Operational Semantics

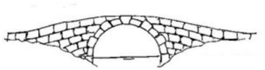
Simon Castellan
https://sicastel.gitlabpages.inria.fr/m2-sos

SOS, Master Recherche Science Informatique, U. Rennes 1

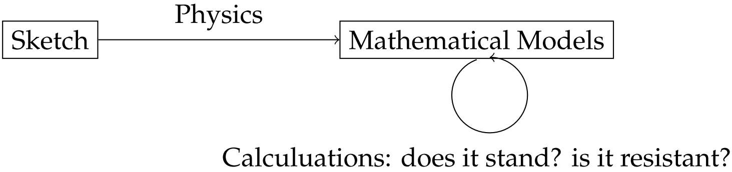
2021-2022

A mostly wrong history of bridge building

1. **Once upon a time:** bridge-building recipes



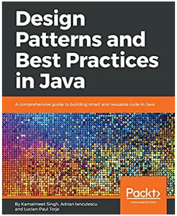
2. **Nowadays**: Maths and physics to the rescue.



- What is a mathematical representation of a bridge?
- How do you go from the sketch to the model?
- How do you check for safety (it stands)?
- How do you check against attacks (ie. different scenario)?

Meanwhile, in computer science

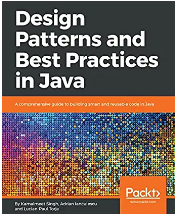
1. We have the recipes:



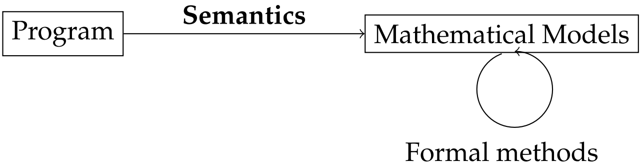
2. What about the fancy science?

Meanwhile, in computer science

1. We have the recipes:

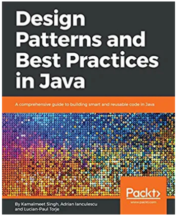


2. What about the fancy science? In its infancy!

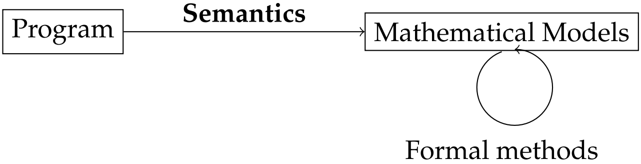


Meanwhile, in computer science

1. We have the recipes:



2. What about the fancy science? In its infancy!

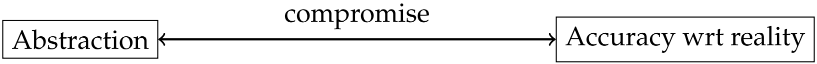


Semantics can make **formal properties** about programs:

$$\text{“The program } P \text{ computes factorial”} \rightsquigarrow \forall n \in \mathbb{N}, [\![P]\!](n) = n!.$$

The tradeoff of semantics

A model is a mathematical *point of view*: there are **many** models.



More abstract Easier to reason with, to prove properties.

More accurate Can state more properties.

Example 1 (Spectre)

Spectre is a recent attack using the branch prediction of processors.

(No current models cannot state the property: “This program is resistant to the Spectre attack”.)

Static analysis

Definition 2

A static analysis is an algorithm to check whether a program has a certain property.

Example: does my C program make access to uninitialised variables?

```
int sum(int []array, int len) {
  int sum;
  for(int i = 0; i < len; i ++)
    sum = sum + array[i]; // wrong: access to uninitialised sum
}
```

Static analysis and approximation

There is a gap between the algorithm $\mathscr{A}$ and the mathematical property $\mathscr{P}$.

A static analysis $\mathscr{A}$ is:

- **Sound** when if $\mathscr{A}$ says yes on a program p, then $\mathscr{P}$ holds on p.
- **Complete** when if $\mathscr{P}$ holds on p then $\mathscr{A}$ says yes.

Static analysis and approximation

There is a gap between the algorithm $\mathscr{A}$ and the mathematical property $\mathscr{P}$.

A static analysis $\mathscr{A}$ is:

- **Sound** when if $\mathscr{A}$ says yes on a program p, then $\mathscr{P}$ holds on p.
- **Complete** when if $\mathscr{P}$ holds on p then $\mathscr{A}$ says yes.

The grail: an algorithm $\mathscr{A}$ **sound and complete**.

Static analysis and approximation

There is a gap between the algorithm $\mathscr{A}$ and the mathematical property $\mathscr{P}$.

A static analysis $\mathscr{A}$ is:

- **Sound** when if $\mathscr{A}$ says yes on a program p, then $\mathscr{P}$ holds on p.
- **Complete** when if $\mathscr{P}$ holds on p then $\mathscr{A}$ says yes.

The grail: an algorithm $\mathscr{A}$ **sound and complete**.

Theorem 3 (Rice)

If the programming language is Turing-complete, and $\mathscr{P}$ is non-trivial, then there are no sound and complete algorithm for $\mathscr{P}$.

Static analysis and approximation

There is a gap between the algorithm $\mathscr{A}$ and the mathematical property $\mathscr{P}$.

A static analysis $\mathscr{A}$ is:

- **Sound** when if $\mathscr{A}$ says yes on a program p, then $\mathscr{P}$ holds on p.
- **Complete** when if $\mathscr{P}$ holds on p then $\mathscr{A}$ says yes.

The grail: an algorithm $\mathscr{A}$ **sound and complete**.

Theorem 3 (Rice)

If the programming language is Turing-complete, and $\mathscr{P}$ is non-trivial, then there are no sound and complete algorithm for $\mathscr{P}$.

$\rightsquigarrow$ In practice: **sound approximation**.

Illustration of Rice's theorem

Theorem 4 (Consequence of Rice's theorem)

There is no algorithm sound and complete for uninitialised accesses in C.

Illustration of Rice's theorem

Theorem 4 (Consequence of Rice's theorem)

There is no algorithm sound and complete for uninitialised accesses in C.

Proof.

We show that if there were such an algorithm $\mathscr{A}$ then we could decide whether a given piece of C code P terminates. This is impossible since C is Turing-complete.

Indeed, we can form the following C program

```c
int x; // choose x not occurring in P
// Insert P here
int y = x;
```

This program has an uninitialised access iff P terminates. □

This lecture (4h)

An illustration of the static analysis methodology:

1. We isolate a subset of C called *While*: “prototypical imperative language”
2. We formulate an algorithm that checks if a given program *may* perform uninitialised accessees
3. To show it is *sound*, we need to define mathematically the property of having no uninitialised accesses.
   $\rightsquigarrow$ We give a semantics to *While*
4. Using the semantics, we prove our algorithm sound.

Lecture 1: An introduction to verified static analysis

Detecting unitialised accesses in imperative programs

Outline

Idealising a language

Real life programming languages are **complex**.
$\rightsquigarrow$ The specification of the C language: 500 pages.

Idealising a language

Real life programming languages are **complex**.
$\leadsto$ The specification of the C language: 500 pages.

In semantics, we start a small set of features and grow over time.

Idealising a language

Real life programming languages are **complex**.
$\rightsquigarrow$ The specification of the C language: 500 pages.

In semantics, we start a small set of features and grow over time.

The standard starting point, **imperative programming**:

- Variables (of type `int`)
- Assignments of arithmetic expressions (involving variables) to variables
- Conditionals on boolean expressions derived from variables
- While statements

We create an **idealised language** that combines this features, *While*.

The *While* language

$n \in \mathbf{Num} \qquad x \in \mathbf{Var}$	integers and variables
$a \in \mathbf{Aexp}$	arithmetic expressions
$a \ ::= \ n \mid x \mid a_1 + a_2 \mid \ldots$	
$b \in \mathbf{Bexp}$	boolean expressions
$b \ ::= \ \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 > a_2$	
$\quad \mid \texttt{not}\ b \mid b_1\ \texttt{and}\ b_2 \mid \ldots$	
$c \in \mathbf{Cmd}$	commands (i.e. programs)
$c \ ::= \ x := a \mid \texttt{skip} \mid c_1\ ;\ c_2$	
$\quad \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ b\ \texttt{do}\ c$	

This definition is called a **BNF grammar**:

- Different syntactic categories
- Certain basic categories are assumed: numbers, variables.
- Valid programs are described by **abstract syntax trees** (ASTs);

Example 5 (Factorial in *While*)

```
result:=1; while n>1 do (result := n * result; n := n -1).
```

Outline

Overview of the algorithm

Our algorithm is a very simple case of **liveness analysis**.

- We say that a variable is **safe** when it has been initialised in the past.
- We build a *partial* function $\mathscr{S} : \mathbf{Cmd} \times \mathscr{P}(\mathbf{Var}) \rightharpoonup \mathscr{P}(\mathbf{Var})$

Overview of the algorithm

Our algorithm is a very simple case of **liveness analysis**.

- We say that a variable is **safe** when it has been initialised in the past.
- We build a *partial* function $\mathscr{S} : \mathbf{Cmd} \times \mathscr{P}(\mathbf{Var}) \rightharpoonup \mathscr{P}(\mathbf{Var})$:
  - If $\mathscr{S}(c, X)$ is undefined: c makes uninitialised accesses outside X
  - If $\mathscr{S}(c, X) = Y$: assuming variables in X are initialised, all accesses in c are initialised, and at the end variables in Y are initialised.

Overview of the algorithm

Our algorithm is a very simple case of **liveness analysis**.

- We say that a variable is **safe** when it has been initialised in the past.
- We build a *partial* function $\mathscr{S} : \mathbf{Cmd} \times \mathscr{P}(\mathbf{Var}) \rightharpoonup \mathscr{P}(\mathbf{Var})$:
  - If $\mathscr{S}(c, X)$ is undefined: c makes uninitialised accesses outside X
  - If $\mathscr{S}(c, X) = Y$: assuming variables in X are initialised, all accesses in c are initialised, and at the end variables in Y are initialised.
- This partial function is defined **by induction on** c.

Overview of the algorithm

Our algorithm is a very simple case of **liveness analysis**.

- We say that a variable is **safe** when it has been initialised in the past.
- We build a *partial* function $\mathscr{S} : \mathbf{Cmd} \times \mathscr{P}(\mathbf{Var}) \rightharpoonup \mathscr{P}(\mathbf{Var})$:
  - If $\mathscr{S}(c, X)$ is undefined: c makes uninitialised accesses outside X
  - If $\mathscr{S}(c, X) = Y$: assuming variables in X are initialised, all accesses in c are initialised, and at the end variables in Y are initialised.
- This partial function is defined **by induction on** c.
- We use notations:

$$\begin{array}{rcl} \{X\}c\{Y\} & \text{means} & \mathscr{S}(c, X) = Y \\ \{X\}c\{\bot\} & \text{means} & S(c, X) \text{ undef.} \end{array}$$

$$\begin{array}{rcl} \mathsf{var} & : & (\mathbf{Bexp} \cup \mathbf{Aexp}) \to \mathscr{P}(\mathbf{Var}) \\ \mathsf{var}(x) & := & \{x\} \\ \mathsf{var}(n) & := & \emptyset \\ \mathsf{var}(e_1 + e_2) & := & \mathsf{var}(e_1) \cup \mathtt{var}(e_2) \end{array}$$

Static analysis: definition of $\mathscr{S}$ (1)

We now give the cases for the inductive definition of $\mathscr{S}$

$$\mathscr{S}(\,\texttt{skip}\,, X) \quad := \quad X$$

Static analysis: definition of $\mathscr{S}$ (1)

We now give the cases for the inductive definition of $\mathscr{S}$

$$\mathscr{S}(\,\texttt{skip}\,, X) \quad := \quad X$$

$$\mathscr{S}(x := e, X) \quad := \quad \begin{cases} X \cup \{x\} & \text{if } \texttt{var}(e) \subseteq X \\ \text{undef} & \text{otherwise} \end{cases}$$

Static analysis: definition of $\mathscr{S}$ (1)

We now give the cases for the inductive definition of $\mathscr{S}$

$$\mathscr{S}(\,\texttt{skip}\,,X) \quad := \quad X$$

$$\mathscr{S}(x := e, X) \quad := \quad \begin{cases} X \cup \{x\} & \text{if } \texttt{var}(e) \subseteq X \\ \text{undef} & \text{otherwise} \end{cases}$$

$$\mathscr{S}(c; c', X) \quad := \quad \begin{cases} Z & \{X\}c\{Y\} \wedge \{Y\}c'\{Z\} \\ \text{undef} & \text{otherwise} \end{cases}$$

Static analysis: definition of $\mathscr{S}$ (1)

We now give the cases for the inductive definition of $\mathscr{S}$

$$\mathscr{S}(\,\texttt{skip}\,, X) \quad := \quad X$$

$$\mathscr{S}(x := e, X) \quad := \quad \begin{cases} X \cup \{x\} & \text{if } \texttt{var}(e) \subseteq X \\ \text{undef} & \text{otherwise} \end{cases}$$

$$\mathscr{S}(c; c', X) \quad := \quad \begin{cases} Z & \{X\}c\{Y\} \wedge \{Y\}c'\{Z\} \\ \text{undef} & \text{otherwise} \end{cases}$$

Static analysis: definition of $\mathscr{S}$ (2)

$$\mathscr{S}(\,\texttt{if}\; b\; \texttt{then}\; c_1\; \texttt{else}\; c_2, X) \quad := \quad \begin{cases} Y_1 \cap Y_2 & \mathsf{var}(b) \subseteq X \\ & \wedge \{X\}c_1\{Y_1\} \wedge \{X\}c_2\{Y_2\} \\ \text{undef} & \text{otherwise} \end{cases}$$

Static analysis: definition of $\mathscr{S}$ (2)

$$\mathscr{S}(\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, X) \quad := \quad \begin{cases} Y_1 \cap Y_2 & \mathsf{var}(b) \subseteq X \\ & \wedge \{X\}c_1\{Y_1\} \wedge \{X\}c_2\{Y_2\} \\ \text{undef} & \text{otherwise} \end{cases}$$

$$\mathscr{S}(\ \texttt{while}\ b\ \texttt{do}\ c, X) \quad := \quad \begin{cases} X & \mathsf{var}(b) \subseteq X \wedge \{X\}c\{Y\} \\ \text{undef} & \text{otherwise} \end{cases}$$

Static analysis: definition of $\mathscr{S}$ (2)

$$\mathscr{S}(\,\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, X) \quad := \quad \begin{cases} Y_1 \cap Y_2 & \mathsf{var}(b) \subseteq X \\ & \wedge \{X\}c_1\{Y_1\} \wedge \{X\}c_2\{Y_2\} \\ \text{undef} & \text{otherwise} \end{cases}$$

$$\mathscr{S}(\,\texttt{while}\ b\ \texttt{do}\ c, X) \quad := \quad \begin{cases} X & \mathsf{var}(b) \subseteq X \wedge \{X\}c\{Y\} \\ \text{undef} & \text{otherwise} \end{cases}$$

```
x = 10;
while(x > 0) { y = y + 1; x = x - 1 }
```

Static analysis: definition of $\mathscr{S}$ (2)

$$\mathscr{S}(\,\texttt{if}\; b \;\texttt{then}\; c_1 \;\texttt{else}\; c_2, X) \quad := \quad \begin{cases} Y_1 \cap Y_2 & \mathsf{var}(b) \subseteq X \\ & \wedge \{X\} c_1 \{Y_1\} \wedge \{X\} c_2 \{Y_2\} \\ \text{undef} & \text{otherwise} \end{cases}$$

$$\mathscr{S}(\,\texttt{while}\; b \;\texttt{do}\; c, X) \quad := \quad \begin{cases} X & \mathsf{var}(b) \subseteq X \wedge \{X\} c \{Y\} \\ \text{undef} & \text{otherwise} \end{cases}$$

```
x = 10;
while(x > 0) { y = y + 1; x = x - 1 }
```

```
x = 10;
while(x > 0) { y = 1; x = x - 1 }
```

Static analysis: examples

- Our algorithm works on some examples: $\{\emptyset\}x := 1; z := y\{\bot\}$
- However, our formula for the conditional is an **approximation**

$$\{\emptyset\}\,\texttt{if true then}\,x := 1\,\texttt{else}\,x := y\{\bot\}$$

  however, this program does not do any uninitialised accesses since the else branch is never taken.
- To improve the analysis, we need to be able to guess better if a certain branch will never be taken.

Static analysis: examples

- Our algorithm works on some examples: $\{\emptyset\} x := 1; z := y \{\bot\}$
- However, our formula for the conditional is an **approximation**

$$\{\emptyset\}\ \texttt{if true then}\ x := 1\ \texttt{else}\ x := y \{\bot\}$$

  however, this program does not do any uninitialised accesses since the else branch is never taken.
- To improve the analysis, we need to be able to guess better if a certain branch will never be taken.
- We now would like to show this analysis is sound.
  $\rightsquigarrow$ How to formalise the property of having no uninitialised access?

Static analysis: examples

- Our algorithm works on some examples: $\{\emptyset\}x := 1; z := y\{\bot\}$
- However, our formula for the conditional is an **approximation**

$$\{\emptyset\}\ \texttt{if true then}\ x := 1\ \texttt{else}\ x := y\{\bot\}$$

  however, this program does not do any uninitialised accesses since the else branch is never taken.
- To improve the analysis, we need to be able to guess better if a certain branch will never be taken.
- We now would like to show this analysis is sound.
  $\rightsquigarrow$ How to formalise the property of having no uninitialised access?

For this,we need to define the semantics of *While*, i.e. how programs are executed.

Outline

Different semantics of *While*

There are several ways of expressing the mathematical behaviour of a program, say `fact` :=(`result` := 1; `while`($n > 0$) `do` `result` := n * result; `n--`).

Informally:

1. `fact` always terminates
2. If `n` is equal to $k \in \mathbb{N}$ before the execution, at the end `result` $= k$!

Different semantics of *While*

There are several ways of expressing the mathematical behaviour of a program, say `fact` :=(`result` := 1; `while`($n > 0$) `do` `result` := n * result; `n--`).

Informally:

1. `fact` always terminates
2. If `n` is equal to $k \in \mathbb{N}$ before the execution, at the end `result` $= k!$

Different families of models based on **memory states** $\sigma \in \mathbf{Var} \to \mathbb{Z}$:

- **Denotational**: programs become functions on memory states.

$$[\![P]\!] = \sigma \mapsto \sigma[\mathtt{n} \mapsto 1, \mathtt{result} \mapsto \sigma(\mathtt{n})!]$$

Different semantics of *While*

There are several ways of expressing the mathematical behaviour of a program, say `fact` :=(`result` := 1; `while`($n > 0$) `do result` := n * result; `n--`).

Informally:

1. `fact` always terminates
2. If `n` is equal to $k \in \mathbb{N}$ before the execution, at the end `result` $= k!$

Different families of models based on **memory states** $\sigma \in \mathbf{Var} \to \mathbb{Z}$:

- **Denotational**: programs become functions on memory states.

$$[\![P]\!] = \sigma \mapsto \sigma[\mathtt{n} \mapsto 1, \mathtt{result} \mapsto \sigma(\mathtt{n})!]$$

- **Axiomatic**: programs become memory predicate transformers.

$$\{\mathtt{n} = k\}P\{\mathtt{n} = 1 \ \mathbf{and}\ \mathtt{result} = k!\}$$

Different semantics of *While*

There are several ways of expressing the mathematical behaviour of a program, say `fact` :=(`result` := 1; `while`($n > 0$) `do` `result` := n * result; `n--`).

Informally:

1. `fact` always terminates
2. If `n` is equal to $k \in \mathbb{N}$ before the execution, at the end `result` $= k!$

Different families of models based on **memory states** $\sigma \in \mathbf{Var} \to \mathbb{Z}$:

- **Denotational**: programs become functions on memory states.

$$[\![P]\!] = \sigma \mapsto \sigma[\mathtt{n} \mapsto 1, \mathtt{result} \mapsto \sigma(\mathtt{n})!]$$

- **Axiomatic**: programs become memory predicate transformers.

$$\{\mathtt{n} = k\}P\{\mathtt{n} = 1 \textbf{ and } \mathtt{result} = k!\}$$

- **Operational**: programs are executed by a mathematical machine.

$$(P, \sigma) \to^* \sigma[\mathtt{n} \mapsto 1, \mathtt{result} \mapsto \sigma(\mathtt{n})!]$$

Different semantics of *While*

There are several ways of expressing the mathematical behaviour of a program, say `fact` :=(`result` := 1; `while`($n > 0$) `do` `result` := n * result; `n--`).

Informally:

1. `fact` always terminates
2. If `n` is equal to $k \in \mathbb{N}$ before the execution, at the end $\texttt{result} = k!$

Different families of models based on **memory states** $\sigma \in \mathbf{Var} \to \mathbb{Z}$:

- **Denotational**: programs become functions on memory states.

$$[\![P]\!] = \sigma \mapsto \sigma[\texttt{n} \mapsto 1, \texttt{result} \mapsto \sigma(\texttt{n})!]$$

- **Axiomatic**: programs become memory predicate transformers.

$$\{\texttt{n} = k\}P\{\texttt{n} = 1 \textbf{ and } \texttt{result} = k!\}$$

- **Operational**: programs are executed by a mathematical machine.

$$(P, \sigma) \to^* \sigma[\texttt{n} \mapsto 1, \texttt{result} \mapsto \sigma(\texttt{n})!]$$

Different semantics of *While*

There are several ways of expressing the mathematical behaviour of a program, say `fact` :=(`result` := 1; `while`($n > 0$) `do` `result` := n * result; `n--`).

Informally:

1. `fact` always terminates
2. If `n` is equal to $k \in \mathbb{N}$ before the execution, at the end `result` $= k!$

Different families of models based on **memory states** $\sigma \in \mathbf{Var} \to \mathbb{Z}$:

- **Denotational**: programs become functions on memory states.

$$[\![P]\!] = \sigma \mapsto \sigma[\mathtt{n} \mapsto 1, \mathtt{result} \mapsto \sigma(\mathtt{n})!]$$

- **Axiomatic**: programs become memory predicate transformers.

$$\{\mathtt{n} = k\}P\{\mathtt{n} = 1 \textbf{ and } \mathtt{result} = k!\}$$

- **Operational**: programs are executed by a mathematical machine.

$$(P, \sigma) \to^* \sigma[\mathtt{n} \mapsto 1, \mathtt{result} \mapsto \sigma(\mathtt{n})!]$$

Operational semantics: References

References

- G. Winskel, *The Formal Semantics of Programming Languages* MIT Press, 1993 (chapters 2 and 3)
- H.R. Nielson and F. Nielson, *Semantics with Applications - A Formal Introduction*, Wiley 1992. (chapter 2)
- G. Plotkin, *A Structural Approach to Operational Semantics*, Technical Report, Aarhus University, 1981.
- G. Kahn, *Natural Semantics*, In Proc. of the Symposium on Theoretical Aspects of Computer Science, LNCS 247, pp. 22–39, Springer-Verlag, 1987.

Interpretation of values

Define the sets and functions used to describe the meaning of expressions.

- Values : integers and booleans

$$\mathbb{Z} \qquad\qquad \mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$$

- Interpretation function for constants

$$\mathcal{N} \in \mathbf{Num} \to \mathbb{Z}$$

- A memory state or environment, σ maps variables to values

$$\sigma \in \mathbf{State} = \mathbf{Var} \to \mathbb{Z}$$

Reading the value of variable x in σ $\qquad \sigma(x)$
Updating σ by setting a new value v for x $\qquad \sigma' = \sigma[x \mapsto v]$

- Semantics of expressions

$$\mathcal{A} \in \mathbf{Aexp} \to \mathbf{State} \to \mathbb{Z} \qquad\qquad \mathcal{B} \in \mathbf{Bexp} \to \mathbf{State} \to \mathbb{B}$$

Expressions denote functions from states to integer values.
Notation: $\mathcal{A}(a)(\sigma)$ traditionally written $\mathcal{A}[\![a]\!]\sigma$.

Arithmetic expressions

Remember: The set of arithmetic expressions is defined **inductively**

$$a \in \textbf{Aexp} ::= n \mid x \mid a_1 + a_2 \mid \ldots$$

$\mathcal{A}[\![.]\!]$ is defined by induction on the definition of **Aexp**, following the structure of expressions.

$$\begin{array}{lcl} \mathcal{A} \in \textbf{Aexp} \to \textbf{State} \to \mathbb{Z} & & \\ \mathcal{A}[\![n]\!]\sigma & = & \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![x]\!]\sigma & = & \sigma(x) \\ \mathcal{A}[\![a_1 + a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma + \mathcal{A}[\![a_2]\!]\sigma \end{array}$$

Note: + is the syntactic operator, $+$ is the sum operator defined on integers.

The semantics is **compositional**: *the meaning of a syntactic construction is defined from the meaning of its constituent parts.*

Boolean expressions

Similarly, define $\mathcal{B}[\![.]\!]$ by induction on the definition of **Bexp**.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{B} \in \textbf{Bexp} \to \textbf{State} \to \mathbb{B}} \\
\mathcal{B}[\![\, \texttt{true} \,]\!]\sigma & = & \textbf{tt} \\
\mathcal{B}[\![\, \texttt{false} \,]\!]\sigma & = & \textbf{ff} \\
\mathcal{B}[\![a_1 \texttt{=} a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma = \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![a_1 \texttt{<} a_2]\!]\sigma & = & \mathcal{A}[\![a_1]\!]\sigma < \mathcal{A}[\![a_2]\!]\sigma \\
\mathcal{B}[\![\, \texttt{not}\ b]\!]\sigma & = & \neg(\mathcal{B}[\![b]\!]\sigma) \\
\mathcal{B}[\![b_1 \ \texttt{and}\ b_2]\!]\sigma & = & \mathcal{B}[\![b_1]\!]\sigma \wedge \mathcal{B}[\![b_2]\!]\sigma
\end{array}
$$

where $\neg$, $\wedge$, $=$ are operators defined on booleans and integers.

Proof technique

The set of arithmetic expressions **Aexp** is defined **inductively**

$$a ::= n \mid x \mid a_1 + a_2 \mid \ldots$$

Structural induction

To prove a property $\mathcal{P}$ of all arithmetic expressions:

1. Base cases: show the property for each atomic expression
2. Inductive cases : show the property for each composite expression, under the hypothesis that it holds on its constituent parts.

Formally, the induction principle for arithmetic expressions is :

$$\left.\begin{array}{ll} & \forall n \in \mathbf{Num}, \mathcal{P}(n) \\ \wedge & \forall x \in \mathbf{Var}, \mathcal{P}(x) \\ \wedge & \forall a_1, a_2 \in \mathbf{Aexp}, \mathcal{P}(a_1) \wedge \mathcal{P}(a_2) \Rightarrow \mathcal{P}(a_1 + a_2) \end{array}\right\} \Rightarrow \forall a \in \mathbf{Aexp}, \mathcal{P}(a)$$

Vocabulary: the above $\mathcal{P}(a_1)$ and $\mathcal{P}(a_2)$ are called the **induction hypotheses**.

Exercises

Exercise 3.1

Let $\sigma \in \textbf{State}$ *and* $x \in \textbf{Var}$ *such that* $\sigma(x) = 3$*. Show that* $\mathcal{B}[\![\textbf{not}(\textbf{x} = \textbf{1})]\!]\sigma = \textbf{tt}$.

Exercise 3.2

We extend the language **Aexp** *with the unary minus operator and the construction* $-a$*. Extend the semantic function* $\mathcal{A}$ *to give a compositional semantics for this construction.*

Exercise 3.3

We extend the language **Bexp** *with the construction* b_1 `or` b_2*.*

- *Extend the semantic function* $\mathcal{B}$ *to give a compositional semantics for this construction.*
- *Prove that for all* $\bar{b}$ *belonging to the extended language there exists a* b *belonging to the original language such that:*

$$\mathcal{B}[\![b]\!] = \mathcal{B}[\![\bar{b}]\!]$$

Operational semantics: Transition systems

Describe how the execution of *While* programs is done, operationally.

The operational semantics of a language is defined by an abstract machine, formalised as a **transition system**.

Transition system

A transition system is a triple $(\Gamma, T, \rightsquigarrow)$ where

- Γ is a set of **configurations** (states of the machine)
- $T \subseteq \Gamma$ is a set of **final** configurations
- $\rightsquigarrow \subseteq \Gamma \times \Gamma$ is a **transition relation**

Two main styles of definitions for the transition relation:

- **Small-step semantics** *Structural Operational Semantics* (SOS)
  Relation $\rightarrow$ describes all intermediate, individual steps
- **Big-step semantics** *Natural semantics* (NS)
  Relation $\Downarrow$ describes how to obtain the final result of computation

Transition systems: some definitions

Transition system

A transition system is a triple $(\Gamma, T, \rightsquigarrow)$ where

- Γ is a set of **configurations**
- $T \subseteq \Gamma$ is a set of **final** configurations
- $\rightsquigarrow \subseteq \Gamma \times \Gamma$ is a **transition relation**

A transition system $(\Gamma, T, \rightsquigarrow)$ is said

- **deterministic** when relation $\rightsquigarrow$ is functional

$$\gamma \rightsquigarrow \gamma_1 \text{ and } \gamma \rightsquigarrow \gamma_2 \text{ implies } \gamma_1 = \gamma_2$$

- **non-blocking** when relation $\rightsquigarrow$ is total on $\Gamma \setminus T$

$$\text{for all } \gamma \in \Gamma \setminus T, \text{ there exists } \gamma' \text{ such that } \gamma \rightsquigarrow \gamma'$$

The notion of program **execution** will be defined on top of $\rightsquigarrow$.

Transition systems for *While*: configurations

To run a *While* program, we need a command $c \in \mathbf{Cmd}$, and a state $\sigma \in \mathbf{State}$.

For *While*, configurations are defined:

- $\Gamma = \{(c, \sigma) \mid c \in \mathbf{Cmd}, \sigma \in \mathbf{State}\} \cup \mathbf{State}$
- Final configurations : $T = \mathbf{State}$

So, either :

- $(c, \sigma) \rightsquigarrow (c', \sigma')$
  execution of c has not terminated, and (c', σ') is left to execute
- or $(c, \sigma) \rightsquigarrow \sigma'$
  execution of c has terminated in the final configuration σ'

Next slides: define two transition relations, following the structure of *While* commands

Small-step transition relation

$$\textbf{Cmd} \ni c ::= x \texttt{ := } a \mid \texttt{skip} \mid c_1 \texttt{ ; } c_2$$
$$\mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c$$

Easy for atomic commands:

- Executing skip terminates in 1 step and doesn't modify the state.
  For all possible σ, we have $(\texttt{skip}, \sigma) \to \sigma$
- Executing an assignment terminates in 1 step, and updates the state.
  For all possible σ, x, and a, we have $(x \texttt{ := } a, \sigma) \to \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$

For compound commands, like sequences ?

$$(c_1 \texttt{ ; } c_2, \sigma) \to ???$$

Intuitively, we have to first execute c_1 in small-step.

- The transition relation needs to be defined inductively!

Small-step transition relation, inductively

Inductively defined relations are usually formalised by a **rule system**.

A rule is of the form :

$$\textsc{RuleName} \qquad \textit{if} \ldots (\textit{side conditions}) \; \frac{\textit{premise}_1 \quad \ldots \quad \textit{premise}_n}{\text{conclusion}}$$

where $premise_i$ and *conclusion* are elements of the relation being defined.[1]

It reads: "If $premise_1$ and $\ldots premise_n$, and if side conditions are satisfied, then *conclusion*". Premises must be, in turn, justified by rules.

- the conclusion holds whenever there is a **finite derivation tree** whose leaves are axioms of the system.

For the transition relation $\rightsquigarrow$, rules are of the form:

$$\textsc{RuleName} \qquad \textit{if} \ldots (\textit{side conditions}) \; \frac{\gamma_0 \rightsquigarrow \gamma_0' \quad \ldots \quad \gamma_i \rightsquigarrow \gamma_i'}{\gamma_j \rightsquigarrow \gamma_j'}$$

[1] A rule with no premise is called an axiom.

Structural operational semantics (SOS)

Rule system defining the small-step transition relation.

Precisely: these are rule **schemas**, to be instantiated on particular commands and states.

$$\text{ASSIG}\ \frac{}{(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]}$$

$$\text{SKIP}\ \frac{}{(\texttt{skip}, \sigma) \to \sigma}$$

$$\text{SEQ1}\ \frac{(c_1, \sigma) \to \sigma'}{(c_1 \texttt{;}\ c_2, \sigma) \to (c_2, \sigma')}$$

$$\text{SEQ2}\ \frac{(c_1, \sigma) \to (c_1', \sigma')}{(c_1 \texttt{;}\ c_2, \sigma) \to (c_1' \texttt{;}\ c_2, \sigma')}$$

$$\text{IFT}\quad \textit{if}\ \mathcal{B}[\![b]\!]\sigma = \mathbf{tt}\ \frac{}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma) \to (c_1, \sigma)}$$

$$\text{IFE}\quad \textit{if}\ \mathcal{B}[\![b]\!]\sigma = \mathbf{ff}\ \frac{}{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma) \to (c_2, \sigma)}$$

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

Small-step executions and semantics

A small-step **execution** of a *While* command is a sequence of configurations

$$\gamma_0, \ldots, \gamma_p, \ldots \text{ such that, for each } i, \gamma_i \to \gamma_{i+1}$$

We write :

$\to^*$ Reflexive and transitive closure of $\to$: finite number of transitions

$\to^+$ Transitive closure of $\to$: finite, non-zero number of transitions

$\to^i$ Exactly i transitions

Execution of (c, σ) is said

- to **terminate** iff there exists σ' such that $(c, \sigma) \to^* \sigma'$
- to **loop** iff there exists an infinite transition sequence starting from (c, σ)

Exercises

Exercise 3.4 (In class)

Show that for all σ *with* $\sigma(n) \geqslant 1$*:*

$$(P, \sigma) \to^* \sigma'$$

with $\sigma'(\texttt{result}) = \sigma(n)!$ *where* P *is the factorial program.*

Exercise 3.5 (At home)

*Give an SOS to the arithmetic expressions (***Aexp***) of the While language. Is your corresponding transition system deterministic? Explain why.*

Proving our static analysis correct

We want to show:

Theorem 6

If our algorithm says yes, then the program does not access unitialised references.

How to formalise:

1. "If our algorithm says yes"
2. "The program does not access unitialised references."

Proving our static analysis correct

We want to show:

Theorem 6

If our algorithm says yes, then the program does not access unitialised references.

How to formalise:

1. "If our algorithm says yes"
2. "The program does not access unitialised references."

Proving our static analysis correct

We want to show:

Theorem 6

If our algorithm says yes, then the program does not access unitialised references.

How to formalise:

1. "If our algorithm says yes"

$$\mathscr{S}(P, \emptyset) = Y$$

2. "The program does not access unitialised references."

Proving our static analysis correct

We want to show:

Theorem 6

If our algorithm says yes, then the program does not access unitialised references.

How to formalise:

1. "If our algorithm says yes"

$$\mathscr{S}(P, \emptyset) = Y$$

2. "The program does not access unitialised references."

$$(P, \sigma_1) \to^* \sigma_1' \wedge (P, \sigma_2) \to^* \sigma_2' \quad \Rightarrow \quad \sigma_1'|_Y = \sigma_2'|_Y$$

The final memory state does not depend on the input state.

Proving our static analysis correct

We want to show:

Theorem 6

If our algorithm says yes, then the program does not access unitialised references.

How to formalise:

1. "If our algorithm says yes"

$$\mathscr{S}(P, \emptyset) = Y$$

2. "The program does not access unitialised references."

$$(P, \sigma_1) \to^* \sigma_1' \wedge (P, \sigma_2) \to^* \sigma_2' \quad \Rightarrow \quad \sigma_1'|_Y = \sigma_2'|_Y$$

The final memory state does not depend on the input state.

$\leadsto$ Problem: reasoning on $\to^*$ is tedious. Define the final state directly?

Outline

Forgetting the intermediate steps

Our semantics allows to view commands as state transformers:

Definition 7

Command c turns state σ into state σ' when $(c, \sigma) \rightarrow^* \sigma'$.
We write $\langle c, \sigma \rangle \Downarrow \sigma'$.

Can we define the relation $\langle c, \sigma \rangle \Downarrow \sigma'$ directly (by induction)?

$\leadsto$ Yes: it is called **natural semantics**.

Natural (or big-step) semantics (NS)

Rule system defining the big-step transition relation.

Focuses on final state reached: no elementary computation step described. So, the transition relation is such that $\Downarrow \subseteq (\textbf{Cmd} \times \textbf{State}) \times \textbf{State} \subseteq \Gamma \times T$

$$\text{ASSIG}\ \frac{}{(x := a, \sigma) \Downarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]} \qquad \text{SKIP}\ \frac{}{(\texttt{skip}, \sigma) \Downarrow \sigma}$$

$$\text{SEQ}\ \frac{(S_1, \sigma) \Downarrow \sigma' \qquad (S_2, \sigma') \Downarrow \sigma''}{(S_1\ ;\ S_2, \sigma) \Downarrow \sigma''}$$

$$\text{IFT} \qquad \textit{if}\ \mathcal{B}[\![b]\!]\sigma = \textbf{tt}\ \frac{(S_1, \sigma) \Downarrow \sigma'}{(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma) \Downarrow \sigma'}$$

$$\text{IFE} \qquad \textit{if}\ \mathcal{B}[\![b]\!]\sigma = \textbf{ff}\ \frac{(S_2, \sigma) \Downarrow \sigma'}{(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma) \Downarrow \sigma'}$$

$$\text{WHI1} \qquad \textit{if}\ \mathcal{B}[\![b]\!]\sigma = \textbf{tt}\ \frac{(S, \sigma) \Downarrow \sigma' \qquad (\texttt{while}\ b\ \texttt{do}\ S, \sigma') \Downarrow \sigma''}{(\texttt{while}\ b\ \texttt{do}\ S, \sigma) \Downarrow \sigma''}$$

$$\text{WHI2} \qquad \textit{if}\ \mathcal{B}[\![b]\!]\sigma = \textbf{ff}\ \frac{}{(\texttt{while}\ b\ \texttt{do}\ S, \sigma) \Downarrow \sigma}$$

Big-step executions and semantics

A big-step execution of a *While* command is simply a **derivable** $(c, \sigma) \Downarrow \sigma'$

Execution of (c, σ) is said

- to **terminate** iff there exists σ' such that $(c, \sigma) \Downarrow \sigma'$
- to **loop/block** iff there is no state σ' such that $(c, \sigma) \Downarrow \sigma'$

Semantics of commands: partial function $\mathcal{S}_{nat} \in \mathbf{Cmd} \to \mathbf{State} \hookrightarrow \mathbf{State}$

$$\mathcal{S}_{nat}[\![c]\!]\sigma = \sigma' \qquad \textit{if } (c, \sigma) \Downarrow \sigma'$$

Commands c_1 and c_2 are **semantically equivalent** iff

$$\forall \sigma, \sigma'. \ (c_1, \sigma) \Downarrow \sigma' \Leftrightarrow (c_2, \sigma) \Downarrow \sigma'$$

Exercises

Exercise 4.1

Show that the NS semantics of the factorial program gives the expected behaviour.

Exercise 4.2 (At home)

The While language is extended with the construction `repeat` S `until` b. *Extend the NS accordingly.*

Proof technique associated with NS

Induction principle for derivation trees

$$\dfrac{\dfrac{\vdots}{P_1} \qquad \dfrac{}{P_2}}{P}$$

1. Prove the property for the the axioms of the rule system
2. For each rule, prove the property for the conclusion of the rule, **under the hypothesis** that the property holds for each of the premises, and that side conditions are satisfied.

Intuition: the property is proved:

- to hold for the leaves of the tree,
- and to propagate to any possible derivable conclusion.

Exercises

Exercise 4.3 (At home)

Prove that the NS of While is deterministic.

Exercise 4.4 (At home)

Prove that c_1 ; $(c_2$; $c_3)$ *and* $(c_1$; $c_2)$; c_3 *are semantically equivalent.*
Hint: induction is not necessary here.

Exercise 4.5 (At home ⋆)

Prove that

$$\texttt{while}\ b\ \texttt{do}\ c$$

and

$$\texttt{if}\ b\ \texttt{then}\ (c\ \texttt{;}\ \texttt{while}\ b\ \texttt{do}\ c)\ \texttt{else skip}$$

are semantically equivalent.
Hint: induction is not necessary here.

An equivalence of two semantics

Theorem

For all c and all σ, we have $\langle c, \sigma\rangle \to^* \sigma'$ *iff* $\langle c, \sigma\rangle \Downarrow \sigma'$.

The theorem is a direct consequence of the following two lemmas:

Lemma 8

For all command c and states σ, σ'

$$(c, \sigma) \Downarrow \sigma' \Rightarrow (c, \sigma) \to^* \sigma'$$

Lemma 9

For all command c and states σ, σ'

$$(c, \sigma) \to^k \sigma' \Rightarrow (c, \sigma) \Downarrow \sigma'$$

Proof of Lemma 8

Goal: for all command c and states σ, σ' $(c, \sigma) \Downarrow \sigma' \Rightarrow (c, \sigma) \to^* \sigma'$.

By induction on the derivation tree of $(S, \sigma) \Downarrow \sigma'$.

Case $(x := a, \sigma) \Downarrow \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$

Immediate from the SOS axiom $(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]$

Case

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

Thus

$(S_1, \sigma) \to^* \sigma'$ and $(S_2, \sigma') \to^* \sigma''$ *(by induction hypothesis)*

$(S_1 \;;\; S_2, \sigma) \to^* (S_2, \sigma')$

$(S_1 \;;\; S_2, \sigma) \to^* \sigma''$ *(by composition of transition sequences)*

Proof of Lemma 8

Case

$$\text{WHI1} \qquad b/c\ \mathcal{B}[\![b]\!]\sigma = \mathbf{tt}\ \frac{(S,\sigma) \Downarrow \sigma' \qquad (\texttt{while}\ b\ \texttt{do}\ S, \sigma') \Downarrow \sigma''}{(\texttt{while}\ b\ \texttt{do}\ S, \sigma) \Downarrow \sigma''}$$

The induction hypothesis gives us that

$$(S,\sigma) \to^* \sigma' \qquad \text{and} \qquad (\texttt{while}\ b\ \texttt{do}\ S, \sigma') \to^* \sigma''$$

According to the SOS, we have the following derivation:

$$\begin{aligned}(\texttt{while}\ b\ \texttt{do}\ S, \sigma) &\to (\texttt{if}\ b\ \texttt{then}\ (S\ \texttt{;}\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}, \sigma) \\ &\to (S\ \texttt{;}\ \texttt{while}\ b\ \texttt{do}\ S, \sigma)\end{aligned}$$

Composing the transition sequences, we obtain

$$(\texttt{while}\ b\ \texttt{do}\ S, \sigma) \to^* \sigma''$$

Other cases same idea (exercise)

Proof of Lemma 9

Goal: for all S, σ, k, σ', $(S, \sigma) \rightarrow^k \sigma' \Rightarrow (S, \sigma) \Downarrow \sigma'$.

Proceed by induction on the length of the transition sequence of $(S, \sigma) \rightarrow^k \sigma'$:

- If $k = 0$ then (S, σ) and σ' should be identical. Vacuously holds.
- Otherwise, suppose the lemma holds for all $k \leqslant k_0$ and prove it for a sequence of length $k_0 + 1$.
  We proceed by case analysis on the command S :

  Case $x := a$. This command reduces in one step to a final state (so $k_0 = 0$) by SOS axiom ASSIG. Result then follows from NS axiom ASSIG.

Proof of Lemma 9

Case $(S_1 \,;\, S_2, \sigma) \rightarrow^{k_0+1} \sigma''$

There exists k_1 and k_2 such that

$$(S_1, \sigma) \rightarrow^{k_1} \sigma' \quad \text{and} \quad (S_2, \sigma') \rightarrow^{k_2} \sigma'' \quad \text{with} \quad k_1 + k_2 = k_0 + 1$$

By induction hypothesis,

$$(S_1, \sigma) \Downarrow \sigma' \quad \text{and} \quad (S_2, \sigma') \Downarrow \sigma''$$

By the NS rule SEQ, we conclude that $(S_1 \,;\, S_2, \sigma) \Downarrow \sigma''$.

Case $(\,\texttt{while}\; b\; \texttt{do}\; S, \sigma)$
$\rightarrow (\,\texttt{if}\; b\; \texttt{then}\; (S \,;\, \texttt{while}\; b\; \texttt{do}\; S)\; \texttt{else skip}\,, \sigma) \quad \rightarrow^{k_0} \sigma''$

From the induction hypothesis, we get

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

In Exercise 4.5, we proved this command semantically equivalent to $\texttt{while}\; b\; \texttt{do}\; S$, hence $(\,\texttt{while}\; b\; \texttt{do}\; S, \sigma) \Downarrow \sigma''$.

Other cases same technique

Outline

Theorem 10

Consider a program P with $\{\emptyset\}P\{X\}$. For any states $\sigma_1, \sigma_1', \sigma_2, \sigma_2'$, if:

$$\langle P, \sigma_1 \rangle \Downarrow \sigma_1' \qquad \langle P, \sigma_2 \rangle \Downarrow \sigma_2'$$

then

$$\sigma_2'|_Y = \sigma_2'|_Y$$

Theorem 10

Consider a program P with $\{\emptyset\}P\{X\}$. For any states $\sigma_1, \sigma_1', \sigma_2, \sigma_2'$, if:

$$\langle P, \sigma_1\rangle \Downarrow \sigma_1' \qquad \langle P, \sigma_2\rangle \Downarrow \sigma_2'$$

then

$$\sigma_2'|_Y = \sigma_2'|_Y$$

Proof.

By induction on P.

Theorem 10

Consider a program P with $\{\emptyset\}P\{X\}$*. For any states* $\sigma_1, \sigma_1', \sigma_2, \sigma_2'$*, if:*

$$\langle P, \sigma_1\rangle \Downarrow \sigma_1' \qquad \langle P, \sigma_2\rangle \Downarrow \sigma_2'$$

then

$$\sigma_2'|_Y = \sigma_2'|_Y$$

Proof.

By induction on P.

- When $P = (x := e)$. By definition of $\mathscr{S}$, we know that e must be constant and $Y = X \cup \{x\}$. Since e is a constant, $[\![e]\!](\sigma_1) = [\![e]\!](\sigma_2) = n$ and we have $\sigma_1'(x) = \sigma_2'(x) = n$ as desired.
- $P = P_1; P_2$. By assumption $\{\emptyset\}P_1\{X_0\}$ and $\{X_0\}P_2\{X\}$. How to apply induction hypothesis to P_2?

Theorem 10

Consider a program P with $\{\emptyset\}P\{X\}$. For any states $\sigma_1, \sigma_1', \sigma_2, \sigma_2'$, if:

$$\langle P, \sigma_1 \rangle \Downarrow \sigma_1' \qquad \langle P, \sigma_2 \rangle \Downarrow \sigma_2'$$

then

$$\sigma_2'|_Y = \sigma_2'|_Y$$

Proof.

By induction on P.

- When $P = (x := e)$. By definition of $\mathscr{S}$, we know that e must be constant and $Y = X \cup \{x\}$. Since e is a constant, $[\![e]\!](\sigma_1) = [\![e]\!](\sigma_2) = n$ and we have $\sigma_1'(x) = \sigma_2'(x) = n$ as desired.
- $P = P_1; P_2$. By assumption $\{\emptyset\}P_1\{X_0\}$ and $\{X_0\}P_2\{X\}$. How to apply induction hypothesis to P_2?

Solution: we need to **strengthen the induction hypothesis**. □

Proof of the analysis: (2)

We need a statement that matches the definition of $\mathscr{S}$.

Proof of the analysis: (2)

We need a statement that matches the definition of $\mathscr{S}$.

Theorem 11

Consider a program P with $\{X\}P\{Y\}$. *For any states* $\sigma_1, \sigma_1', \sigma_2, \sigma_2'$ *with* $\sigma_1|_X = \sigma_2|_X$, *if:*

$$\langle P, \sigma_1 \rangle \Downarrow \sigma_1' \qquad \langle P, \sigma_2 \rangle \Downarrow \sigma_2'$$

then

$$\sigma_2'|_Y = \sigma_2'|_Y$$

Outline

Extension of *While* (1) : termination operator

We extend the *While* language with the command `abort`.

Informal description: *the command halts execution of the program.*

One way of modeling `abort` : semantic rules stay unchanged and the configurations of form (`abort` , σ) are blocking.

- SOS: `abort` is different from `skip` and from `while true do skip`.
- NS: `abort` is different from `skip` but equivalent to `while true do skip`.

One solution that allows the NS to distinguish between termination by `abort` and non-termination: introduce special state σ_{abort} and

$$(\texttt{abort}, \sigma) \Downarrow \sigma_{abort}$$

But: must modify all other rules to take σ_{abort} into account!

Extensions to *While* (2) : non-deterministic choice

Extend *While* with the non-deterministic choice operator $c_1 \,\square\, c_2$.

Informal description: *choose non-deterministically to execute one of c_1 and c_2.*

The language now becomes non-deterministic.

- Formalisation as an SOS

$$\text{CH1}\ \frac{}{(c_1 \,\square\, c_2, \sigma) \to (c_1, \sigma)}$$

$$\text{CH2}\ \frac{}{(c_1 \,\square\, c_2, \sigma) \to (c_2, \sigma)}$$

- Formalisation as a NS

$$\text{CH1}\ \frac{(c_1, \sigma) \Downarrow \sigma'}{(c_1 \,\square\, c_2, \sigma) \Downarrow \sigma'}$$

$$\text{CH2}\ \frac{(c_2, \sigma) \Downarrow \sigma'}{(c_1 \,\square\, c_2, \sigma) \Downarrow \sigma'}$$

SOS can choose an expression that loops, while NS will always choose to eliminate non-termination.

Ex : `(x := 1)` □ `( while true do skip )`

Extension of *While* (3) : concurrency

Add a parallel composition to commands:

$$c ::= \ldots \mid (c_1 \parallel c_2).$$

How can we extend the SOS semantics? The natural semantics?