Formal Methods and Functional Programming

Introduction to Language Semantics

Peter Müller

Chair of Programming Methodology
ETH Zurich

2. Introduction to Language 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: Statemens

```
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, "$\forall \mathtt{x}.P(\mathtt{x})$" doesn't make sense since $\mathtt{x}$ 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, "$\forall \mathtt{x}.P(\mathtt{x})$" doesn't make sense since $\mathtt{x}$ 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 $\mathtt{x}$ and $\mathtt{y}$, $\mathtt{x} = \mathtt{y}$ might evaluate to true in some states, but $\mathtt{x} \equiv \mathtt{y}$ 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)

Notation

- We follow naming conventions for meta-variables:

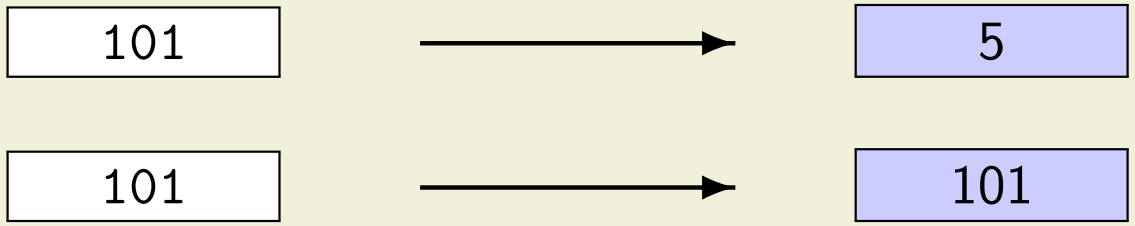
n	for numerals (Numeral)
x, y, z	for variables (Var)
e, e', e_1, e_2	for arithmetic expressions (Aexp)
b, b_1, b_2	for boolean expressions (Bexp)
s, s', s_1, s_2	for statements (Stm)

- Meta-variables are written in math font (e.g., x, y, e, ...)
- 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 $\forall x \in \mathit{Var}.P(x)$

2. Introduction to Language Semantics

Semantic Functions

Syntactic category: Numeral Semantic category: 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

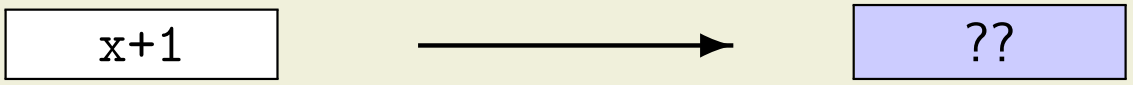
The semantic function

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

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

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

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

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

- We use σ as a meta-variable for states

Constructing and Comparing States

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

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

- 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:

$$\sigma_1 = \sigma_2 \quad \Leftrightarrow \quad \forall x.(\sigma_1(x) = \sigma_2(x))$$

Semantics of Arithmetic Expressions

The semantic function

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

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

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

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

Semantics of Boolean Expressions

The semantic function

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

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

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

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

Boolean Expressions (cont'd)

$$\mathcal{B}[\![b_1 \texttt{ or } b_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b_1]\!]\sigma = tt \text{ or } \mathcal{B}[\![b_2]\!]\sigma = tt \\ ff & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![b_1 \texttt{ and } b_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b_1]\!]\sigma = tt \text{ and } \mathcal{B}[\![b_2]\!]\sigma = tt \\ ff & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![\texttt{not } b]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![b]\!]\sigma = ff \\ ff & \text{otherwise} \end{cases}$$

2. Introduction to Language Semantics

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

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

- 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

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

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

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

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

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

$$\frac{\Gamma \vdash P(x) \quad \Gamma \vdash P(n) \quad \Gamma, P(e_1), P(e_2) \vdash P(e_1\ op\ e_2)}{\Gamma \vdash \forall e \in Aexp.P(e)}\ (^*)$$
$$^*(x, n, e_1, e_2, op\ not\ free\ in\ \Gamma)$$

Using Structural Induction

- Lemma: The equations for $\mathcal{N}$ define a total function $\mathcal{N} : \text{Numeral} \to \text{Val}$

- 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:

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

- This suggests proving the lemma by structural induction on n

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

- 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:

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

- This suggests a proof by structural induction on 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:** $e \equiv e_1\ op\ e_2$, for some e_1, e_2 and $op \in Op$
   - 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 $\mathcal{A}[\![e_1\ op\ e_2]\!]\sigma = \mathcal{A}[\![e_1]\!]\sigma\ \overline{op}\ \mathcal{A}[\![e_2]\!]\sigma$
   - By using (a) and (b), we get: $\mathcal{A}[\![e_1\ op\ e_2]\!]\sigma = v_1\ \overline{op}\ v_2$
   - Since $\overline{op}$ is a total function (addition, subtraction, or multiplication), we can conclude that there is exactly one value for $v_1\ \overline{op}\ v_2$ and, thus, for $\mathcal{A}[\![e_1\ op\ e_2]\!]\sigma$

Inductive Definitions: Example

New arithmetic expression: $-e$

- Inductive definition of $\mathcal{A}[\![-e]\!]\sigma$

$$\mathcal{A}[\![-e]\!]\sigma = 0 - \mathcal{A}[\![e]\!]\sigma$$

- e is a subterm of $-e$
- When proving a property by structural induction on Aexp, the case for $-e$ may assume the induction hypothesis for e

- Non-inductive definition of $\mathcal{A}[\![-e]\!]\sigma$

$$\mathcal{A}[\![-e]\!]\sigma = \mathcal{A}[\![0-e]\!]\sigma$$

- $0-e$ is not a subterm of $-e$
- When proving a property by structural induction on Aexp, the case for $-e$ must not assume the induction hypothesis for $0-e$

Free Variables

Arithmetic expressions

$$
\begin{array}{ll}
FV(e_1\ op\ e_2) & = FV(e_1) \cup FV(e_2) \\
FV(n) & = \varnothing \\
FV(x) & = \{x\}
\end{array}
$$

Boolean expressions

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

Statements

$$
\begin{array}{ll}
FV(\texttt{skip}) & = \varnothing \\
FV(x\texttt{:=}e) & = \{x\} \cup FV(e) \\
FV(s_1\texttt{;}s_2) & = FV(s_1) \cup FV(s_2) \\
FV(\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}) & = FV(b) \cup FV(s_1) \cup FV(s_2) \\
FV(\texttt{while}\ b\ \texttt{do}\ s\ \texttt{end}) & = FV(b) \cup FV(s)
\end{array}
$$

Substitution

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

- Arithmetic expressions

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

- Boolean expressions

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

- 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`