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:

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

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** $\stackrel{\sim}{\rightarrow}$ **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, ... , 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

$1 + 1/2 + \ldots + 1/n$

```
scheme FRACTION_SUM =
  class
    variable
      counter : Nat,
      result : Real
    value
      fraction_sum : Nat -~-> 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 -~-> 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
$\quad$ *readonly_list*–value_expr$_1$ • *readonly_logical*–value_expr$_p$
do
$\quad$ *unit*–value_expr$_2$
end

has type **Unit**

for binding **in** value_expr$_1$ • value_expr$_p$
do
$\quad$ value_expr$_2$
end

equivalent to:

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

Example

scheme FRACTION_SUM =
$\quad$ **class**
$\quad\quad$ **variable**
$\quad\quad\quad$ result : **Real**
$\quad\quad$ **value**
$\quad\quad\quad$ fraction_sum : **Nat** $\stackrel{\sim}{\to}$ **write** result **Unit**
$\quad\quad\quad$ fraction_sum(n) ≡
$\quad\quad\quad\quad$ result := 0.0 ;
$\quad\quad\quad\quad$ **for** i **in** ⟨1 .. n⟩ **do**
$\quad\quad\quad\quad\quad$ result := result + 1.0/(**real** i)
$\quad\quad\quad\quad$ **end**
$\quad\quad\quad$ **pre** n > 0
$\quad$ **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 =
$\quad$ **class**
$\quad\quad$ **variable**
$\quad\quad\quad$ set : **Int-set**
$\quad\quad$ **value**
$\quad\quad\quad$ choose : **Unit** $\stackrel{\sim}{\to}$ **write** set **Int**
$\quad\quad$ **axiom**
$\quad\quad\quad$ choose() **as** i **post** $i \in set \wedge set = set \backslash \{i\}$
$\quad\quad\quad$ **pre** set ≠ {}
$\quad$ **end**

Avoiding pre-names:

axiom
$\quad$ **let** s = set **in**
$\quad\quad$ choose() **as** i **post** $i \in s \wedge set = s \backslash \{i\}$
$\quad\quad$ **pre** set ≠ {}
$\quad$ **end**

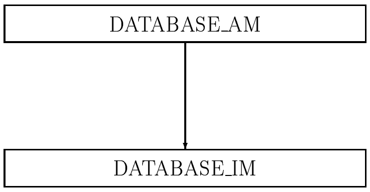
Post Expressions, Example 2

scheme INSERT_SORTED =
  class
    variable
      list : $\mathbf{Int}^*$:= $\langle\rangle$

    value
      is_sorted : **Unit** $\to$ **read** list **Bool**
      is_sorted() $\equiv$
        ($\forall$ idx1,idx2 : **Nat** $\cdot$
          ($\{$idx1,idx2$\}$ $\subseteq$ **inds** list $\wedge$ idx1 $<$ idx2) $\Rightarrow$
            list(idx1) $<$ list(idx2)),

      insert : **Int** $\to$ **write** list **Unit**
      insert(i)
        post elems list = **elems** list` $\cup$ $\{$i$\}$ $\wedge$ is_sorted()
  end

From Applicative to Imperative



Applicative (Model-oriented) Specification

scheme DATABASE_AM =
  class
    type
      Database = Key $\overrightarrow{m}$ Data,
      Key, Data

    value
      empty : Database
      empty $\equiv$ [ ],

      insert : Key $\times$ Data $\times$ Database $\to$ Database
      insert(k,d,db) $\equiv$ db $\dagger$ [k $\mapsto$ d],

      remove : Key $\times$ Database $\to$ Database
      remove(k,db) $\equiv$ db $\setminus$ $\{$k$\}$,

      defined : Key $\times$ Database $\to$ **Bool**
      defined(k,db) $\equiv$ k $\in$ **dom** db,

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

Imperative (Model-oriented) Specification

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

      insert : Key $\times$ Data $\to$ **write** database **Unit**
      insert(k,d) $\equiv$ database := database $\dagger$ [k $\mapsto$ d],

      remove : Key $\to$ **write** database **Unit**
      remove(k) $\equiv$ database := database $\setminus$ $\{$k$\}$,

      defined : Key $\to$ **read** database **Bool**
      defined(k) $\equiv$ k $\in$ **dom** database,

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