, \sigma \rangle$

The Abstract Machine

Definition 16.1 (Abstract machine)

The abstract machine (AM) is given by

- configurations of the form $\langle d, e, \sigma \rangle \in \mathit{Cnf}$ where
  - $d \in \mathit{Code}$ is the sequence of instructions (code) to be executed
  - $e \in \mathit{Stk} := (\mathbb{Z} \cup \mathbb{B})^*$ is the evaluation stack (top left)
  - $\sigma \in \Sigma := (\mathit{Var} \to \mathbb{Z})$ is the (storage) state

  (thus $\mathit{Cnf} = \mathit{Code} \times \mathit{Stk} \times \Sigma$)
- initial configurations of the form $\langle d, \varepsilon, \sigma \rangle$
- final configurations of the form $\langle \varepsilon, e, \sigma \rangle$
- code sequences d and instructions i:

$$
\begin{aligned}
d &::= \varepsilon \mid i : d \\
i &::= \texttt{PUSH}(z) \mid \texttt{ADD} \mid \texttt{MULT} \mid \texttt{SUB} \mid \\
&\quad\;\; \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{EQ} \mid \texttt{GT} \mid \texttt{AND} \mid \texttt{OR} \mid \texttt{NEG} \mid \\
&\quad\;\; \texttt{LOAD}(x) \mid \texttt{STORE}(x) \mid \texttt{NOOP} \mid \texttt{BRANCH}(d, d) \mid \texttt{LOOP}(d, d)
\end{aligned}
$$

  (where $z \in \mathbb{Z}$ and $x \in \mathit{Var}$)

Definition 16.2 (Transition relation of AM)

The transition relation $\rhd \subseteq Cnf \times Cnf$ is given by

$$
\begin{aligned}
\langle \mathtt{PUSH(}z\mathtt{)} : d, e, \sigma\rangle &\rhd \langle d, z : e, \sigma\rangle \\
\langle \mathtt{ADD} : d, z_1 : z_2 : e, \sigma\rangle &\rhd \langle d, (z_1 + z_2) : e, \sigma\rangle \\
\langle \mathtt{MULT} : d, z_1 : z_2 : e, \sigma\rangle &\rhd \langle d, (z_1 * z_2) : e, \sigma\rangle \\
\langle \mathtt{SUB} : d, z_1 : z_2 : e, \sigma\rangle &\rhd \langle d, (z_1 - z_2) : e, \sigma\rangle \\
\langle \mathtt{TRUE} : d, e, \sigma\rangle &\rhd \langle d, \mathsf{true} : e, \sigma\rangle \\
\langle \mathtt{FALSE} : d, e, \sigma\rangle &\rhd \langle d, \mathsf{false} : e, \sigma\rangle \\
\langle \mathtt{EQ} : d, z_1 : z_2 : e, \sigma\rangle &\rhd \langle d, (z_1 = z_2) : e, \sigma\rangle \\
\langle \mathtt{GT} : d, z_1 : z_2 : e, \sigma\rangle &\rhd \langle d, (z_1 > z_2) : e, \sigma\rangle \\
\langle \mathtt{AND} : d, t_1 : t_2 : e, \sigma\rangle &\rhd \langle d, (t_1 \wedge t_2) : e, \sigma\rangle \\
\langle \mathtt{OR} : d, t_1 : t_2 : e, \sigma\rangle &\rhd \langle d, (t_1 \vee t_2) : e, \sigma\rangle \\
\langle \mathtt{NEG} : d, t : e, \sigma\rangle &\rhd \langle d, \neg t : e, \sigma\rangle \\
\langle \mathtt{LOAD(}x\mathtt{)} : d, e, \sigma\rangle &\rhd \langle d, \sigma(x) : e, \sigma\rangle \\
\langle \mathtt{STORE(}x\mathtt{)} : d, z : e, \sigma\rangle &\rhd \langle d, e, \sigma[x \mapsto z]\rangle \\
\langle \mathtt{NOOP} : d, e, \sigma\rangle &\rhd \langle d, e, \sigma\rangle \\
\langle \mathtt{BRANCH(}d_{\mathsf{true}}\mathtt{,}d_{\mathsf{false}}\mathtt{)} : d, t : e, \sigma\rangle &\rhd \langle d_t : d, e, \sigma\rangle \\
\langle \mathtt{LOOP(}d_1\mathtt{,}d_2\mathtt{)} : d, e, \sigma\rangle &\rhd \langle d_1 : \mathtt{BRANCH(}d_2 : \mathtt{LOOP(}d_1\mathtt{,}d_2\mathtt{),NOOP)} : d, e, \sigma\rangle
\end{aligned}
$$

Remark: more traditional machine architectures

- Variables referenced by address (and not by name)
  - configurations $\langle d, e, m \rangle$ with memory $m \in \mathbb{Z}^*$
  - $\mathtt{LOAD}(x)/\mathtt{STORE}(x)$ replaced by $\mathtt{GET}(n)/\mathtt{PUT}(n)$ (where $n \in \mathbb{N}$)

Alternative Choices

Remark: more traditional machine architectures

- Variables referenced by address (and not by name)
  - configurations $\langle d, e, m\rangle$ with memory $m \in \mathbb{Z}^*$
  - `LOAD(`x`)`/`STORE(`x`)` replaced by `GET(`n`)`/`PUT(`n`)` (where $n \in \mathbb{N}$)
- `BRANCH` and `LOOP` instruction replaced by code addresses (labels) and jumping instructions
  - configurations $\langle pc, d, e, m\rangle$ with program counter $pc \in \mathbb{N}$
  - `BRANCH` and `LOOP` implemented by control flow, using `JUMP(`l`)` and `JUMPFALSE(`l`)` ($l \in \mathbb{N}$)

Alternative Choices

Remark: more traditional machine architectures

- Variables referenced by address (and not by name)
  - configurations $\langle d, e, m\rangle$ with memory $m \in \mathbb{Z}^*$
  - $\texttt{LOAD}(x)/\texttt{STORE}(x)$ replaced by $\texttt{GET}(n)/\texttt{PUT}(n)$ (where $n \in \mathbb{N}$)
- BRANCH and LOOP instruction replaced by code addresses (labels) and jumping instructions
  - configurations $\langle pc, d, e, m\rangle$ with program counter $pc \in \mathbb{N}$
  - BRANCH and LOOP implemented by control flow, using $\texttt{JUMP}(l)$ and $\texttt{JUMPFALSE}(l)$ ($l \in \mathbb{N}$)
- Registers for storing intermediate values (in place of evaluation stack e)

Terminating and Looping Computations

Definition 16.3 (AM computations)

- A finite computation is a finite configuration sequence of the form $\gamma_0, \gamma_1, \ldots, \gamma_k$ where $k \in \mathbb{N}$ and $\gamma_{i-1} \rhd \gamma_i$ for each $i \in \{1, \ldots, k\}$

Terminating and Looping Computations

Definition 16.3 (AM computations)

- A finite computation is a finite configuration sequence of the form $\gamma_0, \gamma_1, \ldots, \gamma_k$ where $k \in \mathbb{N}$ and $\gamma_{i-1} \rhd \gamma_i$ for each $i \in \{1, \ldots, k\}$
- If, in addition, there is no γ such that $\gamma_k \rhd \gamma$, then $\gamma_0, \gamma_1, \ldots, \gamma_k$ is called terminating

Terminating and Looping Computations

Definition 16.3 (AM computations)

- A finite computation is a finite configuration sequence of the form $\gamma_0, \gamma_1, \ldots, \gamma_k$ where $k \in \mathbb{N}$ and $\gamma_{i-1} \rhd \gamma_i$ for each $i \in \{1, \ldots, k\}$
- If, in addition, there is no γ such that $\gamma_k \rhd \gamma$, then $\gamma_0, \gamma_1, \ldots, \gamma_k$ is called terminating
- A looping computation is an infinite configuration sequence of the form $\gamma_0, \gamma_1, \gamma_2, \ldots$ where $\gamma_i \rhd \gamma_{i+1}$ for each $i \in \mathbb{N}$

Terminating and Looping Computations

Definition 16.3 (AM computations)

- A finite computation is a finite configuration sequence of the form $\gamma_0, \gamma_1, \ldots, \gamma_k$ where $k \in \mathbb{N}$ and $\gamma_{i-1} \rhd \gamma_i$ for each $i \in \{1, \ldots, k\}$
- If, in addition, there is no γ such that $\gamma_k \rhd \gamma$, then $\gamma_0, \gamma_1, \ldots, \gamma_k$ is called terminating
- A looping computation is an infinite configuration sequence of the form $\gamma_0, \gamma_1, \gamma_2, \ldots$ where $\gamma_i \rhd \gamma_{i+1}$ for each $i \in \mathbb{N}$

Note: a terminating computation may end in a final configuration ($\langle \varepsilon, e, \sigma \rangle$) or in a stuck configuration (e.g., $\langle \mathtt{ADD}, 1, \sigma \rangle$)

A Terminating Computation

Example 16.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$\langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle$$

A Terminating Computation

Example 16.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$\begin{aligned} & \langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\ \rhd \quad & \langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \end{aligned}$$

A Terminating Computation

Example 16.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$\begin{aligned}
&\langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\
\rhd\quad &\langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \\
\rhd\quad &\langle \texttt{ADD:STORE(x)}, 3 : 1, \sigma \rangle
\end{aligned}$$

A Terminating Computation

Example 16.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$
\begin{aligned}
& \langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\
\rhd \quad & \langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \\
\rhd \quad & \langle \texttt{ADD:STORE(x)}, 3 : 1, \sigma \rangle \\
\rhd \quad & \langle \texttt{STORE(x)}, 4, \sigma \rangle
\end{aligned}
$$

A Terminating Computation

Example 16.4

For $d := \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\texttt{x}) = 3$, we obtain the following terminating computation:

$$
\begin{aligned}
& \langle \texttt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\
\rhd \quad & \langle \texttt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \\
\rhd \quad & \langle \texttt{ADD:STORE(x)}, 3 : 1, \sigma \rangle \\
\rhd \quad & \langle \texttt{STORE(x)}, 4, \sigma \rangle \\
\rhd \quad & \langle \varepsilon, \varepsilon, \sigma[\texttt{x} \mapsto 4] \rangle
\end{aligned}
$$

A Terminating Computation

Example 16.4

For $d := \mathtt{PUSH(1):LOAD(x):ADD:STORE(x)}$ and $\sigma(\mathtt{x}) = 3$, we obtain the following terminating computation:

$$\begin{array}{ll} & \langle \mathtt{PUSH(1):LOAD(x):ADD:STORE(x)}, \varepsilon, \sigma \rangle \\ \rhd & \langle \mathtt{LOAD(x):ADD:STORE(x)}, 1, \sigma \rangle \\ \rhd & \langle \mathtt{ADD:STORE(x)}, 3 : 1, \sigma \rangle \\ \rhd & \langle \mathtt{STORE(x)}, 4, \sigma \rangle \\ \rhd & \langle \varepsilon, \varepsilon, \sigma[\mathtt{x} \mapsto 4] \rangle \end{array}$$

Remark: implements statement `x := x + 1`

A Looping Computation

Example 16.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$

A Looping Computation

Example 16.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma \rangle$$

A Looping Computation

Example 16.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma \rangle$$

A Looping Computation

Example 16.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma \rangle$$
$$\rhd \quad \langle \texttt{NOOP:LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$

A Looping Computation

Example 16.5

The following computation loops:

$$\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma \rangle$$
$$\rhd \quad \langle \texttt{NOOP:LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$
$$\rhd \quad \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle$$

A Looping Computation

Example 16.5

The following computation loops:

$$\begin{aligned}
&\langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle \\
&\quad \triangleright \quad \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma\rangle \\
&\quad \triangleright \quad \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma\rangle \\
&\quad \triangleright \quad \langle \texttt{NOOP:LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle \\
&\quad \triangleright \quad \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma\rangle \\
&\quad \triangleright \quad \ldots
\end{aligned}$$

A Looping Computation

Example 16.5

The following computation loops:

$$
\begin{array}{ll}
& \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle \\
\rhd & \langle \texttt{TRUE:BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \varepsilon, \sigma \rangle \\
\rhd & \langle \texttt{BRANCH(NOOP:LOOP(TRUE,NOOP),NOOP)}, \mathsf{true}, \sigma \rangle \\
\rhd & \langle \texttt{NOOP:LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle \\
\rhd & \langle \texttt{LOOP(TRUE,NOOP)}, \varepsilon, \sigma \rangle \\
\rhd & \ldots
\end{array}
$$

Remark: implements statement `while true do skip`

Outline

A New Inductive Principle

Application: Finite computations (Def. 16.3)

Definition: a finite computation $\gamma_0, \gamma_1, \ldots, \gamma_k$ has length k

Induction base: property holds for all computations of length 0

Induction hypothesis: property holds for all computations of length $\leq k$

Induction step: property holds for all computations of length $k + 1$

Application: Extension of Code and Stack

Lemma 16.6

If $\langle d_1, e_1, \sigma\rangle \rhd^* \langle d', e', \sigma'\rangle$, *then*

$$\langle d_1 : d_2, e_1 : e_2, \sigma\rangle \rhd^* \langle d' : d_2, e' : e_2, \sigma'\rangle$$

for every $d_2 \in Code$ *and* $e_2 \in Stk$.

Interpretation: both the code and the stack component can be extended without changing the behavior of the machine

Application: Extension of Code and Stack

Lemma 16.6

If $\langle d_1, e_1, \sigma\rangle \rhd^* \langle d', e', \sigma'\rangle$, *then*

$$\langle d_1 : d_2, e_1 : e_2, \sigma\rangle \rhd^* \langle d' : d_2, e' : e_2, \sigma'\rangle$$

for every $d_2 \in \mathit{Code}$ *and* $e_2 \in \mathit{Stk}$.

Interpretation: both the code and the stack component can be extended without changing the behavior of the machine

Proof.

by induction on the length of the computation
(on the board) □