Introduction

- We introduce the mathematical notion of inductive definition (generalizing recursive definitions)
- This is used to define the semantics of a simple programming language

Inductive definitions

Example Factorial

- \( f(0) = 1 \)
- \( f(1) = 1 \times f(0) \)
- \( f(2) = 2 \times f(1) \)
- \( f(3) = 3 \times f(2) \)
- ... 
- \( f(n) = n \times f(n-1) \)
- ... 
  \( n \in \mathbb{N} \)
Inductive definition

**Definition 9:** Inductive Definition Let \( U \) be set and \( \langle S, \preceq \rangle \) be a set \( S \) equipped with a well-founded relation \( \preceq \). An inductive definition of \( D \in S \mapsto \varphi(U) \) by induction on \( \preceq \) has the form

- \( D(s) := D_s \) where \( D_s \in \varphi(U) \) for all minimal elements \( s \) of \( S \) (i.e. \( \forall s' \in S : s' \not\preceq s \)).
- \( D(s) := F_s((D(s') | s' < s)) \) where \( F_s \in \{ \{s' \in S | s' < s \} \mapsto \varphi(U) \} \) \( \varphi(U) \) otherwise.

Example I:

- \( U = \mathbb{N}, \langle S, \preceq \rangle := \langle \mathbb{N}, \leq \rangle \)
- \( D(0) := \emptyset \) (empty set)
- \( D(n) := \{0\} \cup \{k+2 | \exists m < n: k \in D(m)\} \)

so \( D(n) = \{2k | 0 < k < n\} \)

We use the symbol ":=\) to mean "is defined as".

---

Well-definedness of inductive definitions

**Theorem 10:** An inductive definition \( D \) by induction on \( \preceq \) is well-defined (that is \( D \) exists and belongs to \( S \mapsto \varphi(U) \)).

**Proof:** By induction on \( \langle S, \preceq \rangle \). We remark that the first case is a special case of the second where the family \( \langle D(s') | s' < s \rangle \) is empty so \( F_s(\emptyset) = D_s \). So we only have to consider the second case. Assuming by induction hypothesis that \( D(s') \) exists for all \( s' < s \) the family \( \langle D(s') | s' < s \rangle \) is well-defined that is each \( D(s') \) is well-defined for all \( s' < s \). So \( D(s) = F_s((D(s') | s' < s)) \) does exist since the function \( F_s \in \{ \{s' \in S | s' < s \} \mapsto \varphi(U) \} \) \( \varphi(U) \) is well-defined and \( \langle D(s') | s' < s \rangle \) is \( \{s' \in S | s' < s \} \mapsto \varphi(U) \) so that \( D(s) \in \varphi(U) \). By induction

\( \forall s \in S : D(s) \in \varphi(U) \).
Syntax of a simple imperative programming language

Variables

\[ V \in \mathcal{V}, \quad \text{variables} \]
\[ \mathcal{V} := \{x, y, \ldots\} \]

Expressions

\[ E \in \mathcal{E}, \quad \text{expressions} \]
\[ E ::= 1 \]
\[ \mid \ V \]
\[ \mid \ ? \]
\[ \mid \ E - E \]
\[ \mid \ (E) \]

This grammar of expressions is ambiguous which means that some sentences can have different syntax trees hence different meanings. For example \(1 - 1 - 1\) could be interpreted has left-associative meaning \((1 - 1) - 1 = 0 - 1 = -1\) or right-associative meaning \(1 - (1 - 1) = 1 - 0 = 1\). To disambiguate the grammatical description of expressions, we have to assume left-associativity so that expressions should be evaluated from left to right that is \((1 - 1) - 1 = -1\).

Conditions

\[ B \in \mathcal{B}, \quad \text{conditions} \]
\[ B ::= E < E \]
\[ \mid B \ \text{nand} \ B \]
\[ \mid (B) \]

Again the grammar is ambiguous and \(B_1 \ \text{nand} \ B_2 \ \text{nand} \ B_3\) is \((B_1 \ \text{nand} \ B_2) \ \text{nand} \ B_3\).

Because the expressions in the condition can contain the random sampling \(?\), the result of a test is in general not deterministic and can return either true, or false, or both as in \(? < ?\).
Commands

Commands specify a flow of successive assignments controlled by tests.

\[
C \in C, \quad \text{commands}
\]

\[
C ::= \text{skip} \\
| \quad V := E \\
| \quad \text{if } B \text{ then } C \text{ else } C \text{ fi} \\
| \quad C ; C \\
| \quad \text{while } B \text{ do } C \text{ od}
\]
The strict syntactic component relation (Cont’d)

for commands

\[ V \prec V := E \]
\[ E \prec V := E \]
\[ B \prec \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi} \]
\[ C_1 \prec \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi} \]
\[ C_2 \prec \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi} \]
\[ C_1 \prec C_1 ; C_2 \]
\[ C_2 \prec C_1 ; C_2 \]
\[ B \prec \text{while } B \text{ do } C \text{ od} \]
\[ C \prec \text{while } B \text{ do } C \text{ od} \]

and for programs

\[ C \prec C \].

Because programs are always finite pieces of text, this relation is well-founded and so can be used to prove program properties by induction on their syntax.

---

**Structural definition and proof**

**Definition 11:** Structural Definition: A structural definition is the definition of a program (abstract) property \(^1\) by induction of the program structure using the well-founded relation \(\prec\) on program syntactic components defined in section 13.1.

**Definition 12:** Proof by Structural Induction: A proof by structural induction is a proof by strong recurrence of a property of the semantics of a command \(C\) based on the well-founded strict syntactic component relation \(\prec\).

---

**Example of proof**

The factorial result always strictly positive

- Base case \((n=0)\): \(f(0) = 1 > 0\)
- Induction step \((n>0)\):
  - Assume by induction hypothesis that \(f(n-1) > 0\)
  - Then \(f(n) = n \cdot f(n-1) > 0\) since it is the product of two strictly positive numbers

- The inductive definition and the proof are on the same well-founded relation \(\preceq\).
Expression evaluation

• In general the value of an expression depends on the memory state where the value of variables is stored.
• A memory state can be formalized by an environment / memory state \( \rho \) which is a function mapping variables names in \( \mathcal{V} \) to variable values in \( \mathcal{V} \).

\[
\mathcal{E} := \mathcal{V} \longrightarrow \mathcal{V} \quad \text{set of environments}
\]

\[
\rho \in \mathcal{E} \quad \text{environment}
\]

\[
\rho(x) \quad \text{value of variable } x \in \mathcal{V}
\]

Expression evaluation (cont’d)

- This is a structural definition, which, by theorem 10, is well-defined.

Example

The evaluation of \(? - 1\) starts by the evaluation of \(?\) which returns any natural number \( n \in \mathbb{N} \) then evaluate 1 which returns 1 and then calculate the difference \( n - 1 \) so the expression \(? - 1\) may return any integer greater than or equal to \(-1\).

\[
E[? - 1]_\rho = \{ v_1 - v_2 \mid v_1 \in E[?]_\rho \land v_2 \in E[1]_\rho \} \\
= \{ v_1 - v_2 \mid v_1 \in \mathbb{N} \land v_2 \in \{1\} \} \\
= \{ v \in \mathbb{Z} \mid -1 \leq v \}
\]
Example of structural definition

The set \( E \) of arithmetic expressions was defined by the following grammar.

\[
E ::= 1 \\
    \mid V \\
    \mid ? \\
    \mid E - E \\
    \mid (E)
\]

The formal definition of \( E \in E \mapsto \{E \mapsto \rho(V)\} \) is a structural definition on \( \langle E, \sqsubseteq \rangle \) where the “syntactic component” relation \( \sqsubseteq \) was defined on expressions \( E \in E \) as follows.

\[
E_1 \sqsubseteq E_1 - E_2 \\
E_2 \sqsubseteq E_1 - E_2 \\
E \sqsubseteq (E)
\]

Example of structural definition (cont’d)

The semantics \( E \in E \mapsto \{E \mapsto \rho(V)\} \) of arithmetic expressions was defined as

\[
E[1] \rho := \{1\} \\
E[V] \rho := \{\rho(V)\} \\
E[?] \rho := \mathbb{N} \\
E[E_1 - E_2] \rho := \{v_1 - v_2 \mid v_1 \in E[E_1] \rho \land v_2 \in E[E_2] \rho\} \\
E[(E)] \rho := E[E] \rho
\]

Observe that 1, V, and ? are minimal elements of \( \langle E, \sqsubseteq \rangle \) so that their semantics is defined as a given set. However, \( E_1 - E_2 \) is not minimal, so its semantics is defined as a given function of the semantics of its syntactic components, specifically in this example, \( E_1 \) and \( E_2 \). The function is \( F \) such that \( F(X_1, X_2) \rho = \{v_1 - v_2 \mid v_1 \in X_1(\rho) \land v_2 \in X_2(\rho)\} \) so that \( E[E_1 - E_2] = F(E[E_1], E[E_2]) \).

Evaluation of conditions

Let us use this relation to prove that the evaluation of an expression in an environment where all variables have integer values always return a nonempty set of integers.

So assume that \( \rho(V) \in \mathbb{Z} \) for all \( V \in V \). For the basis, we have:

- \( E[1] \rho \triangleq \{1\} \), which is a nonempty set of integers.
- \( E[V] \rho \triangleq \{\rho(V)\} \), which, by hypothesis on \( \rho \) is a nonempty set of integers.
- \( E[?] \rho \triangleq \mathbb{N} \) which is a nonempty set of integers.

For the induction step, assume that the property holds for all \( E' \sqsubseteq E \). We must prove that this is holds for \( E \). We proceed be cases.

- Either \( E \) has the form \( E_1 - E_2 \) in which case \( E[E_1] \rho \) and \( E[E_2] \rho \) are nonempty sets of integers by induction hypothesis and so \( E[E_1 - E_2] \rho \triangleq \{v_1 - v_2 \mid v_1 \in E[E_1] \rho \land v_2 \in E[E_2] \rho\} \) is a nonempty set of integers.
- Or \( E \) has the form \( (E') \) where \( E[E'] \rho \) is a nonempty set of integers by induction hypothesis hence so is \( E[(E')] \rho \triangleq E[E] \rho \).

Booleans

\[
\mathbb{B} := \{\text{true}, \text{false}\}
\]

\[
\begin{array}{c}
\mathbb{B} \in \mathbb{B} \mapsto \{E \mapsto \rho(\mathbb{B})\} \\
\mathbb{B}[E_1 < E_2] \rho \triangleq \{\text{true} \mid \exists v_1 \in E[E_1] \rho \land \exists v_2 \in E[E_2] \rho : v_1 < v_2\} \\
\cup \{\text{false} \mid \exists v_1 \in E[E_1] \rho \land \exists v_2 \in E[E_2] \rho : v_1 \geq v_2\} \\
\mathbb{B}[B_1 \text{ nand } B_2] \rho \triangleq \{b_1 \text{ nand } b_2 \mid b_1 \in \mathbb{B}[B_1] \rho \land b_2 \in \mathbb{B}[B_2] \rho\} \\
\mathbb{B}[(\mathbb{B})] \rho \triangleq \mathbb{B}[\mathbb{B}] \rho
\end{array}
\]

The function “not and” \( \text{nand} \in (\mathbb{B} \times \mathbb{B}) \mapsto \mathbb{B} \) returns true if and only if at least one of its parameters is false.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x nand y</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
Example
The evaluation of ? < 1 starts by the evaluation of ? which returns any natural number \( n \in \mathbb{N} \), then evaluates 1 which returns 1 and then evaluates the comparison \( n < 1 \) which may be true for \( n = 0 \) or false for \( n \geq 1 \).

\[
B\left[ ? \,<\,1 \right]_\rho = \{ \text{true} \mid \exists v_1 \in E[?] : \exists v_2 \in E[1] : v_1 < v_2 \} \\
\quad \cup \{ \text{false} \mid \exists v_1 \in E[?] : \exists v_2 \in E[1] : v_1 \geq v_2 \} \\
= \{ \text{true} \mid \exists v_1 \in \mathbb{N} : \exists v_2 \in \{1\} : v_1 < v_2 \} \\
\quad \cup \{ \text{false} \mid \exists v_1 \in \mathbb{N} : \exists v_2 \in \{1\} : v_1 \geq v_2 \} \\
= \{ \text{true} \mid v_1 \in \mathbb{N} : v_1 < 1 \} \\
\quad \cup \{ \text{false} \mid v_1 \in \mathbb{N} : v_1 \geq 1 \} \\
= \{ \text{true}, \text{false} \}
\]

since \( 0 \in \mathbb{N} \land 0 < 1 \) and \( 1 \in \mathbb{N} \land 1 \geq 1 \).

Trace semantics

Traces

- Program:
  1: int x;
  2: x := ?; // input(x);
  3: x := x+1;
  4: 

- Example trace

States

- A program state \( s \in \mathcal{S} \) describes an instantaneous observation of a machine executing the program.
- A state (e.g. \( s = <4:|x:6|7> \)) includes
  - the control state (e.g. 4) that is the program point where machine execution is currently located
  - the memory state that is the current content of the parts of the memory used during program execution (e.g. x:6 more generally stack and heap)
  - the environment state describing the physical condition of the execution environment (e.g. inputs 7)
**Actions**

- An action \( a \in \mathcal{A} \) is an elementary program command performing a single computation step (e.g. skip, assignment, test of a condition, input of a value, exit, etc).

**Completed traces**

- A trace \( t \in \mathcal{T} \) is a non-empty finite or infinite sequence of states \( s_i \in \mathcal{S} \) separated by actions \( a_i \in \mathcal{A} \)

\[
\begin{align*}
S_0 - a_0 & \rightarrow S_1 - a_1 \rightarrow S_2 \ldots S_n \\
S_0 - a_0 & \rightarrow S_1 - a_1 \rightarrow S_2 \ldots S_n - a_n \rightarrow S_{n+1} \ldots
\end{align*}
\]

- This trace describes a program run
  - starting is state \( S_0 \)
  - when reaching state \( S_n \)
  - either \( S_n \) is final and execution stops
  - else action \( a_n \) yields next state \( S_{n+1} \) and so on.

**Program trace semantics**

- The semantics \( S \) of a program \( P \) is the set of traces modeling all possible executions of the program in all possible environments/contexts.

**On computer and semantics**

- A semantics is an infinite set of finite or infinite traces

- Given a finite trace, a computer can check that the trace is in the semantics (by executing the program)

- Given an infinite trace, a computer can check that it is not in the semantics (but not that it belongs to the semantics since this would take an infinite time)

- A computer cannot check that a set of traces is a semantics of a language or represent/compute this semantics (since the semantics and some of its traces are infinite)

- Semantics are therefore a mathematical, non-computable object.
**Inputs**

- Inputs have been represented “on the fly” in traces:

  \[
  \langle 1:|\rangle \rightarrow \text{int } x \rightarrow \langle 2:|x:1789|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|5\rangle \rightarrow \text{input}(x) \rightarrow \langle 3:|x:5|5\rangle \rightarrow x := x+1 \rightarrow \langle 4:|x:6|7\rangle 
  \]

- We could isomorphically represent in each states all the future inputs (say guessed by an oracle):

  \[
  \langle 1:||\rangle \rightarrow \text{int } x \rightarrow \langle 2:|x:1789|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|5\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|5,5,7,... \rangle \rightarrow \text{input}(x) \rightarrow \langle 3:|x:5|5,7,... \rangle \rightarrow x := x+1 \rightarrow \langle 4:|x:6|7,... \rangle 
  \]

**Outputs**

- Similarly we could represent the sequence of past outputs (empty in our example):

  \[
  \langle 1:|\rangle \rightarrow \text{int } x \rightarrow \langle 2:|x:1789|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|5,5,7,... \rangle \rightarrow \text{input}(x) \rightarrow \langle 3:|x:5|5,7,... \rangle \rightarrow \text{output} : \rightarrow x := x+1 \rightarrow \langle 4:|x:6|\rangle 
  \]

- Then, input and output can be handled as ordinary variables like \( x \), with the only difference that their values are sequences

**Silent actions**

- Some actions like “waiting for a key to be depressed” have been represented in traces to account for passing time, although they do not contribute to the result of computations

  \[
  \langle 2:|x:1789|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|5\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|5,5,7,... \rangle \rightarrow \text{input}(x) \rightarrow \langle 3:|x:5|5,7,... \rangle \rightarrow x := x+1 \rightarrow \langle 4:|x:6|7,... \rangle 
  \]

- It time is not observed such silent actions can simply be eliminated

- Similarly silent inputs (no key depressed, key depressed while no input, etc) can be eliminated from the input

**Maximal traces**

- A trace \( t \in T \) is a non-empty finite or infinite sequence of states \( s_i \) separated by actions \( a_i \)

  \[
  s_0 \rightarrow a_0 \rightarrow s_1 \rightarrow a_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \rightarrow (n \geq 0)
  \]

  \[
  s_0 \rightarrow a_0 \rightarrow s_1 \rightarrow a_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \rightarrow a_n \rightarrow s_{n+1} \rightarrow \ldots 
  \]

- The states describe the control, values of variables, past outputs and future effective inputs

- All actions effectively contribute to computations

- Example:

  \[
  \langle 1:|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:1789|\rangle \rightarrow \text{input}(x) \rightarrow \langle 2:|x:5|5;\rangle \rightarrow \text{output} : \rightarrow x := x+1 \rightarrow \langle 3:|x:6|\rangle 
  \]
**States, formally**

- **States** are pairs \( <c|\rho> \in \mathcal{C} \times \mathcal{E} \) (or \( <c, \rho> \)) where
  - \( c \in \mathcal{C} \) is the **control state** chosen in a finite set \( \mathcal{C} \) of program labels
  - \( \rho \in \mathcal{E} \) is a **memory state/environment** mapping program variables \( x \) (including input and output) to their values \( \rho(x) \)
  - \( \mathcal{E} := \mathcal{V} \leftarrow \mathcal{V} \)

- We use the symbol "\( := \)" to mean "is defined as".

**Control encoding**

- Instead of representing control points by program labels, we can represent them by the part of the program that remains to execute from that program point

**Example:**

1: \( x := 1; \)
2: while \( (x < 10) \) {
3: \( x := x+1; \)
4: }
5: \( \epsilon \) (empty string)

**Transition relation**

- We can define the transition relation \( \tau \) between a state and its possible successors:

\[
\tau := \{ <s,s'> | \exists s_0 - a_0 \rightarrow \ldots s = s_i - a_i \rightarrow s' = s_{i+1} \ldots s_n \in S \}
\]

**Transition relation, formal definition**
**States**

- $V$: set of variables
- $\mathcal{V}$: set of values of variables (e.g. $\mathbb{Z} \subseteq \mathcal{V}$)
- $E := V \mapsto \mathcal{V}$, environments/memory states
- $A$, actions
- $C$, commands
- $S := C \times E$, states

**Transition relation**

- The transition relation is
  \[ \tau \in \mathcal{P}(S \times A \times S) \]
- $<s, a, s'> \in \tau$ if and only if an execution of action $a$ in state $s$ may yield next state $s'$
- In general a state $s$ has many possible successor states $s'$ for a given action $a$ (non-determinism) e.g. $x := ?$
- We write $s = a \Rightarrow s'$ for $<s, a, s'> \in \tau$

**Assignment**

- $<\text{skip}, \rho> = \text{skip} \Rightarrow <\varepsilon, \rho>

**Conditional**

- $true \in B[|B|] \rho$
  \[ <\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi}, \rho> = B \Rightarrow <C_1, \rho> \]

- $false \in B[|B|] \rho$
  \[ <\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi}, \rho> = \neg B \Rightarrow <C_2, \rho> \]

- Well-defined by structural induction
- Non-deterministic if $B$ is non-deterministic
Sequential composition

- \(<C_1, \rho> = a \Rightarrow <C'_1, \rho'>\)
- \(<C_1; C_2, \rho> = a \Rightarrow <C'_1; C_2, \rho'>\)
- Well-defined by structural induction
- Examples:
  - \(<x := x + 1, \rho>\) and so \(<x := x + 2, \rho>\)
  - \(<\epsilon, \rho[x := \rho(x) + 1]>\)

Iteration

- \(\text{true} \in B [B] \rho\)
- \(<\text{while } B \text{ do } C \text{ od, } \rho> = B \Rightarrow <C; \text{while } B \text{ do } C \text{ od, } \rho>\)
- \(\text{false} \in B [B] \rho\)
- \(<\text{while } B \text{ do } C \text{ od, } \rho> = \neg B \Rightarrow <\epsilon, \rho>\)
- Well-defined by structural induction
- Non-deterministic if \(B\) is non-deterministic

Traces

- Finite traces
  - \(T^+ [C] := \{s_0 \xrightarrow{a_0} \ldots s_i \xrightarrow{a_i} s_{i+1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n | \exists \rho_0: s_0 = <C, \rho_0> \land n \geq 0 \land \forall i < n: s_i = a_i \Rightarrow s_{i+1} \land \exists \rho_n: s_n = <\epsilon, \rho_n>\}\)
- Infinite traces
  - \(T^\infty [C] :=\{s_0 = <C, \rho_0> \rightarrow \ldots s_i \rightarrow a_i \rightarrow s_{i+1} \ldots | \exists \rho_0: s_0 = <C, \rho_0> \land \forall i \geq 0: s_i = a_i \Rightarrow s_{i+1}\}\)
- Traces
  - \(T[C] := T^+ [C] \cup T^\infty [C]\)
(Facultative) Homework

Homework (facultative)

- Extend the language with commands
  
  \[\text{input}(\text{x})\]  
  \[\text{output}(\text{x})\]

for input/output of the value of variable \(x\) and define
- the revised notion of states to include future inputs and past outputs, as well as
- the transition semantics for these commands.

Conclusion

- **Formal methods** use mathematics to define and prove properties of programs
- This requires to define the **semantics** of programming languages i.e. to provide a model of all possible executions of all programs in the language
- This semantics can be defined by **structural induction** on the syntax of programs
References on mathematics

(Malitz, 1987) is an elementary introduction to logic, (Halmos, 1960) is a basic introduction to naïve set theory, and (Jones and Jones, 1998) is an elementary introduction to number theory.

(Chomsky, 1956) introduced grammars. The equivalence with language-

attribute grammars was established by (Malitz, 1987).


References on languages and semantics


The End