Imperative Specification
Contents

- variable definitions
- function types revisited
- statements as imperative expressions
- equivalence versus equality, revisited
- conditional equivalence expressions
- evaluation order
- pure and readonly requirements
- axioms revisited
- call of imperative functions
- if expressions revisited
- while, until and for loops
- post expressions revisited
- from applicative to imperative

Terminology

Imperative: With variables

Applicative: Without variables

Imperative Specification
Example

scheme COUNTER =
  class
    variable
      counter : **Nat** := 0
    value
      increase : **Unit** → **write** counter **Nat**
      increase() ≡ counter := counter + 1 ; counter
  end

Variable Definitions

variable
  $\text{variable\_definition}_1$,
  ⋮
  $\text{variable\_definition}_n$

single variable definition:

  id : type_expr := value_expr

multiple variable definition:

  $\text{id}_1,\ldots,\text{id}_n$: type_expr

Functions with Variable Access

Function types revisited:

  $\text{type\_expr}_1 \xrightarrow{\sim} \text{access\_desc}_1 \ldots \text{access\_desc}_n\ \text{type\_expr}_2$

access_desc_i:

- **read** $\text{id}_1,\ldots,\text{id}_n$
- **write** $\text{id}_1,\ldots,\text{id}_n$

Function definitions:

bodies must not statically access variables
which are not mentioned in the access descriptions

Imperative Expressions

No syntactic distinction between

- statements and
- expressions

Imperative expressions ($\sim$ statements):

- assignments (id := value_expr)
- sequencing (*unit*-value_expr$_1$; value_expr$_2$)
- conditionals (if, case)
- loops (while, until, for)
- ...

Imperative expressions may have side-effects

side-effect = state change
state = particular contents of variables

All expressions return a value

Exercise

Complete the table below.

expression	return value	side-effect
1 + 5		
x := 3		
x := 3 ; x		
x := 3 ; x + 2		
x := 3 ; x := x + 1;		

Equivalence versus Equality
Revisited

- $\equiv$ and $=$ is the same if
  the arguments are convergent and pure
- non-convergency has been discussed
- $\equiv$ compares side-effects as well as results;
  $=$ only compares results.
- $\equiv$ has simultaneous evaluation;
  $=$ has left-to-right evaluation.
- $\equiv$ gives no side-effects;
  $=$ may give side-effects.

Example 1:

In a state where x = 0

$$(x := x + 1 ; 1) \equiv (x := x + 1 ; x)$$

is equivalent to **true**

and

$$(x := x + 1 ; 1) = (x := x + 1 ; x)$$

is equivalent to x := x + 2; **false**

Conditional Equivalence Expressions

value_expr$_1$ $\equiv$ value_expr$_2$ **pre** value_expr$_3$

is short for:

(value_expr$_3$ $\equiv$ **true**) $\Rightarrow$ (value_expr$_1$ $\equiv$ value_expr$_2$)

Example:

scheme DECREASE =
  extend COUNTER **with**
  class
    value
      decrease : **Unit** $\xrightarrow{\sim}$ **write** counter **Nat**
      decrease() $\equiv$ counter := counter $-$ 1 ; counter
      pre counter $>$ 0
  end

Evaluation Order

The evaluation order is *left to right*

e.g. in $\langle e_1, \ldots, e_n \rangle$
and now (after language revision) also for $e(e_1, ..., e_2)$

The evaluation order has importance
when the constituent expressions have side-effects

Examples:

Given **variable** x : **Int**

$\langle x := 1 ; x \ , x := 2 ; x \rangle \equiv x := 2 ; \langle 1,2 \rangle$
$\langle x := 2 ; x \ , x := 1 ; x \rangle \equiv x := 1 ; \langle 2,1 \rangle$

$x + (x := x + 1 ; x) \equiv x := x + 1 ; 2 * x - 1$
$(x := x + 1 ; x) + x \equiv x := x + 1 ; 2 * x$

Pure and Read-only

Properties of value expressions:

- *read* a variable
- *write* to a variable (e.g. x := e writes to x)
- *access* a variable: read or write to a variable
- *pure*: does not statically access any variable
  (e.g. 5)
- *read-only*: does not statically write to any variable
  (e.g. 5, x + 1)

Context conditions:

- must be pure

  {| binding : type_expr • *pure*−value_expr |}

- must be read-only

  { *readonly*−value_expr$_1$ .. *readonly*−value_expr$_2$ }

Axioms
revisited

Axioms must be read-only

axiom value_expr

short for:

axiom □value_expr

means: value_expr should be true in all states.

Example:

variable x : **Int**
axiom x = 7

is false, since x is not 7 in all states

Call of Imperative Functions
Example 1

scheme COUNTER =
  class
    variable
      counter : **Nat** := 0
    value
      increase : **Unit** → **write** counter **Nat**
      increase() ≡ counter := counter + 1 ; counter
  end

scheme TEST_COUNTER =
  extend COUNTER **with**
  class
    value
      increase_and_test : **Nat** → **write** counter **Bool**
      increase_and_test(n) ≡ increase() ≤ n
  end

Call of Imperative Functions

Example 2

scheme INCREASE_TWICE =
  extend COUNTER **with**
  class
    value
      increase_twice : **Unit** → **write** counter **Nat**
      increase_twice() ≡ increase() ; increase()
  end

is wrong

scheme INCREASE_TWICE =
  extend COUNTER **with**
  class
    value
      increase_twice : **Unit** → **write** counter **Nat**
      increase_twice() ≡
        let dummy = increase() **in** increase() **end**
  end

is correct

Call of Imperative Functions

Example 3

scheme COUNTER =
  class
    variable
      counter : **Nat** := 0
    value
      increase : **Unit** → **write** counter **Unit**
      increase() ≡ counter := counter + 1,

      return_counter : **Unit** → **read** counter **Nat**
      return_counter() ≡ counter
  end

scheme INCREASE_TWICE =
  extend COUNTER **with**
  class
    value
      increase_twice : **Unit** → **write** counter **Nat**
      increase_twice() ≡
        increase() ; increase() ; return_counter()
  end

If Expressions

Revisited

Standard form:

if value_expr$_1$ **then** value_expr$_2$ **else** value_expr$_3$ **end**

Derived form:

if value_expr$_1$ **then** value_expr$_2$ **end**

short for:

if value_expr$_1$ **then** value_expr$_2$ **else skip end**

where **skip** ≡ ()

Example

variable counter : **Nat**
value
  decrease : **Unit** → **write** counter **Unit**
  decrease() ≡
    if counter > 0 **then** counter := counter − 1 **end**

While Expressions

General form:

while *logical*−value_expr$_1$ **do** *unit*−value_expr$_2$ **end**

has type **Unit**

while value_expr$_1$ **do** value_expr$_2$ **end**

equivalent to:

if value_expr$_1$ **then**
value_expr$_2$; **while** value_expr$_1$ **do** value_expr$_2$ **end**
else skip
end

Example

1 + 1/2 + ... + 1/n

scheme FRACTION_SUM =
class
variable
counter : **Nat**,
result : **Real**
value
fraction_sum : **Nat** $\xrightarrow{\sim}$ **write** counter, result **Unit**
fraction_sum(n) ≡
counter := n ;
result := 0.0 ;
while counter > 0 **do**
result := result + 1.0/(**real** counter) ;
counter := counter − 1
end
pre n > 0
end

Until Expressions

General form:

do *unit*−value_expr$_1$ **until** *logical*−value_expr$_2$ **end**

has type **Unit**

do value_expr$_1$ **until** value_expr$_2$ **end**

equivalent to:

value_expr$_1$; **while** ∼value_expr$_2$ **do** value_expr$_1$ **end**

Example

scheme FRACTION_SUM =
class
variable
counter : **Nat**,
result : **Real**
value
fraction_sum : **Nat** $\xrightarrow{\sim}$ **write** counter, result **Unit**
fraction_sum(n) ≡
counter := n ;
result := 0.0 ;
do
result := result + 1.0/(**real** counter) ;
counter := counter − 1
until counter = 0 **end**
pre n > 0
end

For Expressions

General form:

for
binding
in
readonly_list−value_expr$_1$ · *readonly_logical*−value_expr$_p$
do
unit−value_expr$_2$
end

has type **Unit**

for binding **in** value_expr$_1$ · value_expr$_p$
do
value_expr$_2$
end

equivalent to:

let
id = ⟨value_expr$_2$|binding **in** value_expr$_1$ · value_expr$_p$⟩
in skip end

Example

scheme FRACTION_SUM =
class
variable
result : **Real**
value
fraction_sum : **Nat** $\tilde{\to}$ **write** result **Unit**
fraction_sum(n) ≡
result := 0.0 ;
for i **in** ⟨1 .. n⟩ **do**
result := result + 1.0/(**real** i)
end
pre n > 0
end

Post Expressions
revisited

General form:

value_expr$_1$ **as** binding
post *readonly*−*logical*−value_expr$_2$
pre *readonly*−*logical*−value_expr$_3$

value_expr$_1$ **as** binding **post** value_expr$_2$ **pre** value_expr$_3$

is short for:

(value_expr$_3$ ≡ **true**) ⇒
value_expr$_1$ **as** binding **post** value_expr$_2$

value_expr$_1$ **as** binding **post** value_expr$_2$

- evaluates left-to-right
- has no side-effects
- returns **true** if
  - value_expr$_1$ is convergent
  - value_expr$_2$ is convergent and evaluates to **true**

  else **false**

Post Expressions, Example 1

scheme CHOOSE =
class
variable
set : **Int-set**
value
choose : **Unit** $\tilde{\to}$ **write** set **Int**
axiom
choose() **as** i **post** i ∈ set ∧ set = set\{i}
pre set ≠ {}
end

Avoiding pre-names:

axiom
let s = set **in**
choose() **as** i **post** i ∈ s ∧ set = s\{i}
pre set ≠ {}
end

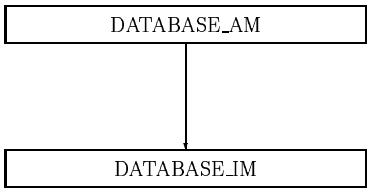
Post Expressions, Example 2

scheme INSERT_SORTED =
  class
    variable
      list : **Int*** := ⟨⟩

    value
      is_sorted : **Unit** → **read** list **Bool**
      is_sorted() ≡
        (∀ idx1,idx2 : **Nat** ·
          ({idx1,idx2} ⊆ **inds** list ∧ idx1 < idx2) ⇒
            list(idx1) < list(idx2)),

      insert : **Int** → **write** list **Unit**
      insert(i)
        post elems list = **elems** list ∪ {i} ∧ is_sorted()
  end

From Applicative to Imperative



Applicative (Model-oriented) Specification

scheme DATABASE_AM =
  class
    type
      Database = Key $\xrightarrow[m]{}$ Data,
      Key, Data

    value
      empty : Database
      empty ≡ [ ],

      insert : Key × Data × Database → Database
      insert(k,d,db) ≡ db † [k ↦ d],

      remove : Key × Database → Database
      remove(k,db) ≡ db \ {k},

      defined : Key × Database → **Bool**
      defined(k,db) ≡ k ∈ **dom** db,

      lookup : Key × Database $\xrightarrow{\sim}$ Data
      lookup(k,db) ≡ db(k)
      pre defined(k,db)
  end

Imperative (Model-oriented) Specification

scheme DATABASE_IM =
  class
    type
      Key, Data
    variable
      database : Key $\xrightarrow[m]{}$ Data
    value
      empty : **Unit** → **write** database **Unit**
      empty() ≡ database := [ ],

      insert : Key × Data → **write** database **Unit**
      insert(k,d) ≡ database := database † [k ↦ d],

      remove : Key → **write** database **Unit**
      remove(k) ≡ database := database \ {k},

      defined : Key → **read** database **Bool**
      defined(k) ≡ k ∈ **dom** database,

      lookup : Key $\xrightarrow{\sim}$ **read** database Data
      lookup(k) ≡ database(k) **pre** defined(k)
  end