

Formal Methods and Functional Programming

Introduction to Language Semantics

Peter Müller

Chair of Programming Methodology
ETH Zurich

2. Introduction to Language Semantics

2.1 The Language IMP

2.2 Semantics of IMP Expressions

2.3 Properties of the Expression Semantics

The Language IMP

- Expressions
 - Boolean and arithmetic expressions
 - No side-effects in expressions
- Variables
 - All variables range over integers
 - All variables are initialized
- IMP does not include
 - Heap allocation and pointers
 - Variable declarations
 - Procedures
 - Concurrency
 - (But, we will discuss some as extensions later)

Syntax of IMP: Example

```
y := 1;  
while x > 1 do  
  y := y * x;  
  x := x - 1  
end
```

Concrete Syntax of IMP: Characters and Tokens

Characters

```
Letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
Digit  = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Tokens

```
Ident  = Letter { Letter | Digit }*
Numeral = Digit | Numeral Digit
Var    = Ident
```

Concrete and Abstract Syntax of IMP: Expressions

Arithmetic expressions

```
Aexp = '(' Aexp Op Aexp ')'
      | Var
      | Numeral
Op    = '+' | '-' | '*'
```

```
data Aexp = Bin Op Aexp Aexp
          | Var String
          | Num Integer
data Op    = Add | Sub | Mul
```

Boolean expressions

```
Bexp = '(' Bexp 'or' Bexp ')'
      | '(' Bexp 'and' Bexp ')'
      | 'not' Bexp
      | Aexp Rop Aexp
Rop   = '=' | '#' | '<'
      | '<=' | '>' | '>='
```

```
data Bexp = Or Bexp Bexp
          | And Bexp Bexp
          | Not Bexp
          | Rel Rop Aexp Aexp
data Rop   = Eq | Neq | Le
          | Leq | Ge | Geq
```

Concrete and Abstract Syntax of IMP: Statements

```
Stm = 'skip'
      | Var ':=' Aexp
      | '(' Stm ';' Stm ')'
      | 'if' Bexp 'then' Stm 'else' Stm 'end'
      | 'while' Bexp 'do' Stm 'end'
```

```
data Stm = Skip
          | Assign String Aexp
          | Seq Stm Stm
          | If Bexp Stm Stm
          | While Bexp Stm
```

We omit parentheses if permitted by the usual operator precedence

Meta-variables vs. Program Variables

- In proofs, we often need to reason about program variables without specifying their concrete name
 - For example, " $\forall x. P(x)$ " doesn't make sense since x is a fixed symbol in the source language syntax
- We use meta-variables to denote some program variable (concrete name unspecified)

Meta-variables vs. Program Variables

- In proofs, we often need to reason about program variables without specifying their concrete name
 - For example, “ $\forall x. P(x)$ ” doesn’t make sense since x is a fixed symbol in the source language syntax
- We use meta-variables to denote some program variable (concrete name unspecified)
- We write \equiv for syntactic equality on variables, statements, etc.
- For program variables x and y , $x = y$ might evaluate to true in some states, but $x \equiv y$ is always false (not syntactically equal)
- For meta-variables x and y , $x \equiv y$ might be true (i.e., they both could denote the same program variable)

2. Introduction to Language Semantics

2.1 The Language IMP

2.2 Semantics of IMP Expressions

2.3 Properties of the Expression Semantics

Notation

- We follow naming conventions for meta-variables:

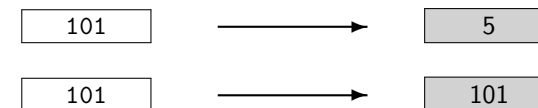
n	for numerals (Numeral)
x, y, z	for variables (Var)
e, e', e_1, e_2	for arithmetic expressions (Aexp)
b, b_1, b_2	for boolean expressions (Bexp)
s, s', s_1, s_2	for statements (Stm)

- Meta-variables are written in math font (e.g., x, y, e, \dots)
- Program variables are written in typewriter font
- We use the naming conventions to avoid the need for explicit types (especially in proofs)
 - For example, when we write $\forall x. P(x)$, we mean $\forall x \in \text{Var}. P(x)$

Semantic Functions

Syntactic category: Numeral

Semantic category: $\text{Val} = \mathbb{Z}$



- Semantic functions map elements of syntactic categories to elements of semantic categories
- To define the semantics of IMP, we need semantic functions for
 - Numerals (syntactic category Numeral)
 - Arithmetic expressions (syntactic category Aexp)
 - Arithmetic operators (syntactic category Op)
 - Boolean expressions (syntactic category Bexp)
 - Relational operators (syntactic category Rop)
 - Statements (syntactic category Stm)

Semantics of Numerals

The semantic function

$$\mathcal{N} : \text{Numeral} \rightarrow \text{Val}$$

maps a numeral n to an integer value $\mathcal{N}[[n]]$

$$\begin{array}{ll} \mathcal{N}[[0]] &= 0 & \mathcal{N}[[1]] &= 1 \\ \dots & & \dots & \\ \mathcal{N}[[8]] &= 8 & \mathcal{N}[[9]] &= 9 \\ \mathcal{N}[[n\ 0]] &= \mathcal{N}[[n]] \times 10 + 0 & \mathcal{N}[[n\ 1]] &= \mathcal{N}[[n]] \times 10 + 1 \\ \dots & & \dots & \\ \mathcal{N}[[n\ 8]] &= \mathcal{N}[[n]] \times 10 + 8 & \mathcal{N}[[n\ 9]] &= \mathcal{N}[[n]] \times 10 + 9 \end{array}$$

Constructing and Comparing States

- We define a designated (constant) state σ_{zero} , in which all variables have the value 0:

$$\sigma_{\text{zero}}(x) = 0 \text{ for all } x$$

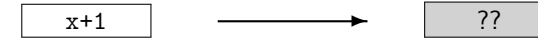
- Updating States: $\sigma[y \mapsto v]$ is the function that overrides the association of y in σ by $y \mapsto v$.

$$(\sigma[y \mapsto v])(x) = \begin{cases} v & \text{if } x \equiv y \\ \sigma(x) & \text{if } x \not\equiv y \end{cases}$$

- Two states σ_1 and σ_2 are equal if they are equal as functions:

$$\sigma_1 = \sigma_2 \iff \forall x. (\sigma_1(x) = \sigma_2(x))$$

States



- The meaning of an expression depends on the values bound to the variables that occur in it
- A state is a total function, associating a value with each program variable

$$\text{State} : \text{Var} \rightarrow \text{Val}$$

- We use σ as a meta-variable for states

Semantics of Arithmetic Expressions

The semantic function

$$\mathcal{A} : \text{Aexp} \rightarrow \text{State} \rightarrow \text{Val}$$

maps an arithmetic expression e and a state σ to a value $\mathcal{A}[[e]]\sigma$

$$\begin{array}{ll} \mathcal{A}[[x]]\sigma &= \sigma(x) \\ \mathcal{A}[[n]]\sigma &= \mathcal{N}[[n]] \\ \mathcal{A}[[e_1\ op\ e_2]]\sigma &= \mathcal{A}[[e_1]]\sigma\ \overline{op}\ \mathcal{A}[[e_2]]\sigma \text{ for } op \in \text{Op} \end{array}$$

\overline{op} is the operation $\text{Val} \times \text{Val} \rightarrow \text{Val}$ corresponding to op

Semantics of Boolean Expressions

The semantic function

$$\mathcal{B} : \text{Bexp} \rightarrow \text{State} \rightarrow \text{Bool}$$

maps a boolean expression b and a state σ to a truth value $\mathcal{B}[[b]]\sigma$

$$\mathcal{B}[[e_1 \text{ op } e_2]]\sigma = \begin{cases} tt & \text{if } \mathcal{A}[[e_1]]\sigma \overline{\text{op}} \mathcal{A}[[e_2]]\sigma \\ ff & \text{otherwise} \end{cases}$$

$\text{op} \in \text{Rop}$; $\overline{\text{op}}$ is the relation $\text{Val} \times \text{Val}$ corresponding to op

2. Introduction to Language Semantics

2.1 The Language IMP

2.2 Semantics of IMP Expressions

2.3 Properties of the Expression Semantics

Boolean Expressions (cont'd)

$$\begin{aligned} \mathcal{B}[[b_1 \text{ or } b_2]]\sigma &= \begin{cases} tt & \text{if } \mathcal{B}[[b_1]]\sigma = tt \text{ or } \mathcal{B}[[b_2]]\sigma = tt \\ ff & \text{otherwise} \end{cases} \\ \mathcal{B}[[b_1 \text{ and } b_2]]\sigma &= \begin{cases} tt & \text{if } \mathcal{B}[[b_1]]\sigma = tt \text{ and } \mathcal{B}[[b_2]]\sigma = tt \\ ff & \text{otherwise} \end{cases} \\ \mathcal{B}[[\text{not } b]]\sigma &= \begin{cases} tt & \text{if } \mathcal{B}[[b]]\sigma = ff \\ ff & \text{otherwise} \end{cases} \end{aligned}$$

Inductive Definitions

The semantics is given by recursive definitions of functions \mathcal{A} and \mathcal{B}

- The values for the basic elements (e.g., x) are defined directly
- The values for composite elements are defined inductively in terms of the immediate constituents

$$\begin{aligned} \mathcal{A}[[x]]\sigma &= \sigma(x) \\ \mathcal{A}[[n]]\sigma &= \mathcal{N}[[n]] \\ \mathcal{A}[[e_1 \text{ op } e_2]]\sigma &= \mathcal{A}[[e_1]]\sigma \overline{\text{op}} \mathcal{A}[[e_2]]\sigma \quad \text{for } \text{op} \in \text{Op} \end{aligned}$$

- Since the decomposition of the elements is unique, the definition gives a well-defined function (proof shortly)
- Inductive definitions suggest proofs by structural induction

Reminder: Structural Induction

```
data Nat = Zero | Succ Nat
```

- Induction over structure of terms

$$\frac{\Gamma \vdash P(\text{Zero}) \quad \Gamma, P(n) \vdash P(\text{Succ } n)}{\Gamma \vdash \forall n \in \text{Nat}. P(n)} \quad n \text{ not free in } \Gamma$$

- Sufficient to show $P(\text{Zero})$, $P(\text{Succ Zero})$, ...
- Useful (see first half of the course) to prove theorems like

$$\forall x, y, z \in \text{Nat}. \text{plus } x \text{ (plus } y \text{ } z) = \text{plus (plus } x \text{ } y) \text{ } z$$

- Structural induction on x helps because *plus* is defined inductively (over its first argument)

Using Structural Induction

- Lemma: The equations for \mathcal{N} define a total function $\mathcal{N} : \text{Numeral} \rightarrow \text{Val}$
- To prove the lemma, we show that for each $n \in \text{Numeral}$ there is exactly one $v \in \text{Val}$ such that $\mathcal{N}[[n]] = v$
- \mathcal{N} is defined inductively over the structure of the numeral:

$\mathcal{N}[[0]] = 0$	$\mathcal{N}[[1]] = 1$
...	
$\mathcal{N}[[8]] = 8$	$\mathcal{N}[[9]] = 9$
$\mathcal{N}[[n\ 0]] = \mathcal{N}[[n]] \times 10 + 0$	$\mathcal{N}[[n\ 1]] = \mathcal{N}[[n]] \times 10 + 1$
...	
$\mathcal{N}[[n\ 8]] = \mathcal{N}[[n]] \times 10 + 8$	$\mathcal{N}[[n\ 9]] = \mathcal{N}[[n]] \times 10 + 9$

- This suggests proving the lemma by structural induction on n

Structural Induction over Programs

- Recall: abstract syntax is defined as algebraic data type

```
data Aexp = Bin Op Aexp Aexp
          | Var String
          | Num Integer
data Op   = Add | Sub | Mul
```

- We can prove properties of syntactic elements via structural induction
- In proofs, we will use the concrete syntax instead of the terms of the abstract syntax to identify the cases

$$\frac{\Gamma \vdash P(x) \quad \Gamma \vdash P(n) \quad \Gamma, P(e_1), P(e_2) \vdash P(e_1 \text{ op } e_2)}{\Gamma \vdash \forall e \in \text{Aexp}. P(e)} \quad (*)$$

$^*(x, n, e_1, e_2, \text{op} \text{ not free in } \Gamma)$

Proof: \mathcal{N} is a Total Function

- Case:** $n \equiv d$, for some digit d
There are ten further cases for the ten different possible digits. In each case, \mathcal{N} maps d to exactly one value in Val .
- Case:** $n \equiv n_1\ d$, for some numeral n_1 and digit d
We show here the case $d \equiv 0$ (the other nine cases are analogous).
 - Our induction hypothesis (for n_1) tells us: there is exactly one $v_1 \in \text{Val}$ such that $\mathcal{N}[[n_1]] = v_1$
 - The equations for \mathcal{N} define $\mathcal{N}[[n_1\ 0]] = \mathcal{N}[[n_1]] \times 10 + 0$
 - Therefore, $\mathcal{N}[[n]] = v_1 \times 10 + 0$
 - Since multiplication and addition are total functions, we can conclude that there is exactly one value for $v_1 \times 10 + 0$ and, thus, for $\mathcal{N}[[n]]$

\mathcal{A} is a Total Function

- Lemma: The equations for \mathcal{A} define a total function
 $\mathcal{A} : \text{Aexp} \rightarrow \text{State} \rightarrow \text{Val}$
- To prove the lemma, we show that for each $e \in \text{Aexp}$ and $\sigma \in \text{State}$ there is exactly one $v \in \text{Val}$ such that $\mathcal{A}[[e]]\sigma = v$
- \mathcal{A} is defined inductively over the structure of the expression:

$$\begin{aligned}\mathcal{A}[[x]]\sigma &= \sigma(x) \\ \mathcal{A}[[n]]\sigma &= \mathcal{N}[[n]] \\ \mathcal{A}[[e_1 \text{ op } e_2]]\sigma &= \mathcal{A}[[e_1]]\sigma \overline{\text{op}} \mathcal{A}[[e_2]]\sigma \quad \text{for } \text{op} \in \text{Op}\end{aligned}$$

- This suggests a proof by structural induction on e

Inductive Definitions: Example

New arithmetic expression: $\neg e$

- | | |
|---|--|
| <ul style="list-style-type: none"> • Inductive definition of $\mathcal{A}[[\neg e]]\sigma$ <div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> $\mathcal{A}[[\neg e]]\sigma = 0 - \mathcal{A}[[e]]\sigma$ </div> <ul style="list-style-type: none"> • e is a subterm of $\neg e$ • When proving a property by structural induction on Aexp, the case for $\neg e$ may assume the induction hypothesis for e | <ul style="list-style-type: none"> • Non-inductive definition of $\mathcal{A}[[\neg e]]\sigma$ <div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> $\mathcal{A}[[\neg e]]\sigma = \mathcal{A}[[0 - e]]\sigma$ </div> <ul style="list-style-type: none"> • $0 - e$ is not a subterm of $\neg e$ • When proving a property by structural induction on Aexp, the case for $\neg e$ must not assume the induction hypothesis for $0 - e$ |
|---|--|

Proof: \mathcal{A} is a Total Function

1. **Case:** $e \equiv n$, for some numeral n
The equations define $\mathcal{A}[[n]]\sigma = \mathcal{N}[[n]]$. By the previous lemma, \mathcal{N} is a total function and, thus, $\mathcal{N}[[n]]$ yields exactly one value in Val
2. **Case:** $e \equiv x$, for some variable x
The equations define $\mathcal{A}[[x]]\sigma = \sigma(x)$. σ is a total function, $\sigma(x) \in \text{Val}$
3. **Case:** $e \equiv e_1 \text{ op } e_2$, for some e_1, e_2 and $\text{op} \in \text{Op}$
 - Our induction hypothesis gives us the property for e_1 and e_2 (sub-terms of e)
 - By applying the induction hypothesis for both e_1 and e_2 , we get:
 - (a) there is exactly one $v_1 \in \text{Val}$ such that $\mathcal{A}[[e_1]]\sigma = v_1$ and
 - (b) there is exactly one $v_2 \in \text{Val}$ such that $\mathcal{A}[[e_2]]\sigma = v_2$
 - The equations for \mathcal{A} define $\mathcal{A}[[e_1 \text{ op } e_2]]\sigma = \mathcal{A}[[e_1]]\sigma \overline{\text{op}} \mathcal{A}[[e_2]]\sigma$
 - By using (a) and (b), we get: $\mathcal{A}[[e_1 \text{ op } e_2]]\sigma = v_1 \overline{\text{op}} v_2$
 - Since $\overline{\text{op}}$ is a total function (addition, subtraction, or multiplication), we can conclude that there is exactly one value for $v_1 \overline{\text{op}} v_2$ and, thus, for $\mathcal{A}[[e_1 \text{ op } e_2]]\sigma$

Free Variables

Arithmetic expressions

$$\begin{aligned}FV(e_1 \text{ op } e_2) &= FV(e_1) \cup FV(e_2) \\ FV(n) &= \emptyset \\ FV(x) &= \{x\}\end{aligned}$$

Boolean expressions

$$\begin{aligned}FV(e_1 \text{ op } e_2) &= FV(e_1) \cup FV(e_2) \\ FV(\text{not } b) &= FV(b) \\ FV(b_1 \text{ or } b_2) &= FV(b_1) \cup FV(b_2) \\ FV(b_1 \text{ and } b_2) &= FV(b_1) \cup FV(b_2)\end{aligned}$$

Statements

$$\begin{aligned}FV(\text{skip}) &= \emptyset \\ FV(x := e) &= \{x\} \cup FV(e) \\ FV(s_1 ; s_2) &= FV(s_1) \cup FV(s_2) \\ FV(\text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}) &= FV(b) \cup FV(s_1) \cup FV(s_2) \\ FV(\text{while } b \text{ do } s \text{ end}) &= FV(b) \cup FV(s)\end{aligned}$$

Substitution

Substitution “ $_ [x \mapsto e]$ ” replaces each free occurrence of variable x by e

- Arithmetic expressions

$$\begin{aligned}(e_1 \text{ op } e_2)[x \mapsto e] &\equiv (e_1[x \mapsto e] \text{ op } e_2[x \mapsto e]) \\ n[x \mapsto e] &\equiv n \\ y[x \mapsto e] &\equiv \begin{cases} e & \text{if } x \equiv y \\ y & \text{otherwise} \end{cases}\end{aligned}$$

- Boolean expressions

$$\begin{aligned}(e_1 \text{ op } e_2)[x \mapsto e] &\equiv (e_1[x \mapsto e] \text{ op } e_2[x \mapsto e]) \\ (\text{not } b)[x \mapsto e] &\equiv \text{not } (b[x \mapsto e]) \\ (b_1 \text{ or } b_2)[x \mapsto e] &\equiv (b_1[x \mapsto e] \text{ or } b_2[x \mapsto e]) \\ (b_1 \text{ and } b_2)[x \mapsto e] &\equiv (b_1[x \mapsto e] \text{ and } b_2[x \mapsto e])\end{aligned}$$

- We will use the following substitution lemma (see exercises for proof):

$$\mathcal{B}[[b[x \mapsto e]]]\sigma \Leftrightarrow \mathcal{B}[[b]]\sigma[x \mapsto \mathcal{A}[[e]]\sigma]$$

Syntactic Abbreviations

if b then s end	if b then s else skip end
---------------------	-------------------------------

true	1=1
------	-----

false	0=1
-------	-----