Introduction to the formal semantics of programs
Slides Block 1

ADA 2024/25

Departamento de Matemática Universidade de Aveiro
Alexandre Madeira
(madeira@ua.pt)

September 30, 2024

This UC rigorously approach ADA

In order to deal with "algorithms development" we need to have rigorous notions of:

- what is a **programming language**
- what is a **program**
- how **interpret programs**

This UC rigorously approach ADA

In order to deal with "algorithms development" we need to have rigorous notions of:

- what is a **programming language**
- what is a **program**
- how **interpret programs**

Formal Semantics of programs

This UC rigorously approach ADA

In order to deal with "algorithms development" we need to have rigorous notions of:

- what is a **programming language**
- what is a **program**
- how **interpret programs**

Formal Semantics of programs

To make this "analysis", we mathematically formalise:

- the notions of **property** and **behaviour**
- the notions of **specification** and **algorithm correctness**
- the notion of **correctness proof**

This UC rigorously approach ADA

In order to deal with "algorithms development" we need to have rigorous notions of:

- what is a **programming language**
- what is a **program**
- how **interpret programs**

Formal Semantics of programs

To make this "analysis", we mathematically formalise:

- the notions of **property** and **behaviour**
- the notions of **specification** and **algorithm correctness**
- the notion of **correctness proof**

Formal Verification of programs

Outline

1. A formal semantics of programming languages, why?

2. Revisions: the structural induction principle

Formally treatment of programs, why?

In order to have a scientific discipline of programming:

- Programs shall be treated as **mathematical objects**
- The interpretation of each command shall be mathematically defined, **free of ambiguities**
- The behaviour of a program shall be **predictable and calculable** in a de **unambiguous and systematic** way

Formally treatment of programs, why?

In order to have a scientific discipline of programming:

- Programs shall be treated as **mathematical objects**
- The interpretation of each command shall be mathematically defined, **free of ambiguities**
- The behaviour of a program shall be **predictable and calculable** in a de **unambiguous and systematic** way

However:

- Usually the **behaviour of the PL commands is just informally documented**
- The interpretation of a programm is just provided by the **machine code built by the compiler**

Formally treatment of programs, why?

In order to have a scientific discipline of programming:

- Programs shall be treated as **mathematical objects**
- The interpretation of each command shall be mathematically defined, **free of ambiguities**
- The behaviour of a program shall be **predictable and calculable** in a de **unambiguous and systematic** way

However:

- Usually the **behaviour of the PL commands is just informally documented**
- The interpretation of a programm is just provided by the **machine code built by the compiler**
- There are usually several compilers for the same language; **they are not necessarily consistent with each other**
- Most of these compilers **have bugs**...

Why formally treat programs

The formal semantics (FS) of a program in a given PL

is the concrete meaning (mathematical structure) of a program. E.g.

- an input/output map of variables,
- a state transition system,
- ...

A FS is important for the development of algorithms, since:

- it allows the exact understanding of a program;
- it supports the definition of formalisms capable of:
  - verify properties about programs
  - prove the equivalence of programs (re-use/optimizations/...)
  - ...

Programming Semantics styles

Operational - Expresses the computation as an abstract transition system

$$(seq)\frac{\langle c_1, \sigma\rangle \to \sigma' \qquad \langle c_2, \sigma'\rangle \to \sigma''}{\langle c_1; c_2, \sigma\rangle \to \sigma''}$$

Programming Semantics styles

Operational - Expresses the computation as an abstract transition system

$$(seq)\frac{\langle c_1, \sigma\rangle \to \sigma' \qquad \langle c_2, \sigma'\rangle \to \sigma''}{\langle c_1; c_2, \sigma\rangle \to \sigma''}$$

Denotational - Mathematical definition of the input/output relation of a program, by induction on the syntactic structure of a program

$$\mathfrak{C}[\![.]\!] : \texttt{Cmd} \to (\Sigma \dashrightarrow \Sigma)$$

$$\mathfrak{C}[\![c_1; c_2]\!] := \mathfrak{C}[\![c_2]\!] \circ \mathfrak{C}[\![c_1]\!]$$

Programming Semantics styles

Operational - Expresses the computation as an abstract transition system

$$(seq)\frac{\langle c_1, \sigma\rangle \rightarrow \sigma' \qquad \langle c_2, \sigma'\rangle \rightarrow \sigma''}{\langle c_1; c_2, \sigma\rangle \rightarrow \sigma''}$$

Denotational - Mathematical definition of the input/output relation of a program, by induction on the syntactic structure of a program

$$\mathfrak{C}[\![.]\!] : \mathtt{Cmd} \rightarrow (\Sigma \dashrightarrow \Sigma)$$

$$\mathfrak{C}[\![c_1; c_2]\!] := \mathfrak{C}[\![c_2]\!] \circ \mathfrak{C}[\![c_1]\!]$$

Axiomatic - Formalisation of the programs properties by logic formulas

$$(seq)\frac{\{A\}c_1\{C\} \qquad \{C\}c_2\{B\}}{\{A\}c_1; c_2\{B\}}$$

On this course we focus on:

<u>Operational</u> - Expresses the computation as an abstract transition system

$$(seq)\frac{\langle c_1, \sigma\rangle \to \sigma' \qquad \langle c_2, \sigma'\rangle \to \sigma''}{\langle c_1; c_2, \sigma\rangle \to \sigma''}$$

Denotational - Mathematical definition of the input/output relation of a program, by induction on the syntactic structure of a program

$$\mathfrak{C}[\![.]\!] : \texttt{Cmd} \to (\Sigma \dashrightarrow \Sigma)$$

$$\mathfrak{C}[\![c_1; c_2]\!] := \mathfrak{C}[\![c_2]\!] \circ \mathfrak{C}[\![c_1]\!]$$

<u>Axiomatic</u> - Formalisation of the programs properties by logic formulas

$$(seq)\frac{\{A\}c_1\{C\} \qquad \{C\}c_2\{B\}}{\{A\}c_1; c_2\{B\}}$$

Outline

(Revision) Induction principle

Inductive set

is a set which elements are either:

- atomic
- obtained from atomic elements trough a finite number of applications of a given set of operations

(Revision) Induction principle

Inductive set

is a set which elements are either:

- atomic
- obtained from atomic elements trough a finite number of applications of a given set of operations

Example

The set $\mathbb{N}$ is inductive to the atom 0 and for the operation *suc*, since it is the smallest set that

- contains 0
- contains $suc(n)$ if $n \in \mathbb{N}$

(Revision) Induction principle

Inductive set

is a set which elements are either:

- atomic
- obtained from atomic elements trough a finite number of applications of a given set of operations

Example

The set $\mathbb{N}$ is inductive to the atom 0 and for the operation *suc*, since it is the smallest set that

- contains 0
- contains $suc(n)$ if $n \in \mathbb{N}$

Using the BNF notation

$$\mathbb{N} \ni n ::= 0 \mid suc(n)$$

(Revision) Induction principle

Inductive set

is a set which elements are either,

- atomics
- obtained from the atomic elements with a finite number of applications of a given set of operations (constructors)

Example

The set of binary trees over a set L, $Btree(L)$ is inductive to the atoms L, and for the operation *fork*

(Revision) Induction principle

Inductive set

is a set which elements are either,

- atomics
- obtained from the atomic elements with a finite number of applications of a given set of operations (constructors)

Example

The set of binary trees over a set L, $Btree(L)$ is inductive to the atoms L, and for the operation *fork*, since

- the leafs $l \in L$ are trees
- given two trees $t, t' \in Btree(L)$, $fork(t, t') \in Btree(L)$

(Revision) Induction principle

Inductive set

is a set which elements are either,

- atomics
- obtained from the atomic elements with a finite number of applications of a given set of operations (constructors)

Example

The set of binary trees over a set L, $Btree(L)$ is inductive to the atoms L, and for the operation *fork*, since

- the leafs $l \in L$ are trees
- given two trees $t, t' \in Btree(L)$, $fork(t, t') \in Btree(L)$

using the BNF notation

$$Btree(L) \ni t := l \mid fork(t, t), \text{ for } l \in L$$

Recalling the Induction Principle

Exercise 1

1. *Inductively define the set of the lists over a set A*
2. *Inductively define the set of arithmetic expressions over $\mathbb{Z}$ generated by the operations $+, -, \times$.*

(Revision) Induction principle

Mathematical induction principle (over $\mathbb{N}$)

Let P be a property on the naturals. If

- $P(0)$ is true, and
- the truth of $P(n)$ implies the truth of $P(n+1)$

$P(n)$ is true for any $n \in \mathbb{N}$

(Revision) Induction principle

Mathematical induction principle (over $\mathbb{N}$)

Let P be a property on the naturals. If

- $P(0)$ is true, and
- the truth of $P(n)$ implies the truth of $P(n+1)$

$P(n)$ is true for any $n \in \mathbb{N}$

Exercise 2

Prove by mathematical induction that, for any $n \in \mathbb{N}$,

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Structural induction principle

Structural induction principle

Let P be a property about an inductive set I. If

- $P(a)$ is true for any atom a of I
- For any constructor f of arity k, when $P(a_1) \ldots P(a_k)$ are true, $P(f(a_1, \ldots, a_k))$ is true.

Then, $P(i)$ is true for any $i \in I$.

Exercise

Exercise 3

Inductively define the set $\mathtt{NTerm}$ *of arithmetic expressions over* $\mathbb{Z}$ *with the connectives* $+$ *and* $\times$.

1. *Define a recursive function* $nmr : \mathtt{NTerm} \to \mathbb{N}$ *to calculate the number of occurrences of numbers in an expression (e.g.* $nmr(3 + 5 \times 2) = 3$*)*
2. *Define a recursive function* $cnt : \mathtt{NTerm} \to \mathbb{N}$ *to calculate the number of occurrences of connectives in an expression (e.g.* $cnt(3 + 5 \times 2) = 2$*)*
3. *Define a recursive function* $ent : \mathtt{NTerm} \to \mathbb{N}$ *to calculate the number of occurrences of entities in an expression (e.g.* $ent(3 + 5 \times 2) = 5$*)*
4. *Prove that, for any* $e \in \mathtt{NTerm}$,

$$ent(e) = nmr(e) + cnt(e)$$

Exercise

Exercise 4

Inductively define the set $\mathtt{AExp}$ *of arithmetic expressions over* $\mathtt{Var}$ *and* $\mathbb{Z}$ *with the connectives* $+$, $-$ *and* $*$.

1. *Define a recursive function* $fv : \mathtt{AExp} \to \mathcal{P}(\mathtt{Var})$ *to collect the set of variables of an expression (e.g.* $fv(3 * x + 5 * y) = \{x, y\}$*)*
2. *Define a recursive function* $occ : \mathtt{AExp} \times \mathtt{Var} \to \mathbb{N}$ *to calculate the number of occurrences of a given variable in an expression (e.g.* $occ(3 * x + 5 * y - x, x) = 2$*)*
3. *Define a recursive function for the "substitution" operator* $a[x := a']$*, that replaces the occurrences of* x *in* a *by the expression* a' *(e.g.* $(3 * x + 5 * y - x)[y := x + 1] = 3 * x + 5 * (x + 1) - x$*)*
4. *Prove that*
$$fv(a[x := a']) \subseteq (fv(a) \setminus \{x\}) \cup fv(a')$$
5. *Define a recursive function* $ent : \mathtt{AExp} \to \mathbb{N}$ *to calculate the number of entities in an expression (e.g.* $ent(3 + y * z) = 5$*)*
6. *Determine an expression to* $ent(a[x := a'])$ *using* $occ(a, x)$ *and* $ent(a)$ *and* $ent(a')$*. Prove its correctness.*

Exercise

Exercise 5

For the binary trees over L,

$$Btree(L) \ni t := l \mid fork(t, t), \ \textbf{with} \, l \in L$$

1. *Define a function* $nodes : Btree(L) \to \mathbb{N}$ *to calculate the number of nodes in a tree.*
2. *Define a function* $leafs : Btree(L) \to \mathbb{N}$ *to calculate the number of leafs in a tree*
3. *Prove that, for any* $t \in Btree(L)$,

$$leafs(t) = nodes(t) + 1$$

Exercise

Exercise 6

Consider the set of lists $List(A) \ni l := \epsilon \mid a.l, a \in A$

1. *Define functions* $odd : List(A) \to List(A)$ *and* $even : List(A) \to List(A)$ *that filter a list with the elements of odd and even positions, respectively.*
   Eg. $odd(a_1.a_2.a_3.a_4) = a_1.a_3$ *and* $even(a_1.a_2.a_3.a_4) = a_2.a_4$.
2. *Define a function* $lg : List(A) \to \mathbb{N}$ *that returns the length of a list (i.e. the number of its elements)*
3. *Define a function* $zip : List(A) \times List(A) \to List(A)$ *that zips the elements of two non empty lists.*
   E.g. $zip(a_1.a_2, b_1.b_2.b_3) = a_1.b_1.a_2.b_2.b_3$
4. *Prove that:*
   - *for any* $l_1, l_2 \in List(A)$, $lg(zip(l_1, l_2)) = lg(l_1) + lg(l_2)$
   - *for any* $l \in List(A)$, $l = zip(odd(l), even(l))$