Deductive Verification of Programs

Laura Kovács

TU Wien

Outline

Computer Systems and Correctness

- Suppose we design a (complex) software system.

Computer Systems and Correctness

- Suppose we design a (complex) software system.
- We have requirements on how the system should function, for example safety, security, availability, etc.

Computer Systems and Correctness

- Suppose we design a (complex) software system.
- We have requirements on how the system should function, for example safety, security, availability, etc.
- How can one ensure that the system satisfies these requirements?

Computer Systems and Correctness

- Suppose we design a (complex) software system.
- We have requirements on how the system should function, for example safety, security, availability, etc.
- How can one ensure that the system satisfies these requirements?

  Deductive verification of programs

A Small Example

Consider the following program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);

  for (i = 0;i <= length;i++)
    array[i] = 0;
  return array;
}
```

A Small Example

Consider the following program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);

  for (i = 0;i <= length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

A Small Example

Consider the following program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);

  for (i = 0;i <= length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

Hardly: it writes into memory that has not been allocated.

A Small Example

Consider the following program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);

  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

A Small Example

Consider the following program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length); // may return 0!

  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

No: it may write to the null address.

A Small Example

Consider the following program:

```
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);
  if (!array) return 0;
  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?

A Small Example

Consider the following program:

```
/* Returns a new array of integers of a given
 length initialised by a non-zero value */
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);
  if (!array) return 0;
  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

Is this program correct?
No: it initialises the array by zeros

A Small Example

Consider the following program:

```
/* Returns a new array of integers of a given
 length initialised by a non-zero value */
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);
  if (!array) return 0;
  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

We discussed correctness of a program without defining what it means.

A Small Example

Consider the following program:

```
/* Returns a new array of integers of a given
 length initialised by a non-zero value */
int* allocateArray(int length)
{
  int i;
  int* array;
  array = malloc(sizeof(int)*length);
  if (!array) return 0;
  for (i = 0;i < length;i++)
    array[i] = 0;
  return array;
}
```

We discussed correctness of a program without defining what it means.
So what is correctness?

Notes

- We could spot the first two errors without knowing anything about the intended meaning of the program. But we had to understand the meaning of the program in general and some specific properties of programming in C-like languages (or other programming languages).

Notes

- We could spot the first two errors without knowing anything about the intended meaning of the program. But we had to understand the meaning of the program in general and some specific properties of programming in C-like languages (or other programming languages).
- To understand the last “error” we had to know something about the intended behaviour of the program.

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.

State Transition Systems and its Semantics

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.

- Find a formal language for expressing intended properties.

State Transition Systems and its Semantics

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.

- Find a formal language for expressing intended properties.

  This language must have a semantics that explains what are possible interpretations of the sentences of the formal language.

  The semantics is normally based on notions is true, is false, satisfies.

State Transition Systems and its Semantics

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.

  State Transition Systems and its Semantics

- Find a formal language for expressing intended properties.

  First-Order Logic (FOL), Temporal Logic

  This language must have a semantics that explains what are possible interpretations of the sentences of the formal language.

  The semantics is normally based on notions is true, is false, satisfies.

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.

State Transition Systems and its Semantics

- Find a formal language for expressing intended properties.

First-Order Logic (FOL), Temporal Logic

- Write a specification, that is, intended properties of the system in this language.

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.

State Transition Systems and its Semantics

- Find a formal language for expressing intended properties.

First-Order Logic (FOL), Temporal Logic

- Write a specification, that is, intended properties of the system in this language.

SAT/SMT/FOL Formulas, Temporal Logic Formulas

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.
- Find a formal language for expressing intended properties.
- Write a specification, that is, intended properties of the system in this language.
- Prove that the system model satisfies the specification.

State Transition Systems and its Semantics

First-Order Logic (FOL), Temporal Logic

SAT/SMT/FOL Formulas, Temporal Logic Formulas

How to establish correctness

- Consider the program/system as a mathematical object.
  Build a formal model of the system.

State Transition Systems and its Semantics

- Find a formal language for expressing intended properties.

First-Order Logic (FOL), Temporal Logic

- Write a specification, that is, intended properties of the system in this language.

SAT/SMT/FOL Formulas, Temporal Logic Formulas

- Prove that the system model satisfies the specification.

Deductive verification, model checking

Deductive Verification of Programs

- Consider the program/system as a mathematical object.
  Build a formal model of the system.
- Find a formal language for expressing intended properties.
- Write a specification, that is, intended properties of the system in this language.
- Prove that the system model satisfies the specification.

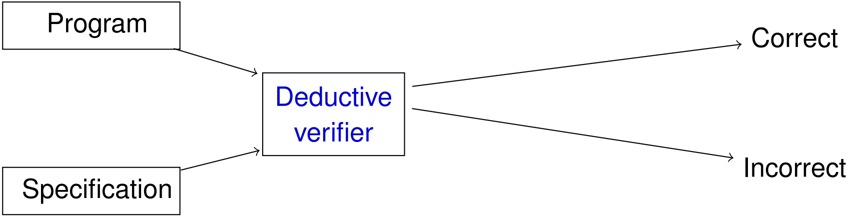
State Transition Systems and its Semantics

First-Order Logic (FOL),

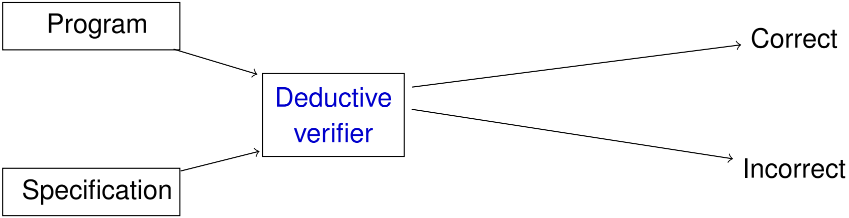
SAT/SMT/FOL Formulas,

Deductive verification,

Deductive Verification of Programs

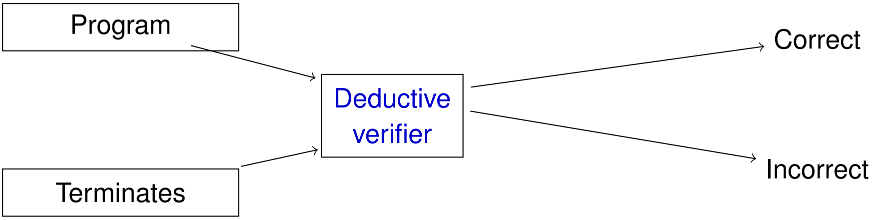


Deductive Verification of Programs

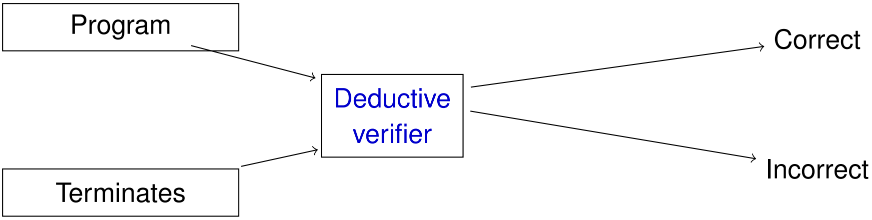


- Statically reason about program behavior without executing the program: consider all possible program executions under all inputs
- Examples of specifications: absence of arithmetic overflow, array is sorted, program terminates . . .

Deductive Verification of Programs

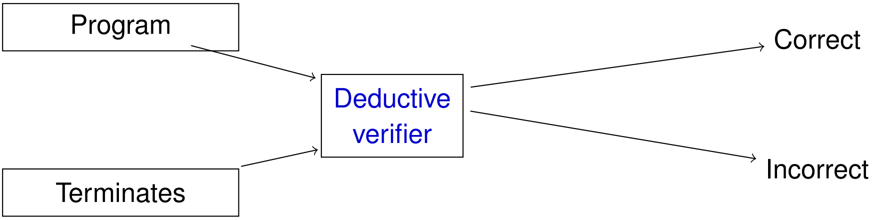


Deductive Verification of Programs



The Halting Problem

Deductive Verification of Programs

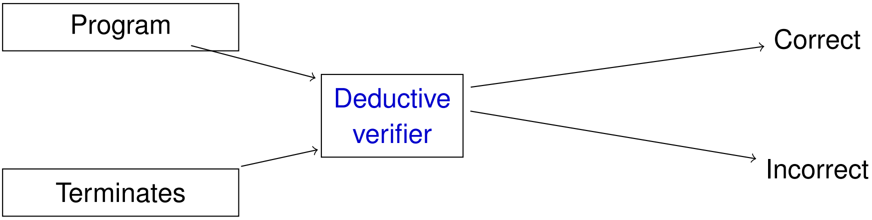


The Halting Problem: Undecidable

- A problem is undecidable if there does not exist a Turing machine that can solve it

Alan Turing, 1936

Deductive Verification of Programs

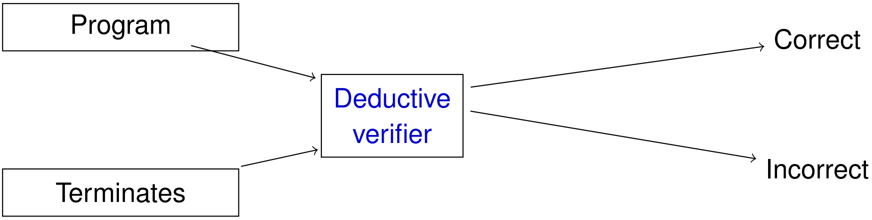


The Halting Problem: Undecidable

- A problem is undecidable if there does not exist a C/Java program that can solve it

Alan Turing, 1936

Deductive Verification of Programs

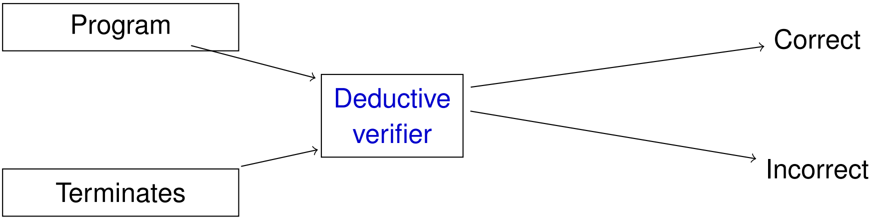


The Halting Problem: Undecidable

- A problem is undecidable if there does not exist a computer program that can solve it

Alan Turing, 1936

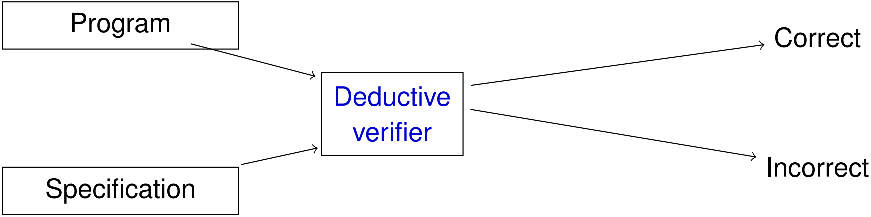
Deductive Verification of Programs



Undecidable: Verifier cannot exist

Alan Turing, 1936

Deductive Verification of Programs



Undecidable: Verifier cannot exist

- There is no computer program that can always decide whether a computer program satisfies a non-trivial specification (Rice Theorem)

Alan Turing, 1936

Deductive Verification – Living with Undecidability

- Programs that “sometimes” diverge

Deductive Verification – Living with Undecidability

- Programs that “sometimes” diverge

- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, ...

Deductive Verification – Living with Undecidability

- Programs that "sometimes" diverge
- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, . . .
- Partial/unsound verification approaches
  - **Testing:** analyze program executions on a fixed input set

Deductive Verification – Living with Undecidability

- Programs that “sometimes” diverge

- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, . . .

- Partial/unsound verification approaches
  - **Testing:** analyze program executions on a fixed input set

```
if (x-100<=0)
 if (y-100<=0)
   if (x+y-200==0)
     crash();
```

Deductive Verification – Living with Undecidability

- Programs that “sometimes” diverge

- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, ...

- Partial/unsound verification approaches
  - **Testing:** analyze program executions on a fixed input set

```
if (x-100<=0)
 if (y-100<=0)
   if (x+y-200==0)
      crash();
```

If `x` and `y` are 32-bit integers, what is the probability of a crash?

- Programs that “sometimes” diverge

- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, . . .

- Partial/unsound verification approaches
  - **Testing:** analyze program executions on a fixed input set

```
if (x-100<=0)
 if (y-100<=0)
   if (x+y-200==0)
     crash();
```

If `x` and `y` are 32-bit integers, what is the probability of a crash?

- $1/2^{64}$

Deductive Verification – Living with Undecidability

- Programs that “sometimes” diverge

- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, . . .

- Partial/unsound verification approaches
  - **Testing:** analyze program executions on a fixed input set

```
if (x-100<=0)
 if (y-100<=0)
   if (x+y-200==0)
     crash();
```

If x and y are 32-bit integers, what is the probability of a crash?

- $1/2^{64}$

Testing is hard!

Deductive Verification – Living with Undecidability

- Programs that “sometimes” diverge

- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, . . .

- Partial/unsound verification approaches
  - **Testing:** analyze program executions on a fixed input set

- Abstractions
  - **Model checking:** verify a superset of program executions

Deductive Verification – Living with Undecidability

- Programs that "sometimes" diverge

- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, . . .

- Partial/unsound verification approaches
  - **Testing:** analyze program executions on a fixed input set

- Abstractions
  - **Model checking:** verify a superset of program executions

- User assistance
  - **Deductive verification:** using program annotations (e.g. loop invariants)

Deductive Verification – Living with Undecidability

- Programs that “sometimes” diverge

- Limit programs that can be analyzed
  - finite-state, loop-free, bounded arithmetic, . . .

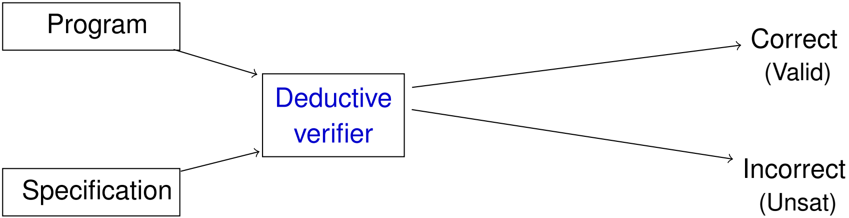
- Partial/unsound verification approaches
  - **Testing:** analyze program executions on a fixed input set

- Abstractions
  - **Model checking:** verify a superset of program executions

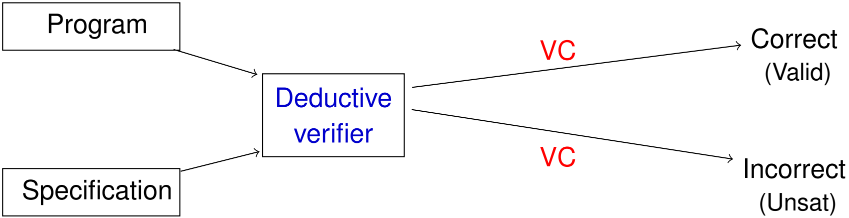
- User assistance
  - **Deductive verification:** using program annotations (e.g. loop invariants)

    Algorithmic approach to deductive verification: **(my research area)**
    - automate the generation of annotations,
    - fully automatic, push-button verification for classes of programs

Deductive Verification of Programs

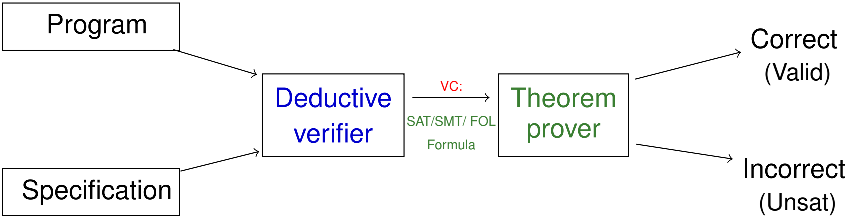


Deductive Verification of Programs in Practice



- Verification condition (VC): A logical formula ϕ s.t. the program is correct iff ϕ is valid

Deductive Verification of Programs in Practice



- Verification condition (VC): SAT/SMT/FOL formula ϕ s.t. the program is correct iff ϕ is valid
- Available tools: Dafny (Microsoft), Why3, KeY, ESC/Java, ...

Deductive Verification of Programs in Dafny

Revisiting our `allocateArray` example

Deductive Verification of Programs in Dafny

```
method allocateArray(a:array<int>, length:int)



{
  var i:int;
  i:=0;
  while (i<length)

  {
    a[i]:=1; i:=i+1;
  }
}
```

Deductive Verification of Programs in Dafny

```
method allocateArray(a:array<int>, length:int)
requires 0<=length<a.Length
modifies a

{
  var i:int;
  i:=0;
  while (i<length)

  {
    a[i]:=1; i:=i+1;
  }
}
```

Deductive Verification of Programs in Dafny

```
method allocateArray(a:array<int>, length:int)
requires 0<=length<a.Length
modifies a
ensures forall k::0<=k<length==>a[k]!=0
{
  var i:int;
  i:=0;
  while (i<length)

  {
    a[i]:=1; i:=i+1;
  }
}
```

Deductive Verification of Programs in Dafny

```
method allocateArray(a:array<int>, length:int)
requires 0<=length<a.Length
modifies a
ensures forall k::0<=k<length==>a[k]!=0
{
  var i:int;
  i:=0;
  while (i<length)
  invariant (forall k::0<=k<i&& k<length==>a[k]!=0);
  {
    a[i]:=1; i:=i+1;
  }
}
```

Deductive Verification of Programs

- Living with undecidability;
- Algorithmic approach, using Hoare logic;
- Automated approach, thanks to SAT/SMT solving and theorem proving.

Deductive Verification of Programs

Tentative list of topics to be addressed:

- Small imperative language IMP
  - Syntax and semantics
- Hoare logic: basis of all deductive verification techniques
- Deductive verification of IMP
  - Axiomatic Semantics
  - Predicate transformers - for algorithmic verification
- Inferring loop invariants automatically

Outline

Small Imperative Language IMP – Syntax

We consider a small imperative language IMP.

Syntactic Sets of IMP

•	Int	positive and negative (integer) numerals	$n, m, \ldots$ (e.g. $-5, 0, 10$)
•	Loc	locations	$x, y, z, \ldots$
•	AExp	arithmetic expressions	a
•	BExp	boolean expressions	b
•	P	programs	p

Small Imperative Language IMP – Syntax

Abstract Syntax of IMP: Arithmetic Expressions AExp

$$
\begin{array}{rcll}
a & ::= & n & \text{for } n \in \mathrm{Int} \\
& \mid & x & \text{for } x \in \mathrm{Loc} \\
& \mid & a_1 + a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
& \mid & a_1 - a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
& \mid & a_1 * a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
& \mid & a_1 / a_2 & \text{for } a_1, a_2 \in \mathrm{AExp}
\end{array}
$$

Small Imperative Language IMP – Syntax

Abstract Syntax of IMP: Arithmetic Expressions AExp

$$
\begin{array}{lcll}
a & ::= & n & \text{for } n \in \mathrm{Int} \\
 & | & x & \text{for } x \in \mathrm{Loc} \\
 & | & a_1 + a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
 & | & a_1 - a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
 & | & a_1 * a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
 & | & a_1 / a_2 & \text{for } a_1, a_2 \in \mathrm{AExp}
\end{array}
$$

- In the abstract synthax, we omit parantheses.

$$2 + 3 * 4 - 5$$

can be parsed as $2 + ((3 * 4) - 5$ or as $(2 + 3) * (4 - 5)$, depending on the *concrete syntax* (e.g. using operator precedence).

Small Imperative Language IMP – Syntax

Abstract Syntax of IMP: Arithmetic Expressions AExp

$$
\begin{array}{lcll}
a & ::= & n & \text{for } n \in \mathrm{Int} \\
 & | & x & \text{for } x \in \mathrm{Loc} \\
 & | & a_1 + a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
 & | & a_1 - a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
 & | & a_1 * a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \\
 & | & a_1 / a_2 & \text{for } a_1, a_2 \in \mathrm{AExp}
\end{array}
$$

- In the abstract synthax, we omit parantheses.

  $$2 + 3 * 4 - 5$$

  can be parsed as $2 + ((3 * 4) - 5$ or as $(2 + 3) * (4 - 5)$, depending on the *concrete syntax* (e.g. using operator precedence).
- $3 + 5$ is not syntactically identical to $5 + 3$.
  But, $(3 + 5)$ is syntactically identical to $3 + 5$.

Small Imperative Language IMP – Syntax

Abstract Syntax of IMP: Boolean Expressions BExp

$$
\begin{array}{rcll}
b & ::= & \mathrm{true} & \\
& | & \mathrm{false} & \\
& | & a_1 \ \mathcal{AOP} \ a_2 & \text{for } a_1, a_2 \in \mathrm{AExp} \text{ and } \mathcal{AOP} \in \{=, <, >, \leq, \geq\} \\
& | & \neg b & \text{for } b \in \mathrm{BExp} \\
& | & b_1 \ \mathcal{BOP} \ b_2 & \text{for } b_1, b_2 \in \mathrm{BExp} \text{ and } \mathcal{BOP} \in \{\wedge, \vee\}
\end{array}
$$

Small Imperative Language IMP – Syntax

Abstract Syntax of IMP: Programs P

$$
\begin{array}{rcll}
p & ::= & \text{skip} & \\
 & | & \text{abort} & \\
 & | & x := a & \text{for } x \in \text{Loc and } a \in \text{AExp} \\
 & | & p_1 ; p_2 & \text{for } p_1, p_2 \in \text{P} \\
 & | & \text{if } b \text{ then } p_1 \text{ else } p_2 & \text{for } b \in \text{BExp and } p_1, p_2 \in \text{P} \\
 & | & \text{while } b \text{ do } p \text{ od} & \text{for } b \in \text{BExp and } p \in \text{P}
\end{array}
$$

Small Imperative Language IMP – Syntax

Example of an IMP Program

p

$$\Longrightarrow^* \quad x := 10; y := 0; \text{ while } x > 0 \text{ do } y := y + x; x := x - 1 \text{ od}$$

Small Imperative Language IMP – Syntax

Example of an IMP Program

$$
\begin{aligned}
p \quad &\Longrightarrow \quad p_1 ; p_2 \\
&\Longrightarrow \quad p_1 ; p_3 ; p_4 \\
&\Longrightarrow^* \quad x := 10; y := 0;\ \text{while } x > 0 \text{ do } y := y + x; x := x - 1 \text{ od}
\end{aligned}
$$

Small Imperative Language IMP – Syntax

Example of an IMP Program

$$
\begin{aligned}
p \quad &\Longrightarrow \quad p_1\,;p_2 \\
&\Longrightarrow \quad p_1\,;p_3\,;p_4 \\
&\Longrightarrow^* \quad x := e_1\,;y := e_2;\ \text{while } b \text{ do } p_5 \text{ od} \\
\\
&\Longrightarrow^* \quad x := 10; y := 0;\ \text{while } x > 0 \text{ do } y := y + x; x := x - 1 \text{ od}
\end{aligned}
$$

Small Imperative Language IMP – Syntax

Example of an IMP Program

$$
\begin{aligned}
p &\Longrightarrow p_1\,;p_2\\
&\Longrightarrow p_1\,;p_3\,;p_4\\
&\Longrightarrow^* x := e_1\,; y := e_2;\ \text{while } b \text{ do } p_5 \text{ od}\\
&\Longrightarrow^* x := 10; y := 0;\ \text{while } a_1 > a_2 \text{ do } p_6; p_7 \text{ od}\\
&\Longrightarrow^* x := 10; y := 0;\ \text{while } x > 0 \text{ do } y := y + x; x := x - 1 \text{ od}
\end{aligned}
$$

Small Imperative Language IMP – Semantics

- Meaning of IMP programs and expressions depends on the values of variables in locations.

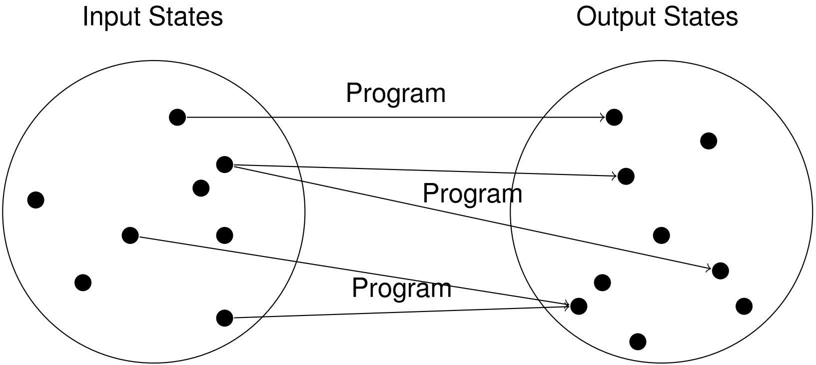
Small Imperative Language IMP – Semantics

- Meaning of IMP programs and expressions depends on the values of variables in locations.
- A **state** σ is a function from Loc to Int.
  - $\sigma(x)$ is the value/content of locations x in state σ

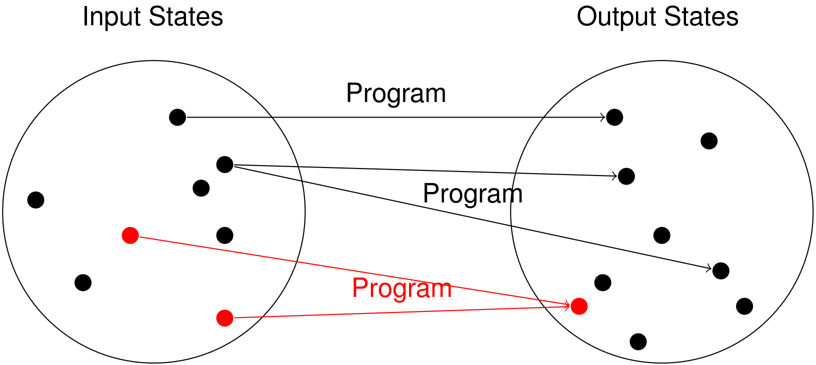
Small Imperative Language IMP – Semantics

- Meaning of IMP programs and expressions depends on the values of variables in locations.
- A state σ is a function from Loc to Int.
  - $\sigma(x)$ is the value/content of locations x in state σ
  - the set of states $\Sigma = \{\sigma | \sigma : \mathrm{Loc} \to \mathrm{Int}\}$

Programs as State Transformers

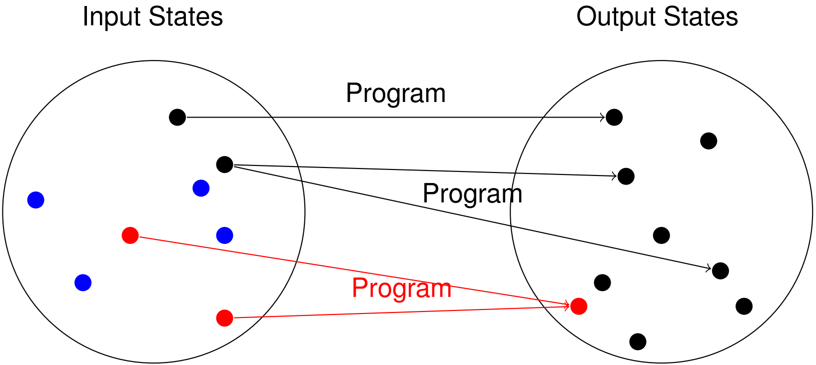


Programs as State Transformers



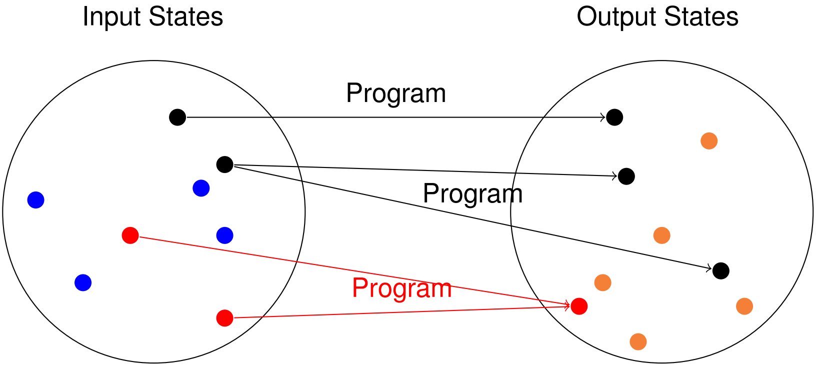
- Several inputs may be mapped to the same output.

Programs as State Transformers



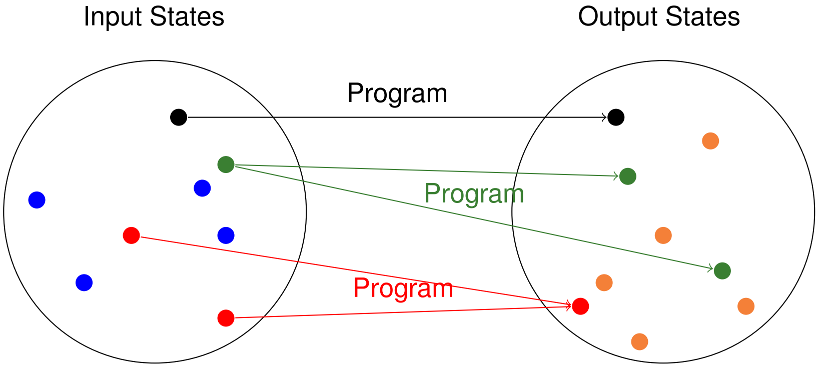
- Several inputs may be mapped to the same output.
- Some inputs are not mapped to any output.
  (Program aborts or loops.)

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.

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 (non-determinism).

Small Imperative Language IMP – Semantics

- Evaluation of IMP expressions:
  - Ternary relation on an $\mathrm{AExp}/\mathrm{BExp}$ expression, a state σ and a value v
  - Denoted as $\langle exp, \sigma \rangle \to v$, where $exp \in \mathrm{AExp}$ or $exp \in \mathrm{BExp}$:

    "Expression exp in state σ evaluates to value v".

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP expressions:
  - Ternary relation on an $\mathrm{AExp}/\mathrm{BExp}$ expression, a state σ and a value v
  - Denoted as $\langle exp, \sigma \rangle \to v$, where $exp \in \mathrm{AExp}$ or $exp \in \mathrm{BExp}$:

    "Expression exp in state σ evaluates to value v".

    Operational semantics: "mapping of expressions to values"
- Why no state on the right of $\to$?

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP expressions:
  - Ternary relation on an AExp/BExp expression, a state σ and a value v
  - Denoted as $\langle exp, \sigma \rangle \rightarrow v$, where $exp \in \mathrm{AExp}$ or $exp \in \mathrm{BExp}$:

    "Expression exp in state σ evaluates to value v".

    Operational semantics: "mapping of expressions to values"

- Why no state on the right of $\rightarrow$?

  Evaluation of IMP expressions has no side-effects (the state remains unchanged)

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP expressions:
  - Ternary relation on an AExp/BExp expression, a state σ and a value v
  - Denoted as $\langle exp, \sigma \rangle \to v$, where $exp \in \mathrm{AExp}$ or $exp \in \mathrm{BExp}$:

    "Expression exp in state σ evaluates to value v".

    Operational semantics: "mapping of expressions to values"
- Why no state on the right of $\to$?

  Evaluation of IMP expressions has no side-effects (the state remains unchanged)
- Can evaluations of expressions be considered as a function of 2 arguments exp and σ?

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP expressions:
  - Ternary relation on an AExp/BExp expression, a state σ and a value v
  - Denoted as $\langle exp, \sigma\rangle \rightarrow v$, where $exp \in \mathrm{AExp}$ or $exp \in \mathrm{BExp}$:

    "Expression exp in state σ evaluates to value v".

    Operational semantics: "mapping of expressions to values"
- Why no state on the right of $\rightarrow$?

  Evaluation of IMP expressions has no side-effects (the state remains unchanged)
- Can evaluations of expressions be considered as a function of 2 arguments exp and σ?

  Only if there is a unique value v for $\langle exp, \sigma\rangle \rightarrow v$

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP expressions:
  - Ternary relation on an $\mathrm{AExp}/\mathrm{BExp}$ expression, a state σ and a value v
  - Denoted as $\langle exp, \sigma\rangle \rightarrow v$, where $exp \in \mathrm{AExp}$ or $exp \in \mathrm{BExp}$:

    "Expression exp in state σ evaluates to value v".

    Operational semantics: "mapping of expressions to values"

- The pair $\langle exp, \sigma\rangle$ is:
  - an arithmetic expression configuration, for $exp \in \mathrm{AExp}$;
  - a boolean expression configuration, for $exp \in \mathrm{BExp}$;

- Evaluation of IMP programs:
  - Ternary relation on a program p, a state σ and a new state σ'
  - Denoted as $\langle p, \sigma \rangle \to \sigma'$:

    "Executing program p from state σ terminates in final state σ'".

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP programs:
  - Ternary relation on a program p, a state σ and a new state σ'
  - Denoted as $\langle p, \sigma \rangle \rightarrow \sigma'$:

    "Executing program p from state σ terminates in final state σ'".
- Execution of a program has effect
  - but no direct value
  - "result" of a program is a new state σ'

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP programs:
  - Ternary relation on a program p, a state σ and a new state σ'
  - Denoted as $\langle p, \sigma \rangle \rightarrow \sigma'$:

    "Executing program p from state σ terminates in final state σ'".
- Execution of a program has effect
  - but no direct value
  - "result" of a program is a new state σ'
- Evaluation of a program:
  - may terminate in a final state
  - may diverge and never yield a final state

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP programs:
  - Ternary relation on a program p, a state σ and a new state σ'
  - Denoted as $\langle p, \sigma \rangle \rightarrow \sigma'$:

    “Executing program p from state σ terminates in final state σ'”.
- Can evaluations of programs be considered as a function of 2 arguments p and σ?

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP programs:
  - Ternary relation on a program p, a state σ and a new state σ'
  - Denoted as $\langle p, \sigma \rangle \to \sigma'$:

    "Executing program p from state σ terminates in final state σ'".

- Can evaluations of programs be considered as a function of 2 arguments p and σ?

  Only if there is a unique successor state

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP programs:
  - Ternary relation on a program p, a state σ and a new state σ'
  - Denoted as $\langle p, \sigma \rangle \rightarrow \sigma'$:

    "Executing program p from state σ terminates in final state σ'".
- The pair $\langle p, \sigma \rangle$ is a program configuration from which it remains to execute p from state σ.
- A final state is also a program configuration, called final configuration.

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP programs:
  - Ternary relation on a program p, a state σ and a new state σ'
  - Denoted as $\langle p, \sigma \rangle \to \sigma'$:

    "Executing program p from state σ terminates in final state σ'".
- The pair $\langle p, \sigma \rangle$ is a program configuration from which it remains to execute p from state σ.
- A final state is also a program configuration, called final configuration.
- Set of program configurations $\mathcal{C} = (P \times \Sigma) \cup \Sigma$

Small Imperative Language IMP – Operational Semantics

- Evaluation of IMP programs:
  - Ternary relation on a program p, a state σ and a new state σ'
  - Denoted as $\langle p, \sigma \rangle \rightarrow \sigma'$:

    “Executing program p from state σ terminates in final state σ'”.
- The pair $\langle p, \sigma \rangle$ is a program configuration from which it remains to execute p from state σ.
- A final state is also a program configuration, called final configuration.
- Set of program configurations $\mathcal{C} = (P \times \Sigma) \cup \Sigma$
  - Transition relation $==>$ of IMP programs is a relation over $\mathcal{C} \times \mathcal{C}$
    - $\langle p, \sigma \rangle ==> \langle p', \sigma' \rangle$ intermediate step of program execution
    - $\langle p, \sigma \rangle ==> \sigma'$ last step of program execution

Evaluation Rule Notation:

$$\frac{F_1 \quad \ldots \quad F_k}{G} \ ,$$

to denote “if $F_1, \ldots, F_k$, then G”.

When $k = 0$, then G is an axiom:

$$\frac{}{G} \ .$$

Small Imperative Language IMP – Operational Semantics

Note: In $\langle a, \sigma \rangle \to v$, the value v is an integer value.

Small Imperative Language IMP – Operational Semantics

Note: In $\langle a, \sigma \rangle \to v$, the value v is an integer value.

Evaluation Rules for AExp

$$\frac{}{\langle n, \sigma \rangle \to \mathbf{n}} \; ,$$

where $\mathbf{n} \in \mathbf{Z}$ represents the integer number corresponding to $n \in \mathrm{Int}$

Small Imperative Language IMP – Operational Semantics

Note: In $\langle a, \sigma \rangle \to v$, the value v is an integer value.

Evaluation Rules for AExp

$$\frac{}{\langle n, \sigma \rangle \to \mathbf{n}} \;,$$

where $\mathbf{n} \in \mathbf{Z}$ represents the integer number corresponding to $n \in \mathrm{Int}$

$$\frac{}{\langle x, \sigma \rangle \to \sigma(x)}$$

Small Imperative Language IMP – Operational Semantics

Note: In $\langle a, \sigma \rangle \rightarrow v$, the value v is an integer value.

Evaluation Rules for AExp

$$\frac{}{\langle n, \sigma \rangle \rightarrow \mathbf{n}} \ ,$$

where $\mathbf{n} \in \mathbf{Z}$ represents the integer number corresponding to $n \in \mathrm{Int}$

$$\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow n_1 +_{\mathbf{Z}} n_2} \ ,$$

where $+_{\mathbf{Z}}$ is the integer sum operator

Note: In $\langle a, \sigma \rangle \rightarrow v$, the value v is an integer value.

Evaluation Rules for AExp

$$\frac{}{\langle n, \sigma \rangle \rightarrow \mathbf{n}} \;,$$

where $\mathbf{n} \in \mathbf{Z}$ represents the integer number corresponding to $n \in \mathrm{Int}$

$$\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow n_1 +_{\mathbf{Z}} n_2} \;,$$

where $+_{\mathbf{Z}}$ is the integer sum operator

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 - a_2, \sigma \rangle \rightarrow n_1 -_{\mathbf{Z}} n_2} \;,$$

where $-_{\mathbf{Z}}$ is the integer substraction operator

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 * a_2, \sigma \rangle \rightarrow n_1 *_{\mathbf{Z}} n_2} \;,$$

where $*_{\mathbf{Z}}$ is the integer product operator

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 / a_2, \sigma \rangle \rightarrow n_1 /_{\mathbf{Z}} n_2} \;,$$

where $/_{\mathbf{Z}}$ is the of integer division operator

An Example of Evaluating $a \in \mathrm{AExp}$

Let a be $(x + 5) + (8 + 10)$.
Consider the state σ_0 with $\sigma_0(x) = 0$.

$$\frac{}{\langle (x + 5) + (8 + 10), \sigma_0 \rangle \to}$$

An Example of Evaluating $a \in \mathrm{AExp}$

Let a be $(x + 5) + (8 + 10)$.
Consider the state σ_0 with $\sigma_0(x) = 0$.

$$\dfrac{\dfrac{}{\langle x + 5, \sigma_0 \rangle \to} \qquad \dfrac{}{\langle 8 + 10, \sigma_0 \rangle \to}}{\langle (x + 5) + (8 + 10), \sigma_0 \rangle \to} \; \mathit{sum}$$

An Example of Evaluating $a \in \mathrm{AExp}$

Let a be $(x+5)+(8+10)$.
Consider the state σ_0 with $\sigma_0(x)=0$.

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma_0\rangle \to} \qquad \dfrac{}{\langle 5, \sigma_0\rangle \to}}{\langle x+5, \sigma_0\rangle \to}\;\textit{sum} \qquad \dfrac{}{\langle 8+10, \sigma_0\rangle \to}}{\langle (x+5)+(8+10), \sigma_0\rangle \to}\;\textit{sum}$$

An Example of Evaluating $a \in \mathrm{AExp}$

Let a be $(x + 5) + (8 + 10)$.
Consider the state σ_0 with $\sigma_0(x) = 0$.

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma_0 \rangle \to 0}\ \textit{locations} \qquad \dfrac{}{\langle 5, \sigma_0 \rangle \to 5}\ \textit{numbers}}{\langle x + 5, \sigma_0 \rangle \to 5}\ \textit{sum} \qquad \dfrac{}{\langle 8 + 10, \sigma_0 \rangle \to}}{\langle (x + 5) + (8 + 10), \sigma_0 \rangle \to}\ \textit{sum}$$

Small Imperative Language IMP – Operational Semantics

An Example of Evaluating $a \in \mathrm{AExp}$

Let a be $(x+5)+(8+10)$.
Consider the state σ_0 with $\sigma_0(x)=0$.

$$\dfrac{\dfrac{\dfrac{}{\langle x,\sigma_0\rangle \to 0}\,\textit{locations} \qquad \dfrac{}{\langle 5,\sigma_0\rangle \to 5}\,\textit{numbers}}{\langle x+5,\sigma_0\rangle \to 5}\,\textit{sum} \qquad \dfrac{\dfrac{}{\langle 8,\sigma_0\rangle \to 8}\,\textit{numbers} \qquad \dfrac{}{\langle 10,\sigma_0\rangle \to 10}\,\textit{numbers}}{\langle 8+10,\sigma_0\rangle \to 18}\,\textit{sum}}{\langle (x+5)+(8+10),\sigma_0\rangle \to}\,\textit{sum}$$

An Example of Evaluating $a \in \mathrm{AExp}$

Let a be $(x+5)+(8+10)$.
Consider the state σ_0 with $\sigma_0(x) = 0$.

$$
\dfrac{
  \dfrac{
    \dfrac{}{\langle x, \sigma_0\rangle \to 0}\,\textit{locations}
    \quad
    \dfrac{}{\langle 5, \sigma_0\rangle \to 5}\,\textit{numbers}
  }{\langle x+5, \sigma_0\rangle \to 5}\,\textit{sum}
  \qquad
  \dfrac{
    \dfrac{}{\langle 8, \sigma_0\rangle \to 8}\,\textit{numbers}
    \quad
    \dfrac{}{\langle 10, \sigma_0\rangle \to 10}\,\textit{numbers}
  }{\langle 8+10, \sigma_0\rangle \to 18}\,\textit{sum}
}{\langle (x+5)+(8+10), \sigma_0\rangle \to 23}\,\textit{sum}
$$

An Example of Evaluating $a \in \mathrm{AExp}$

Let a be $(x + 5) + (8 + 10)$.
Consider the state σ_0 with $\sigma_0(x) = 0$.

$$\dfrac{\dfrac{\dfrac{}{\langle x, \sigma_0 \rangle \to 0}\ \textit{locations} \quad \dfrac{}{\langle 5, \sigma_0 \rangle \to 5}\ \textit{numbers}}{\langle x + 5, \sigma_0 \rangle \to 5}\ \textit{sum} \qquad \dfrac{\dfrac{}{\langle 8, \sigma_0 \rangle \to 8}\ \textit{numbers} \quad \dfrac{}{\langle 10, \sigma_0 \rangle \to 10}\ \textit{numbers}}{\langle 8 + 10, \sigma_0 \rangle \to 18}\ \textit{sum}}{\langle (x + 5) + (8 + 10), \sigma_0 \rangle \to 23}\ \textit{sum}$$

The above structure of evaluation rules illustrates the derivation tree, or simply derivation of $\langle a, \sigma_0 \rangle \to 23$.

Small Imperative Language IMP – Operational Semantics

Note: In $\langle b, \sigma \rangle \to v$, the value v is a boolean value/truth value.

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for BExp

$$\frac{}{\langle \textit{true}, \sigma \rangle \rightarrow \textbf{true}} \,,$$

where **true** represents the boolean value 1 (**true**)

$$\frac{}{\langle \textit{false}, \sigma \rangle \rightarrow \textbf{false}} \,,$$

where **false** represents the boolean value 0 (**false**)

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for BExp

$$\frac{}{\langle \mathit{true}, \sigma \rangle \to \mathbf{true}} \,,$$

where **true** represents the boolean value 1 (**true**)

$$\frac{}{\langle \mathit{false}, \sigma \rangle \to \mathbf{false}} \,,$$

where **false** represents the boolean value 0 (**false**)

$$\frac{\langle a_1, \sigma \rangle \to n_1 \quad \langle a_2, \sigma \rangle \to n_2}{\langle a_1 = a_2, \sigma \rangle \to \mathbf{true}} \,,$$

if n_1 and n_2 are equal

$$\frac{\langle a_1, \sigma \rangle \to n_1 \quad \langle a_2, \sigma \rangle \to n_2}{\langle a_1 = a_2, \sigma \rangle \to \mathbf{false}} \,,$$

if n_1 and n_2 are not equal

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for BExp

$$\frac{}{\langle \mathit{true}, \sigma\rangle \to \mathbf{true}} \;,$$

where **true** represents the boolean value 1 (**true**)

$$\frac{}{\langle \mathit{false}, \sigma\rangle \to \mathbf{false}} \;,$$

where **false** represents the boolean value 0 (**false**)

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 = a_2, \sigma\rangle \to \mathbf{true}} \;,$$

if n_1 and n_2 are equal

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 = a_2, \sigma\rangle \to \mathbf{false}} \;,$$

if n_1 and n_2 are not equal

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 \leq a_2, \sigma\rangle \to \mathbf{true}} \;,$$

if n_1 is less than or equal to n_2

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 \leq a_2, \sigma\rangle \to \mathbf{false}} \;,$$

if n_1 is not less than or equal to n_2

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for BExp (contd)

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 \geq a_2, \sigma\rangle \to \mathbf{true}}\ ,$$

if n_1 is greater than or equal to n_2

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 \geq a_2, \sigma\rangle \to \mathbf{false}}\ ,$$

if n_1 is not greater than or equal to n_2

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 < a_2, \sigma\rangle \to \mathbf{true}}\ ,$$

if n_1 is less than n_2

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 < a_2, \sigma\rangle \to \mathbf{false}}\ ,$$

if n_1 is not less than n_2

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 > a_2, \sigma\rangle \to \mathbf{true}}\ ,$$

if n_1 is greater than n_2

$$\frac{\langle a_1, \sigma\rangle \to n_1 \quad \langle a_2, \sigma\rangle \to n_2}{\langle a_1 > a_2, \sigma\rangle \to \mathbf{false}}\ ,$$

if n_1 is not greater than n_2

Evaluation Rules for BExp (contd)

$$\frac{\langle b, \sigma \rangle \to \textbf{true}}{\langle \neg b, \sigma \rangle \to \textbf{false}}\ \textit{neg} \qquad \frac{\langle b, \sigma \rangle \to \textbf{false}}{\langle \neg b, \sigma \rangle \to \textbf{true}}\ \textit{neg}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for BExp (contd)

$$\frac{\langle b, \sigma \rangle \to \textbf{true}}{\langle \neg b, \sigma \rangle \to \textbf{false}}\ \textit{neg} \qquad \frac{\langle b, \sigma \rangle \to \textbf{false}}{\langle \neg b, \sigma \rangle \to \textbf{true}}\ \textit{neg}$$

$$\frac{\langle b_1, \sigma \rangle \to t_1 \quad \langle b_2, \sigma \rangle \to t_2}{\langle b_1 \wedge b_2, \sigma \rangle \to t}\ ,$$

where t is **true** if both t_1 and t_2 are **true**, and is **false** otherwise.

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for BExp (contd)

$$\frac{\langle b, \sigma \rangle \to \textbf{true}}{\langle \neg b, \sigma \rangle \to \textbf{false}} \; neg \qquad \frac{\langle b, \sigma \rangle \to \textbf{false}}{\langle \neg b, \sigma \rangle \to \textbf{true}} \; neg$$

$$\frac{\langle b_1, \sigma \rangle \to t_1 \quad \langle b_2, \sigma \rangle \to t_2}{\langle b_1 \wedge b_2, \sigma \rangle \to t} \; ,$$

where t is **true** if both t_1 and t_2 are **true**, and is **false** otherwise.

$$\frac{\langle b_1, \sigma \rangle \to t_1 \quad \langle b_2, \sigma \rangle \to t_2}{\langle b_1 \vee b_2, \sigma \rangle \to t} \; ,$$

where t is **false** if both t_1 and t_2 are **false**, and is **true** otherwise.

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \to} \qquad \frac{}{\langle \textbf{abort}, \sigma \rangle \to}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma} \qquad \frac{}{\langle \textbf{abort}, \sigma\rangle \not\to \qquad \text{undefined}}$$

Evaluation of IMP programs is a partial relation!

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma}$$

$$\frac{}{\langle \mathbf{abort}, \sigma \rangle \not\to} \quad \text{undefined}$$

$$\frac{}{\langle x := a, \sigma \rangle \to}$$

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma} \qquad \frac{}{\langle \textbf{abort}, \sigma\rangle \not\to} \ \text{undefined}$$

$$\frac{\langle a, \sigma\rangle \to v}{\langle x := a, \sigma\rangle \to \sigma[x/v]}$$

Notation: We write $\sigma[x/v]$ to denote the state obtained from σ by replacing x by v. That is:

$$\sigma[x/v](y) = \begin{cases} v, & \text{if } y = x, \\ \sigma(y), & \text{otherwise} \end{cases}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \to \sigma}$$

$$\frac{}{\langle \textbf{abort}, \sigma \rangle \not\to} \quad \text{undefined}$$

$$\frac{\langle a, \sigma \rangle \to v}{\langle x := a, \sigma \rangle \to \sigma[x/v]}$$

$$\frac{}{\langle p_1 ; p_2, \sigma \rangle \to}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \rightarrow \sigma} \qquad \frac{}{\langle \textbf{abort}, \sigma\rangle \not\rightarrow \qquad \text{undefined}}$$

$$\frac{\langle a, \sigma\rangle \rightarrow v}{\langle x := a, \sigma\rangle \rightarrow \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma\rangle \rightarrow \sigma' \quad \langle p_2, \sigma'\rangle \rightarrow \sigma''}{\langle p_1 ; p_2, \sigma\rangle \rightarrow \sigma''}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \to \sigma} \qquad \frac{}{\langle \textbf{abort}, \sigma \rangle \not\to \quad \text{undefined}}$$

$$\frac{\langle a, \sigma \rangle \to v}{\langle x := a, \sigma \rangle \to \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma \rangle \to \sigma' \quad \langle p_2, \sigma' \rangle \to \sigma''}{\langle p_1 ; p_2, \sigma \rangle \to}$$

$$\frac{}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma \rangle \to}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}$$

$$\frac{}{\langle \textbf{abort}, \sigma\rangle \not\to} \quad \text{undefined}$$

$$\frac{\langle a, \sigma\rangle \to v}{\langle x := a, \sigma\rangle \to \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma\rangle \to \sigma' \quad \langle p_2, \sigma'\rangle \to \sigma''}{\langle p_1 ; p_2, \sigma\rangle \to}$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle p_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma\rangle \to \sigma'}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma} \qquad \frac{}{\langle \textbf{abort}, \sigma\rangle \not\to} \text{ undefined}$$

$$\frac{\langle a, \sigma\rangle \to v}{\langle x := a, \sigma\rangle \to \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma\rangle \to \sigma' \quad \langle p_2, \sigma'\rangle \to \sigma''}{\langle p_1 ; p_2, \sigma\rangle \to}$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle p_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma\rangle \to \sigma'} \qquad \frac{\langle b, \sigma\rangle \to \textbf{false} \quad \langle p_2, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma\rangle \to \sigma'}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma}$$

$$\frac{}{\langle \textbf{abort}, \sigma\rangle \not\to \quad \text{undefined}}$$

$$\frac{\langle a, \sigma\rangle \to v}{\langle x := a, \sigma\rangle \to \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma\rangle \to \sigma' \quad \langle p_2, \sigma'\rangle \to \sigma''}{\langle p_1 ; p_2, \sigma\rangle \to}$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle p_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma\rangle \to \sigma'}$$

$$\frac{\langle b, \sigma\rangle \to \textbf{false} \quad \langle p_2, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma\rangle \to \sigma'}$$

$$\frac{}{\langle \textbf{while } b \textbf{ do } p \textbf{ od }, \sigma\rangle \to}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma\rangle \to \sigma} \qquad \frac{}{\langle \textbf{abort}, \sigma\rangle \not\to \quad \text{undefined}}$$

$$\frac{\langle a, \sigma\rangle \to v}{\langle x := a, \sigma\rangle \to \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma\rangle \to \sigma' \quad \langle p_2, \sigma'\rangle \to \sigma''}{\langle p_1 ; p_2, \sigma\rangle \to}$$

$$\frac{\langle b, \sigma\rangle \to \textbf{true} \quad \langle p_1, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma\rangle \to \sigma'} \qquad \frac{\langle b, \sigma\rangle \to \textbf{false} \quad \langle p_2, \sigma\rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma\rangle \to \sigma'}$$

$$\frac{}{\langle \textbf{while } b \textbf{ do } p \textbf{ od }, \sigma\rangle \to \sigma}$$

Evaluation Rules for P

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \to \sigma}$$

$$\frac{}{\langle \textbf{abort}, \sigma \rangle \not\to \quad \text{undefined}}$$

$$\frac{\langle a, \sigma \rangle \to v}{\langle x := a, \sigma \rangle \to \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma \rangle \to \sigma' \quad \langle p_2, \sigma' \rangle \to \sigma''}{\langle p_1 ; p_2, \sigma \rangle \to}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{true} \quad \langle p_1, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{false} \quad \langle p_2, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } p_1 \textbf{ else } p_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \textbf{false}}{\langle \textbf{while } b \textbf{ do } p \textbf{ od }, \sigma \rangle \to \sigma}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma} \qquad \frac{}{\langle \mathbf{abort}, \sigma \rangle \not\to \qquad \text{undefined}}$$

$$\frac{\langle a, \sigma \rangle \to v}{\langle x := a, \sigma \rangle \to \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma \rangle \to \sigma' \quad \langle p_2, \sigma' \rangle \to \sigma''}{\langle p_1 ; p_2, \sigma \rangle \to}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle p_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2, \sigma \rangle \to \sigma'} \qquad \frac{\langle b, \sigma \rangle \to \mathbf{false} \quad \langle p_2, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ p\ \mathbf{od}\ , \sigma \rangle \to \sigma} \qquad \frac{}{\langle \mathbf{while}\ b\ \mathbf{do}\ p\ \mathbf{od}\ , \sigma \rangle \to \sigma''}$$

Small Imperative Language IMP – Operational Semantics

Evaluation Rules for P

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma}$$

$$\frac{}{\langle \mathbf{abort}, \sigma \rangle \not\to \qquad \text{undefined}}$$

$$\frac{\langle a, \sigma \rangle \to v}{\langle x := a, \sigma \rangle \to \sigma[x/v]}$$

$$\frac{\langle p_1, \sigma \rangle \to \sigma' \quad \langle p_2, \sigma' \rangle \to \sigma''}{\langle p_1 ; p_2, \sigma \rangle \to}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle p_1, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false} \quad \langle p_2, \sigma \rangle \to \sigma'}{\langle \mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ p\ \mathbf{od}\ , \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to \mathbf{true} \quad \langle p, \sigma \rangle \to \sigma' \quad \langle \mathbf{while}\ b\ \mathbf{do}\ p\ \mathbf{od}\ , \sigma' \rangle \to \sigma''}{\langle \mathbf{while}\ b\ \mathbf{do}\ p\ \mathbf{od}\ , \sigma \rangle \to \sigma''}$$

Small Imperative Language IMP – Operational Semantics

Summary:

The operational semantics of IMP is given by:

- Evaluation rules for AExp expressions,
- Evaluation rules for BExp expressions,
- Evaluation rules for P programs expressions,

by using (partial) evaluation relations.

Learning Objectives

- Understanding the task of deductive program verification
- Limitations and trends in deductive program verification
- Familiarize with the Dafny program verifier
- Syntax of the small imperative language IMP
- Operational semantics of IMP