Formal Software Specification Using RAISE

Anne Elisabeth Haxthausen

Søren Prehn

Chris George

Copyright © **1996** by IT/DTU and UNU/IIST

Permission is hereby granted to copy for non-commercial purposes.

Introduction

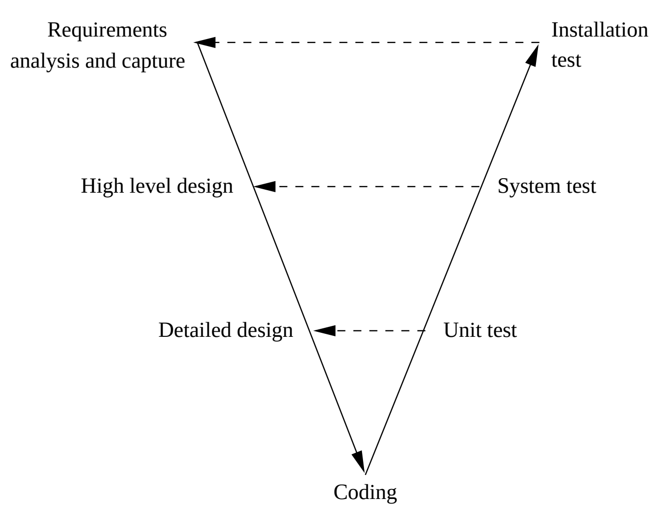
Course Aim

Introduction to formal specification

- languages
- techniques
- methods

Specific method: RAISE

V-diagram model of software life cycle



The V-diagram illustrates the typical re-work cycles when we discover errors by testing.

We aim to *find errors earlier*.

We concentrate on the early stages:

- *requirements analysis and capture*
- *high level design*

Specification

specification	implementation
(abstract)	(concrete)
(what)	(how)

Mathematical models describe real world

[specification] ———> [program]

Formal

Formal specification language:

- precise syntax
- mathematical meaning (semantics)

Formality =>

- unambiguous
- formal reasoning (prove properties)

Methods for Software Development

- Ad hoc
- Systematic
- Rigorous (R in RAISE: Rigorous)
- Formal

What is RAISE?

Rigorous **A**pproach to
Industrial **S**oftware **E**ngineering

RAISE is a product consisting of:

- a method for software development
- a formal specification language: RSL
- computer based tools

developed by:

- DDC/CRI (DK)
- STL/BNR (UK)
- ICL (UK)
- NBB/ABB/SYPRO (DK)

in an ESPRIT-I project, RAISE, 1985 - 1990

Background

model-oriented (VDM, Z, ...)

property-oriented (Clear, ...)

concurrency (CSP, ...)

structuring (ML, ...)

tools

RAISE

RAISE Continuation

ESPRIT-II project, **LaCoS**, 1990 - 1995

Large **S**cale **C**orrect **S**ystems
Using Formal Methods

- industrial applications of RAISE
- evolution of
  RAISE method, language and tools

LaCoS Partners

Producers:

CRI (DK)
SYPRO (DK)
BNR Europe (UK)

Consumers:

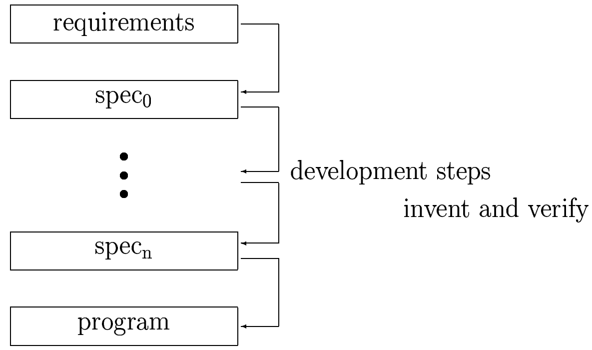
BNR Europe (UK):	Network design toolset
Lloyd's Register (UK):	Ship engine monitoring; security
Bull (F):	Database; security
MATRA Transport (F):	Automatic train protection
Inisel Espacio (E):	Image processing
Space Software Italia (I):	Tethered satellite; air traffic control
Technisystems (GR):	Shipping transaction processing

RAISE Specification Language, RSL

Features:

- Formal
- Wide spectrum
  - model-oriented and property-oriented
  - applicative and imperative
  - sequential and concurrent
- Structuring facilities

RAISE Method



specification is formal (formulated in RSL)

verification may be formal

RAISE Tools

- Storage of
  - specifications
  - development relations
  - “proof” obligations
  - “proofs”
  - etc
- Editors (with syntax and type check)
- Translation (to Ada and C++)
- Pretty printing facilities (LaTeX)
- “Proof” tools

Advantages of Using RAISE

- abstraction
- formal reasoning
- reuse
- tools support

=>

- fewer errors

Example: RSL Specification (Model-oriented)

scheme SET_DATABASE =
  class
    type
      Database = Person-**set**,
      Person = **Text**

    value
      empty : Database = $\{\}$,

      register : Person × Database → Database
      register(p,db) ≡ db ∪ {p},

      check : Person × Database → **Bool**
      check(p,db) ≡ p ∈ db
  end

Some immediate questions

- Do we need any more functions?
- Are the definitions correct?
- Are they what we need?
- Could we use this for registering the people in this class?
- Could we use this for registering the people in this country?
- Is **Text** a good model for Person?

RSL Specification

An RSL specification consists of

- module definitions

A module contains definitions of

- types
- values
- variables
- channels
- modules
- axioms

Abstraction

“What” rather than “how”

RSL allows

- data abstraction
- operation abstraction

Data abstraction: examples

type Database = Person-**set**

type Database = Person*

type Database

type Person = **Text**

type Person = **Nat**

type Person

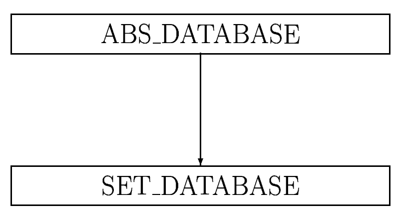
Operation abstraction: example

value
 square_root : **Real** $\overset{\sim}{\to}$ **Real**
 square_root(r) **as** s
  post $s * s = r \wedge s \geq 0.0$
  pre $r \geq 0.0$

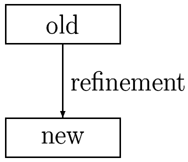
Example: RSL Specification (Property-oriented)

scheme ABS_DATABASE =
 class
  type
   Database,
   Person
  value
   empty : Database,
   register : Person × Database → Database,
   check : Person × Database → **Bool**
  axiom
   ∀ p : Person ·
    check(p, empty) ≡ **false**,
   ∀ p,p′ : Person, db : Database ·
    check(p′, register(p, db)) ≡ p′ = p ∨ check(p′, db)
 end

A Development Step



Refinement Relation



1. new signature includes the old
   (statically decidable)
2. old properties hold in the new
   (⇒ proof obligations: "refinement conditions")

Refinement Conditions

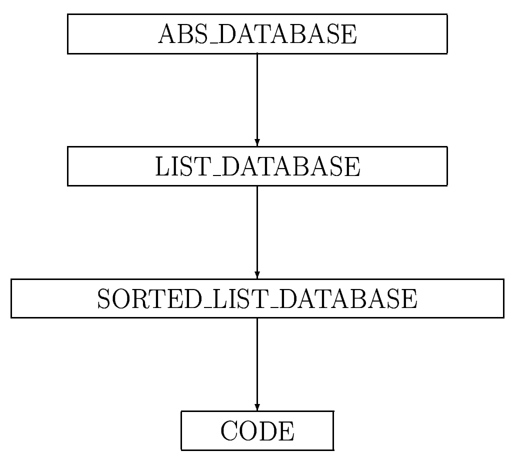
$\llcorner \forall$ p : Person · check(p, empty) $\equiv$ **false**$\lrcorner$

$\llcorner \forall$ p,p$'$: Person, db : Database ·
  check(p$'$, register(p, db)) $\equiv$ p$'$ = p $\vee$ check(p$'$, db)$\lrcorner$

Verification

$\llcorner$check(p$'$, register(p, db)) $\equiv$ p$'$ = p $\vee$ check(p$'$, db)$\lrcorner$
unfold register:
$\llcorner$check(p$'$, db $\cup$ {p}) $\equiv$ p$'$ = p $\vee$ check(p$'$, db)$\lrcorner$
unfold check:
$\llcorner$p$'$ $\in$ (db $\cup$ {p}) $\equiv$ p$'$ = p $\vee$ check(p$'$, db)$\lrcorner$
unfold check:
$\llcorner$p$'$ $\in$ (db $\cup$ {p}) $\equiv$ p$'$ = p $\vee$ p$'$ $\in$ db$\lrcorner$
isin_union:
$\llcorner$p$'$ $\in$ db $\vee$ p$'$ $\in$ {p} $\equiv$ p$'$ = p $\vee$ p$'$ $\in$ db$\lrcorner$
isin_singleton:
$\llcorner$p$'$ $\in$ db $\vee$ p$'$ = p $\equiv$ p$'$ = p $\vee$ p$'$ $\in$ db$\lrcorner$
or_commutativity:
$\llcorner$p$'$ = p $\vee$ p$'$ $\in$ db $\equiv$ p$'$ = p $\vee$ p$'$ $\in$ db$\lrcorner$
is_annihilation:
$\llcorner$**true**$\lrcorner$
qed

Alternative Development



Summary

RAISE

- Method
  - stepwise development
  - invent and verify
  - rigorous
- Specification language (RSL)
  - formal
  - wide spectrum
  - structuring facilities
- Tools

RSL has a range of styles:

- applicative
- imperative
- concurrent

Most of the course will use the applicative style, which is close to mathematics and to functional programming.

The imperative style is closer to traditional programming, with program variables, assignments, sequencing and loops.

The concurrent style supports the specification of concurrent features of software.

Syntax Overview

Specifications

An RSL specification consists of

- module definitions

A module contains definitions of

- types
- values
- variables
- channels
- modules
- axioms

No special order of definitions is required.

Module Definitions

id =
  class
    declaration$_1$
    ⋮
    declaration$_n$
  end

Declarations

A declaration is a list of definitions of the same kind:

type
value
axiom
variable
channel
scheme
object

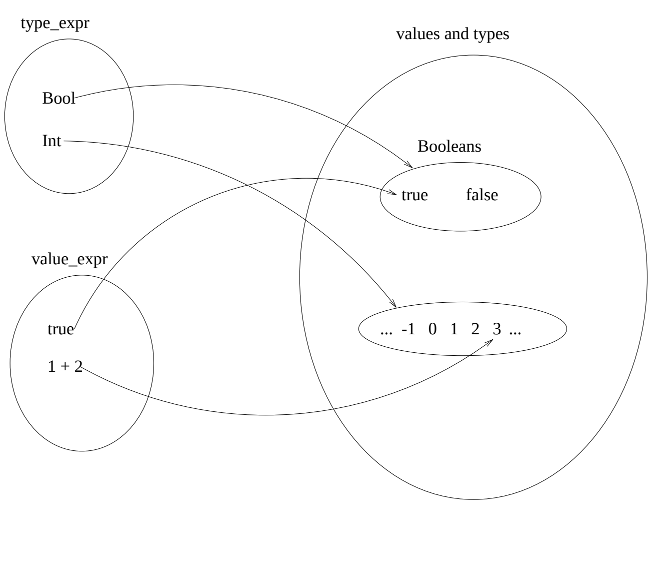
e.g.

type
  type_definition$_1$,
  ⋮
  type_definition$_n$

value
  value_definition$_1$,
  ⋮
  value_definition$_n$

Types

Types are collections of values.



Type Expressions

1. literals (**Int**, ...)
2. type operator applications (**Int-set**, ...)
3. subtype expressions ({| i : **Int** • i ≥ 0 |}, ...)
4. identifiers defined in type definitions (Person)

All have associated =, ≠.

1 and 2:
  have additional associated built-in operators (+, ∪,...).

3 and 4:
  may have additional associated built-in operators.

Type Definitions

- abbreviation

  type id = type_expr

  type Database = Person-**set**

- sort

  type id

  type Database

Literals

Bool

values: **true**, **false**

connectives: $\wedge$, $\vee$, $\Rightarrow$, $\sim$

Int

values: ..., -2, -1, 0, 1, 2, ...

operators: $+$, $-$, $*$, $/$, $\uparrow$, $\backslash$, $<$, $\leq$, $>$, $\geq$, **abs**, **real**

Nat = {| i : **Int** · i ≥ 0 |}

Real

values: ..., -4.3, ..., 0.0, ..., 1.0, ...

operators: $+$, $-$, $*$, $/$, $\uparrow$, $<$, $\leq$, $>$, $\geq$, **abs**, **int**

Char

values: 'a', ...

Text = **Char***

values: "Alice", ...

Unit

value: ()

Composite Types

- products ($\times$)
- functions ($\rightarrow$, $\stackrel{\sim}{\rightarrow}$)
- sets (**-set**, **-infset**)
- lists (*, $^\omega$)
- maps ($\overrightarrow{m}$, $\stackrel{\sim}{\overrightarrow{m}}$)

Value Definitions

- different forms
  - typing (+ axiom(s))
  - explicit value definition
  - implicit value definition
  - explicit function definition
  - implicit function definition
- 4 last forms may be transformed to first form

Constant Value Definitions

- explicit value definition

  value $x : \mathbf{Int} = 1$

- implicit value definition

  value $x : \mathbf{Int} \cdot x > 0$

- typing (+ axiom(s))

  value $x : \mathbf{Int}$
  axiom $x > 0$

Function Value Definitions

- explicit function definition

  value
  $f : \mathbf{Int} \to \mathbf{Int}$
  $f(x) \equiv x + 1$

- implicit function definition

  value
  $f : \mathbf{Int} \to \mathbf{Int}$
  $f(x)$ **as** r **post** $r > x$

- typing (+ axiom(s))

  value $f : \mathbf{Int} \to \mathbf{Int}$
  axiom $\forall\, x : \mathbf{Int} \cdot f(x) > x$

Value expressions

Constructed from

- literals (1, **true**, ...)
- operators (+, ...)
- connectives ($\wedge$, ...)
- identifiers introduced in value definitions (empty, check, ...)
- function application (f(x))
- if expressions (**if** ...)
- quantified expressions ($\forall$, ...)
- equivalence expressions
- ...

Summary

- type definitions: abbreviation and sort
- type expressions
- value definitions: explicit, implicit and axiomatic
- value expressions

Logic

Computing involves *partial* functions

- division
- head of a list
- loops

So we need a logic that can deal with expressions that may not be well defined.

By *well defined* we mean has (or evaluates to) a value.

Expressions and values

An expression may or may not evaluate to a value:

Expression	Value
true	**true**
$1 + 0$	1
$1 / 0$	?
factorial(3)	6
factorial(-1)	?
factorial(x)	?
if $x > 0$ **then** factorial(x) **else** 0 **end**	$\surd$
while true do skip end	$\times$

chaos

Used to represent undefinedness

while true do skip end $\equiv$ **chaos**

$/ :$ **Real** $\times$ **Real** $\stackrel{\sim}{\to}$ **Real**

1.0/0.0 is under-specified

1.0/0.0 might evaluate to **chaos**

f : **Real**$\to$**Real**
f(x) $\equiv$ **if** $x \neq 0.0$ **then** 1.0/x **else** 0.0 **end**

If expressions

Example:

if x > 0 **then** factorial(x) **else** 0 **end**

More general form:

if *logical*-expr **then** expr1 **else** expr2 **end**

Properties:

if true then expr1 **else** expr2 **end** $\equiv$ expr1
if false then expr1 **else** expr2 **end** $\equiv$ expr2
if chaos then expr1 **else** expr2 **end** $\equiv$ **chaos**

Non-strictness:

if true then expr1 **else chaos end** $\equiv$ expr1
if false then chaos else expr2 **end** $\equiv$ expr2

Connectives

Properties:

$\sim$e $\equiv$ **if** e **then false else true end**

e1 $\wedge$ e2 $\equiv$ **if** e1 **then** e2 **else false end**

e1 $\vee$ e2 $\equiv$ **if** e1 **then true else** e2 **end**

e1 $\Rightarrow$ e2 $\equiv$ **if** e1 **then** e2 **else true end**

gives conditional logic

Truth tables

$\wedge$	true	false	chaos
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

$\vee$	true	false	chaos
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

$\Rightarrow$	true	false	chaos
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

Note:

e1 $\wedge$ e2 $\equiv$ e2 $\wedge$ e1
e1 $\vee$ e2 $\equiv$ e2 $\vee$ e1

are not tautologies

p : **Real** $\rightarrow$ **Bool**
p(x) $\equiv$ (x $\neq$ 0.0) $\wedge$ (1.0/x $\leq$ epsilon)

p(0.0)
$\equiv$ (0.0 $\neq$ 0.0) $\wedge$ (1.0/0.0 $\leq$ epsilon)
$\equiv$ **if** (0.0 $\neq$ 0.0) **then** (1.0/0.0 $\leq$ epsilon) **else false end**
$\equiv$ **if false then** (1.0/0.0 $\leq$ epsilon) **else false end**
$\equiv$ **false**

Quantified expressions

Examples:

$\forall$ x : **Nat** · (x = 0) $\vee$ (x > 0)

$\exists$ x : **Int** · x = 7

$\exists!$ x : **Int** · (x $\geq$ 0) $\wedge$ (x $\leq$ 0)

$\forall$ x : **Nat** · x = -7

$\forall$ x, y : **Nat** · ($\exists!$ z : **Nat** · x+y = z)

General form:

quantifier typing$_1$, ... , typing$_n$ · *logical*-expr

All quantification is over values in the types stated, i.e. not over **chaos**.

Products

Products

A product is:

an ordered finite collection of values of possibly different types

Examples:

(1,2)
(1,**true**,"John")

Product Type Expressions

type_expr$_1$ × ... × type_expr$_n$, n $\geq$ 2

Values:

(v$_1$,...,v$_n$), v$_i$: type_expr$_i$

Operators:

=
$\neq$

Examples

Bool × **Bool**

(**true**,**true**)
(**true**,**false**)
(**false**,**true**)
(**false**,**false**)

Nat × **Nat** × **Bool**

(0,0,**true**)
(0,0,**false**)
(0,1,**true**)
(0,1,**false**)
(1,0,**true**)
(1,0,**false**)
(2,0,**true**)
⋮

Example: A System of Coordinates I

SYSTEM_OF_COORDINATES =
  class
    type
      Position = **Real** × **Real**
    value
      origin : Position = (0.0,0.0),
      distance : Position × Position → **Real**
      distance((x_1,y_1),(x_2,y_2)) ≡
        ((x_2−x_1)↑2.0 + (y_2−y_1)↑2.0)↑0.5
  end

Example: A System of Coordinates II

SYSTEM_OF_COORDINATES =
  class
    type
      Position = **Real** × **Real**
    value
      origin : Position = (0.0,0.0),
      distance : Position × Position → **Real**
      distance(p1,p2) ≡
        let
          (x_1,y_1) = p1,
          (x_2,y_2) = p2
        in
          ((x_2−x_1)↑2.0 + (y_2−y_1)↑2.0)↑0.5
        end
  end

(Explicit) Let Expressions

let
  binding_1 = expr_1,
    ⋮
  binding_n = expr_n
in
  expr
end

Bindings

Examples:

x
(x,y)
(x,y,z)
((x1,y1),(x2,y2))
(x,(y,z))

Forms:

id
($binding_1$, ... , $binding_n$)

Use:

- typings
- let expressions
- formal function parameters
- ...

Typings

Examples:

x : **Real**
(x,y) : Position
a, b : **Real**

Basic Form:

binding : type_expr

Context condition:

binding must match type_expr

Derived form:

$binding_1$, ... , $binding_n$: type_expr

Use:

- value definitions
- quantified expressions
- ...

Functions

Total and Partial Functions

- total functions

  $type\_expr_1 \rightarrow type\_expr_2$

- partial functions

  $type\_expr_1 \xrightarrow{\sim} type\_expr_2$

$\forall\, f_{tot} : T_1 \rightarrow T_2,\ f_{par} : T_1 \xrightarrow{\sim} T_2,\ x : T_1 \cdot$

	defined (not **chaos**)	deterministic
$f_{tot}(x)$	yes	yes
$f_{par}(x)$	might be	might be

$\exists !\ y : T_2 \cdot f_{tot}(x) \equiv y$

Preconditions

Partial functions are usually defined with preconditions.

For example:

factorial : **Int** $\stackrel{\sim}{\to}$ **Int**
factorial(x) ≡
  if x = 1 **then** 1 **else** x ∗ factorial(x − 1) **end**
pre x > 0

The defining expression for the function can only be used when the precondition is true. So we can deduce that

f(3) = 3 ∗ f(2) = ... = 3 ∗ 2 ∗ 1 = 6

but we can deduce nothing about f(0), say.

Function Value Expressions

Basic form:

λ binding : type_expr · value_expr

Examples:

λ b : **Bool** · ∼b
λ (x,y) : **Int** × **Int** · x + y
λ (b,(x,y)) : **Bool** × (**Nat** × **Nat**) ·
  if b **then** x **else** y **end**

Semantics:

represents function of type: type_expr $\stackrel{\sim}{\to}$ T,
where T = type_of(value_expr)

Function Application Expressions

Examples:

fraction(0.5)
abs(−7)

Typical form:

function-expr($\mathrm{expr}_1, \ldots, \mathrm{expr}_n$), n ≥ 0

Context conditions:

($\mathrm{expr}_1, \ldots, \mathrm{expr}_n$)

must be of the argument type of expr

Associated Built-in Operators

= , ≠, ∘

$\circ : (T_2 \stackrel{\sim}{\to} T_3) \times (T_1 \stackrel{\sim}{\to} T_2) \to (T_1 \stackrel{\sim}{\to} T_3)$
(f ∘ g)(x) ≡ f(g(x))

Function Value Definitions

- explicit function definition

  value
  f : **Int** $\to$ **Int**
  f(x) $\equiv$ x + 1

- implicit function definition

  value
  f : **Int** $\to$ **Int**
  f(x) **as** r **post** r > x

- typing (+ axiom(s))

  value f : **Int** $\to$ **Int**
  axiom $\forall$ x : **Int** $\cdot$ f(x) > x

Explicit Function Definition

value
fraction : **Real** $\to$ **Real**
fraction(x) $\equiv$ **if** x = 0.0 **then** 0.0 **else** 1.0/x **end**

short for

value
fraction : **Real** $\to$ **Real**
axiom
$\forall$ x : **Real** $\cdot$
fraction(x) $\equiv$ **if** x = 0.0 **then** 0.0 **else** 1.0/x **end**

Explicit Function Definition

value
partial_fraction : **Real** $\stackrel{\sim}{\to}$ **Real**
partial_fraction(x) $\equiv$ 1.0/x
pre x $\neq$ 0.0

short for

value
partial_fraction : **Real** $\stackrel{\sim}{\to}$ **Real**
axiom
$\forall$ x : **Real** $\cdot$
partial_fraction(x) $\equiv$ 1.0/x
pre x $\neq$ 0.0

Implicit Function Definition

value
square_root : **Real** $\stackrel{\sim}{\to}$ **Real**
square_root(x) **as** s
post s $*$ s = x $\wedge$ s $\geq$ 0.0
pre x $\geq$ 0.0

short for:

value
square_root : **Real** $\stackrel{\sim}{\to}$ **Real**
axiom
$\forall$ x : **Real** $\cdot$
square_root(x) **as** s
post s $*$ s = x $\wedge$ s $\geq$ 0.0
pre x $\geq$ 0.0

Algebraic Specification Example

DATABASE =
  class
    type
      Database, Key, Data
    value
      /* generators */
      empty : Database,
      insert : Key × Data × Database → Database,
      remove : Key × Database → Database,
      /* observers */
      defined : Key × Database → **Bool**,
      lookup : Key × Database $\stackrel{\sim}{\rightarrow}$ Data

DATABASE contd.

  axiom
    [defined_empty]
    ∀ k : Key ·
      defined(k,empty) ≡ **false**,
    [defined_insert]
    ∀ k,k1 : Key, d : Data, db : Database ·
      defined(k,insert(k1,d,db)) ≡
        k = k1 ∨ defined(k,db),
    [defined_remove]
    ∀ k,k1 : Key, db : Database ·
      defined(k,remove(k1,db)) ≡
        k ≠ k1 ∧ defined(k,db),
    [lookup_insert]
    ∀ k,k1 : Key, d : Data, db : Database ·
      lookup(k,insert(k1,d,db)) ≡
        if k = k1 **then** d **else** lookup(k,db) **end**
      pre k = k1 ∨ defined(k,db),
    [lookup_remove]
    ∀ k,k1 : Key, db : Database ·
      lookup(k,remove(k1,db)) ≡ lookup(k,db)
      pre k ≠ k1 ∧ defined(k,db)
  end

Function Definition

1. Decide name (f)
2. Decide type
   (a) Argument type T_a
   (b) Result type T_r
   (c) Total (→) or partial ($\stackrel{\sim}{\rightarrow}$):
      i. total: if can define for all values in parameters
      ii. partial: if a pre condition is necessary
3. Decide definition style:
   (a) explicit:
      if it is possible to state a formula
      f(x) ≡ expr[x]
   (b) implicit:
      if it is possible to write an input-output relation
      p[x,r]
   (c) axiomatic:
      - always possible
      - typically used in connection with sorts
      - necessary, if special argument forms are wanted
        e.g. f(g(x),x) ≡ h(f,x)

Sets

Contents

Sets

- finite and infinite sets
- set type expressions
- set operators
- set value expressions
- examples of abstraction using sets

Sets

A set is:

an unordered collection
of
values of same type

Examples:

{1,3,5}
{"John","Peter","Ann"}

Set Type Expressions

- type_expr-**set**

  $\{v_1, \ldots, v_n\}$

  where $n \geq 0$, v_i : type_expr

- type_expr-**infset**

  $\{v_1, \ldots, v_n\}$,
  $\{v_1, \ldots, v_n, \ldots\}$

  where $n \geq 0$, v_i : type_expr

Examples

Bool-set

{}
{**true**}
{**false**}
{**true**,**false**}

Nat-set

{}
{0}
{1}
...
{0,1}
...
{1,2,3}
...

Associated Built-in Operators

$\cup$: T-**infset** $\times$ T-**infset** $\to$ T-**infset**

$\{1, 2, 3\} \cup \{3, 4\} = \{1, 2, 3, 4\}$

$\cap$: T-**infset** $\times$ T-**infset** $\to$ T-**infset**

$\{1, 2, 3\} \cap \{3, 4\} = \{3\}$

$\backslash$: T-**infset** $\times$ T-**infset** $\to$ T-**infset**

$\{1, 2, 3\} \backslash \{3, 4\} = \{1, 2\}$

$\in$: T $\times$ T-**infset** $\to$ **Bool**

$4 \in \{1, 2, 3\}$ = **false**

$\notin$: T $\times$ T-**infset** $\to$ **Bool**

$4 \notin \{1, 2, 3\}$ = **true**

$\subset$: T-**infset** $\times$ T-**infset** $\to$ **Bool**

$\{1, 3\} \subset \{1, 2, 3\}$ = **true**
$\{1, 2, 3\} \subset \{1, 2, 3\}$ = **false**

$\subseteq$: T-**infset** $\times$ T-**infset** $\to$ **Bool**

$\{1, 3\} \subseteq \{1, 2, 3\}$ = **true**
$\{1, 2, 3\} \subseteq \{1, 2, 3\}$ = **true**
$\{1, 2, 3\} \subseteq \{1, 3\}$ = **false**

$\supset$ and $\supseteq$ are similar

card : T-**infset** $\stackrel{\sim}{\to}$ **Nat**

card $\{1, 2, 5, 2, 2, 1, 5\} = 3$
card $\{$n | n : **Nat**$\}$ $\equiv$ **chaos**

Set Value Expressions

Enumerated:

$\{1,2\}$
$\{1,2,1\}$

$\{expr_1,...,expr_n\}$

Ranged:

$\{3 .. 7\} = \{3,4,5,6,7\}$
$\{3 .. 3\} = \{3\}$
$\{3 .. 2\} = \{\}$

$\{$*integer*-$expr_1$ .. *integer*-$expr_2\}$

Comprehended:

$\{2*n \mid n : \textbf{Nat} \cdot n \leq 3\}$

$\{expr_1 \mid typing_1,...,typing_n \cdot$ *logical*-$expr_2\}$

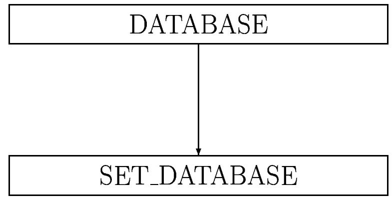
RESOURCE_MANAGER =
  class
    type
      Resource,
      Pool = Resource-**set**

    value
      obtain : Pool $\stackrel{\sim}{\to}$ Pool $\times$ Resource
      obtain(p) **as** (p_1,r_1) **post** $r_1 \in p \wedge p_1 = p \backslash \{r_1\}$
      pre $p \neq \{\}$,

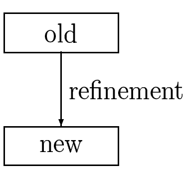
      release : Resource $\times$ Pool $\stackrel{\sim}{\to}$ Pool
      release(r,p) $\equiv p \cup \{r\}$
      pre $r \notin p$
  end

```
SET_DATABASE =
 class
  type
   Record = Key × Data,
   Database = {| rs : Record-set · is_wf_Database(rs) |},
   Key, Data
  value
   is_wf_Database : Record-set → Bool
   is_wf_Database(rs) ≡
    (∀ k : Key, d1,d2 : Data ·
     ((k,d1) ∈ rs ∧ (k,d2) ∈ rs) ⇒ d1 = d2),

   empty : Database = {},

   insert : Key × Data × Database → Database
   insert(k,d,db) ≡ remove(k,db) ∪ {(k,d)},

   remove : Key × Database → Database
   remove(k,db) ≡ db \ {(k,d) | d : Data · true},

   defined : Key × Database → Bool
   defined(k,db) ≡ (∃ d : Data · (k,d) ∈ db),

   lookup : Key × Database ~→ Data
   lookup(k,db) as d post (k,d) ∈ db
   pre defined(k,db)
 end
```


A Development Step



Refinement Relation



1. new signature includes the old (statically decidable)
2. old properties hold in the new (⇒ "refinement conditions")

Refinement Conditions

```
⌞∀ k : Key · defined(k,empty) ≡ false⌟

⌞∀ k,k1 : Key, d : Data, db : Database ·
  defined(k,insert(k1,d,db)) ≡
   k = k1 ∨ defined(k,db)⌟

⌞∀ k,k1 : Key, db : Database ·
  defined(k,remove(k1,db)) ≡
   k ≠ k1 ∧ defined(k,db)⌟

⌞∀ k,k1 : Key, d : Data, db : Database ·
  lookup(k,insert(k1,d,db)) ≡
   if k = k1 then d else lookup(k,db) end
  pre k = k1 ∨ defined(k,db)⌟

⌞∀ k,k1 : Key, db : Database ·
  lookup(k,remove(k1,db)) ≡ lookup(k,db)
  pre k ≠ k1 ∧ defined(k,db)⌟
```

Verification of a Refinement Condition

⌞∀ k : Key · defined(k,empty) ≡ **false**⌟
all_assumption_inf:
⌞defined(k,empty) ≡ **false**⌟
unfold empty:
⌞defined(k,{}) ≡ **false**⌟
unfold defined:
⌞(∃ d : Data · (k,d) ∈ {}) ≡ **false**⌟
isin_empty:
⌞(∃ d : Data · **false**) ≡ **false**⌟
exists_introduction:
⌞**false** ≡ **false**⌟
is_annihilation:
⌞**true**⌟
qed

Lists

Contents

Lists

- finite and infinite lists
- list type expressions
- list value expressions
- list indexing
- list operators
- examples of abstraction using lists

Lists

A list is:

an ordered collection
of
values of same type

Examples:

⟨1,3,3,1,5⟩
⟨**true**,**false**,**true**⟩

List Type Expressions

- type_expr*

  ⟨v_1, ..., v_n⟩

  where $n \geq 0$, v_i : type_expr

- type_expr$^\omega$

  ⟨v_1, ..., v_n⟩,
  ⟨v_1, ..., v_n, ...⟩

  where $n \geq 0$, v_i : type_expr

Examples

Bool*

⟨⟩
⟨**true**⟩
⟨**false**⟩
⟨**true**,**false**⟩
⟨**false**,**true**⟩
⟨**true**,**true**⟩
⟨**false**,**false**⟩
⟨**true**,**false**,**true**⟩
⋮

Bool$^\omega$

⟨⟩
⟨**true**⟩
⟨**false**⟩
⟨**true**,**false**⟩
⟨**false**,**true**⟩
⟨**true**,**true**⟩
⟨**false**,**false**⟩
⟨**true**,**false**,**true**⟩
⋮
⟨**false**,**true**,**true**,**true**,**false**,... ⟩
⋮

List Value Expressions

Enumerated:

⟨1,3,3,1,5⟩
⟨**true**,**false**,**true**⟩

⟨$expr_1$,...,$expr_n$⟩

Ranged:

⟨3 .. 7⟩ = ⟨3,4,5,6,7⟩
⟨3 .. 3⟩ = ⟨3⟩
⟨3 .. 2⟩ = ⟨⟩

⟨*integer*-$expr_1$ .. *integer*-$expr_2$⟩

Comprehended:

⟨2∗n | n **in** ⟨0 .. 3⟩⟩
⟨n | n **in** ⟨0 .. 100⟩ · is_even(n)⟩

⟨$expr_1$ | binding **in** *list*-$expr_2$ · *logical*-$expr_3$⟩

List Indexing

Basic form:

list-expr(*integer*-$expr_1$)

Example:

⟨2,5,3⟩(2) = 5

Derived form:

list-expr(*integer*-$expr_1$)...(*integer*-$expr_n$)

Example:

⟨⟨2,5,3⟩, ⟨3⟩⟩(1) = ⟨2,5,3⟩
⟨⟨2,5,3⟩, ⟨3⟩⟩(1)(2) = ⟨2,5,3⟩(2) = 5

Associated Built-in Operators

```
⌢ : T* × T^ω → T^ω

⟨e1,...,en⟩ ⌢ ⟨en+1,... ⟩ = ⟨e1,...,en,en+1,... ⟩

hd : T^ω ~→ T

hd ⟨e1,e2,... ⟩ = e1

tl : T^ω ~→ T^ω

tl ⟨e1,e2,... ⟩ = ⟨e2,... ⟩

len : T^ω ~→ Nat

len ⟨e1,...,en⟩ = n
len il ≡ chaos

elems : T^ω → T-infset

elems ⟨e1,e2,... ⟩ = {e1,e2,... }

inds : T^ω → Nat-infset

inds fl = {1 .. len fl}
inds il = {idx | idx : Nat • idx ≥ 1}
```

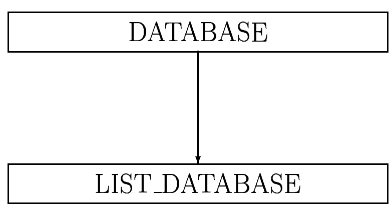


```
QUEUE =
  class
    type
      Element,
      Queue = Element*
    value
      empty : Queue = ⟨⟩,

      enq : Element × Queue → Queue
      enq(e,q) ≡ q ⌢ ⟨e⟩,

      deq : Queue ~→ Queue × Element
      deq(q) ≡ (tl q,hd q)
      pre q ≠ empty
  end
```



```
SORTING =
  class
    value
      sort : Int* → Int*
      sort(l) as l1 post is_permutation(l1,l) ∧ is_sorted(l1)

      is_permutation : Int* × Int* → Bool,
      is_permutation(l1,l2) ≡
        (∀ i : Int • count(i, l1) = count(i, l2)),

      count : Int × Int* → Nat
      count(i, l) ≡
        card {idx | idx : Nat •
                idx ∈ inds l ∧ l(idx) = i},

      is_sorted : Int* → Bool
      is_sorted(l) ≡
        (∀ idx1,idx2 : Nat •
          {idx1,idx2} ⊆ inds l ∧ idx1 < idx2 ⇒
            l(idx1) ≤ l(idx2))
  end
```


A Development Step



Database Representations

type Database

empty
insert(k_1, d_1, insert(k_2,d_2,empty))

type Database = (Key × Data)**-set**

{}
{(k_1,d_1), (k_2,d_2)}

type Database = (Key × Data)*

⟨⟩
⟨(k_1,d_1), (k_2,d_2)⟩


```
LIST_DATABASE =
  class
    type
      Key, Data,
      Record = Key × Data,
      Database = Record*
    value
      empty : Database = ⟨⟩,

      insert : Key × Data × Database → Database
      insert(k,d,db) ≡ ⟨(k,d)⟩ ⁀ db,

      remove : Key × Database → Database
      remove(k,db) ≡
        case db of
          ⟨⟩ → ⟨⟩,
          ⟨(k1,d1)⟩ ⁀ db1 →
            if k = k1 then remove(k,db1)
            else ⟨(k1,d1)⟩ ⁀ remove(k,db1) end
        end,
```



```
      defined : Key × Database → Bool
      defined(k,db) ≡
        case db of
          ⟨⟩ → false,
          ⟨(k1,d1)⟩ ⁀ db1 → k = k1 ∨ defined(k,db1)
        end,

      lookup : Key × Database ~→ Data
      lookup(k, db) ≡
        let (k1,d1) = hd db in
          if k = k1 then d1
          else lookup(k, tl db) end
        end
      pre defined(k, db)
  end
```



```
  ⌞∀ k : Key • remove(k,empty) ≡ empty⌟
all_assumption_inf:
  ⌞remove(k,empty) ≡ empty⌟
unfold empty, unfold empty:
  ⌞remove(k,⟨⟩) ≡ ⟨⟩⌟
unfold remove:
  ⌞case ⟨⟩ of ⟨⟩ → ⟨⟩, ... end ≡ ⟨⟩⌟
case_expansion:
  ⌞⟨⟩ ≡ ⟨⟩⌟
is_annihilation:
  ⌞true⌟
qed
```

Maps

Contents

Maps

- map type expressions
- map value expressions
- map application
- map operators
- examples of abstraction using maps

Maps

A map is:

an unordered collection of pairs of values

Examples:

$[''\texttt{Klaus}'' \mapsto 7, ''\texttt{John}'' \mapsto 2, ''\texttt{Mary}'' \mapsto 7]$
$[1 \mapsto 2, 5 \mapsto 10]$

Maps may be:

- infinite
- partial
- non-deterministic

Map Type Expressions

- type_expr$_1$ $\underset{m}{\rightarrow}$ type_expr$_2$

  $[v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$

  where $n \geq 0$, v_i : type_expr$_1$, w_i : type_expr$_2$
  and $v_i = v_j \Rightarrow w_i = w_j$

  Finite and deterministic when applied to elements in the domain

- type_expr$_1$ $\underset{m}{\tilde{\rightarrow}}$type_expr$_2$

  $[v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$,
  $[v_1 \mapsto w_1, \ldots, v_n \mapsto w_n, \ldots]$,

  where $n \geq 0$, v_i : type_expr$_1$, w_i : type_expr$_2$

  May be infinite and may be non-deterministic when applied to elements in the domain

NB The original RSL book only has $\underset{m}{\rightarrow}$, but with the meaning of $\underset{m}{\tilde{\rightarrow}}$ Finite maps were introduced and the symbols changed in the method book.

Example

Nat $\underset{m}{\rightarrow}$ **Bool**

$[\,]$
$[0 \mapsto$ **true**$]$
$[0 \mapsto$ **true**, $1 \mapsto$ **true**$]$
$[0 \mapsto$ **true**, $1 \mapsto$ **false**$]$
$\vdots$

Nat $\underset{m}{\tilde{\rightarrow}}$**Bool**

$[\,]$
$[0 \mapsto$ **true**$]$
$[0 \mapsto$ **true**, $1 \mapsto$ **true**$]$
$[0 \mapsto$ **true**, $1 \mapsto$ **false**$]$
$[0 \mapsto$ **true**, $0 +$ **false**$]$
$[0 \mapsto$ **true**, $0 +$ **false**, $1 \mapsto$ **true**$]$
$\vdots$

Map Value Expressions

Enumerated:

$[3 \mapsto$ **true**, $5 \mapsto$ **false**$]$
$[$`"Klaus"` $\mapsto 7$, `"John"` $\mapsto 2$, `"Mary"` $\mapsto 7]$

$[\text{expr}_1 \mapsto \text{expr}_1{}', \ldots, \text{expr}_n \mapsto \text{expr}_n{}']$

Comprehended:

$[n \mapsto 2{*}n \mid n : \textbf{Nat} \cdot n \leq 2] = [0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 4]$
$[n \mapsto 2{*}n \mid n : \textbf{Nat}] = [0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 4, \ldots]$

$[\text{expr}_1 \mapsto \text{expr}_2 \mid \text{typing}_1,\ldots,\text{typing}_n \cdot \textit{logical}\text{-expr}_3]$

Map Application

Basic form:

map-expr(expr$_1$)

Examples:

$[$`"Klaus"` $\mapsto 7$, `"John"` $\mapsto 2$, `"Mary"` $\mapsto 7]($`"John"`$) = 2$

$[3 \mapsto$ **true**, $3 \mapsto$ **false**$](3) =$ **true** $\sqcap$ **false**

Derived form:

map-expr(expr$_1$)...(expr$_n$)

Example:

$[1 \mapsto [$`"Per"` $\mapsto 5$, `"Jan"` $\mapsto 7]$, $2 \mapsto [\,]](1)($`"Jan"`$)$

Associated Built-in Operators

dom : $(T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \rightarrow T_1$-**infset**

dom $[3 \mapsto$ **true**, $5 \mapsto$ **false**$] = \{3, 5\}$
dom $[3 \mapsto$ **true**, $5 \mapsto$ **false**, $5 \mapsto$ **true**$] = \{3, 5\}$

rng : $(T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \rightarrow T_2$-**infset**

rng $[3 \mapsto$ **false**, $5 \mapsto$ **false**$] = \{$**false**$\}$
rng $[3 \mapsto$ **false**, $5 \mapsto$ **false**, $5 \mapsto$ **true**$] = \{$**false**, **true**$\}$

$\dagger : (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \times (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \rightarrow (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2)$

$[3 \mapsto$ **true**, $5 \mapsto$ **false**$] \dagger [5 \mapsto$ **true**$]$
$= [3 \mapsto$ **true**, $5 \mapsto$ **true**$]$

$\cup : (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \times (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \rightarrow (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2)$

$[3 \mapsto$ **true**, $5 \mapsto$ **false**$] \cup [5 \mapsto$ **true**$]$
$= [3 \mapsto$ **true**, $5 \mapsto$ **false**, $5 \mapsto$ **true**$]$

$\backslash : (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \times T_1$-**infset** $\rightarrow (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2)$

$m \backslash s = [\, d \mapsto m(d) \mid d : T_1 \cdot d \in$ **dom** $m \wedge d \notin s\,]$
$[3 \mapsto$ **true**, $5 \mapsto$ **false**$] \backslash \{5, 7\} = [3 \mapsto$ **true**$]$

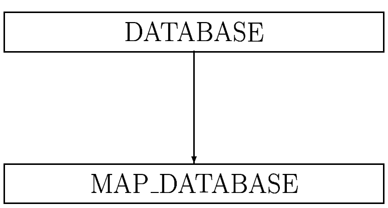
$/ : (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \times T_1$-**infset** $\rightarrow (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2)$

$m \,/\, s = [\, d \mapsto m(d) \mid d : T_1 \cdot d \in$ **dom** $m \wedge d \in s\,]$
$[3 \mapsto$ **true**, $5 \mapsto$ **false**$] \,/\, \{5, 7\} = [5 \mapsto$ **false**$]$

$\circ : (T_2 \underset{m}{\overset{\sim}{\rightarrow}} T_3) \times (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_2) \rightarrow (T_1 \underset{m}{\overset{\sim}{\rightarrow}} T_3)$

$m_1 \circ m_2 =$
$[\, x \mapsto m_1(m_2(x)) \mid x : T_1 \cdot$
$x \in$ **dom** $m_2 \wedge m_2(x) \in$ **dom** $m_1\,]$
$[3 \mapsto$ **true**, $5 \mapsto$ **false**$] \circ [\,$"`Klaus`" $\mapsto 3$, "`John`" $\mapsto 7\,]$
$= [\,$"`Klaus`" $\mapsto$ **true**$\,]$

A Development Step



Database Representations

type Database

empty
insert(k_1, d_1, insert(k_2,d_2,empty))

type Database = (Key × Data)-**set**

{}
{(k_1,d_1), (k_2,d_2)}

type Database = $(\text{Key} \times \text{Data})^*$

$\langle\rangle$
$\langle(k_1,d_1), (k_2,d_2)\rangle$

type Database = Key $\underset{m}{\rightarrow}$ Data

[ ]
$[k_1 \mapsto d_1, k_2 \mapsto d_2]$

MAP_DATABASE =
  class
    type
      Database = Key $\overrightarrow{m}$ Data,
      Key, Data
    value
      empty : Database = [ ],

      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

Subtypes

Contents

Subtypes

- subtype expressions
- maximal types and type checking

Subtype Expressions

Examples:

{| l : **Int*** · **len** l > 0 |}
{| rs : Record-**set** · is_wf_Database(rs) |}

General form:

{| binding : type_expr · *logical*-value_expr |}

Maximal types

The maximal types are

- **Bool**, **Int**, **Real**, **Char**, **Unit**
- Sorts
- Type expressions composed from maximal types and the type constructors ×, **-infset**, $^\omega$, $\overset{\sim}{\underset{m}{\rightarrow}}$ $\overset{\sim}{\rightarrow}$
- Type identifiers defined as abbreviations for maximal types.

Examples of non-maximal types:

- **Nat**, **Text** (= **Char***)
- Type expressions involving the type constructors **-set**, *, $\underset{m}{\rightarrow}$, $\rightarrow$
- Subtypes (unless the type_expr is maximal and the *logical*-value_expr is **true**)

Type checking only checks maximal types.

Example

If f is defined

value
f : **Nat** → **Int**
f(n) ≡
if n = 0 ∨ n = 1 **then** 1
else n ∗ f(n−1) **end**

then f(−1) is not a type error.

Type Definitions

Type Definitions

- abbreviation

  type id = type_expr

- sort

  type id

- variant

  type id == variant_1 | ... | variant_n

  where $n \geq 1$

- record

  type id ::
  d_1 : type_expr_1
  ⋮
  d_n : type_expr_n

  where $n \geq 1$

- union

  type id = id_1 | . . . | id_n

  where $n \geq 2$

type
  type_definition_1,
  ⋮
  type_definition_n

Any order of definitions is allowed.

Cyclic abbreviation definitions are not allowed.

Name versus Structural Equivalence

Abbreviation definitions: structural equivalence

Other type definitions: name equivalence

type
  A = **Int**, B = **Int**
value
  a : A, b : B
axiom
  /* correct */ a = b

type
  A , B
value
  a : A, b : B
axiom
  /* incorrect */ a = b

type
  A :: **Int**, B :: **Int**
value
  a : A, b : B
axiom
  /* incorrect */ a = b,
  /* incorrect */ mk_A(2) = mk_B(2)

Record Definitions

type
  Book ::
    title : Title
    author : Author
    publisher : Publisher
    year : Year
    price : Price ↔ new_price

title : Book → Title
⋮
price : Book → Price   are *destructors*

new_price : Price × Book → Book   is a *reconstructor*

mk_Book :
  Title × Author × Publisher × Year × Price → Book
is the *constructor*

Record Definitions

type
  id ::
    $d_1 : \text{type\_expr}_1 \leftrightarrow r_1$
    ⋮
    $d_n : \text{type\_expr}_n \leftrightarrow r_n$

Values:

$\text{mk\_id}(v_1, \ldots, v_n)$ where $v_i : \text{type\_expr}_i$

Associated functions:

$\text{mk\_id} : \text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n \rightarrow \text{id}$

$d_1 : \text{id} \rightarrow \text{type\_expr}_1$
⋮
$d_n : \text{id} \rightarrow \text{type\_expr}_n$

$r_1 : \text{type\_expr}_1 \times \text{id} \rightarrow \text{id}$
⋮
$r_n : \text{type\_expr}_n \times \text{id} \rightarrow \text{id}$

Records versus Products

1. **type** id :: d_1 : $type\_expr_1$ … d_n : $type\_expr_n$

2. **type** id = $type\_expr_1 \times \ldots \times type\_expr_n$

Use record (1) when

- destructor functions will be useful
- id should be distinct from $type\_expr_1 \times ... \times type\_expr_n$


```
type
  Book ::
    title : Title
    price : Price ↔ new_price
```

is short for

```
type Book
value
  mk_Book : Title × Price → Book,
  title : Book → Title
  price : Book → Price
  new_price : Price × Book → Book
axiom
  [ title_mk_Book ]
    ∀ t : Title, p : Price •
      title(mk_Book(t, p)) ≡ t,
  [ price_mk_Book ]
    ∀ t : Title, p : Price •
      price(mk_Book(t, p)) ≡ p,
  [ new_price_mk_Book ]
    ∀ t : Title, p, p′ : Price •
      new_price(p′, mk_Book(t, p)) ≡ mk_Book(t, p′),
  [ Book_induction ]
     ∀ pred : Book → Bool •
       (∀ t : Title, p : Price • pred(mk_Book(t, p))) ⇒
         (∀ b : Book • pred(b))
```


Variant definitions 1

```
COLOUR =
  class
    type
      Colour == red | green
    value
      is_primary : Colour → Bool
      is_primary(c) ≡
        case c of
          red → true,
          green → false
        end
  end
```



```
type Colour == red | green
```

is short for

```
type Colour
value red, green : Colour
axiom
  [ Colour_disjoint ] red ≠ green,

  [ Colour_induction ]
    ∀ p : Colour → Bool •
      p(red) ∧ p(green) ⇒
        (∀ c : Colour • p(c))
```

Wildcards

```
type Colour == red | green | _
```

is short for

```
type Colour
value red, green : Colour
axiom
  [ Colour_disjoint ] red ≠ green
```

The induction axiom disappears. Further colours may be added during development.

Variant definitions 2

```
TREE =
 class
  type
   Tree ==
    nil |
    node(left : Tree, val : Elem, right : Tree),
   Elem

  value
   traverse : Tree → Elem*
   traverse(t) ≡
    case t of
     nil → ⟨⟩,
     node(l, v, r) →
      traverse(l) ⌢ ⟨v⟩ ⌢ traverse(r)
    end
 end
```



```
type
 Tree ==
  nil |
  node(left : Tree, val : Elem, right : Tree)
```

is short for

```
type Tree
value
 nil : Tree,
 node : Tree × Elem × Tree → Tree,
 left : Tree ~→ Tree,
 val  : Tree ~→ Elem,
 right : Tree ~→ Tree
axiom
 [ left_node ]
  ∀ l, r : Tree, v : Elem •
   left(node(l, v, r)) ≡ l,
 [ val_node ]
  ∀ l, r : Tree, v : Elem •
   val(node(l, v, r)) ≡ v,
 [ right_node ]
  ∀ l, r : Tree, v : Elem •
   right(node(l, v, r)) ≡ r,
```



```
 [ Tree_disjoint ]
  ∀ l, r : Tree, v : Elem •
   nil ≠ node(l, v, r),
 [ Tree_induction ]
  ∀ p : Tree → Bool •
   p(nil) ∧
   (∀ l, r : Tree, v : Elem •
    p(l) ∧ p(r) ⇒ p(node(l, v, r))) ⇒
    (∀ t : Tree • p(t))
```

Imperative Specification

Imperative Specification Example

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:

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

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:

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

Imperative expressions may have effects

effect = state change

state = particular contents of variables

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$_2$}

Evaluation Order

- left to right

The order has particular importance
when the constituent expressions have effects

Examples:

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

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

x + (x := x + 1 ; x) ≡ x := x + 1 ; 2 ∗ x − 1
(x := x + 1 ; x) + x ≡ x := x + 1 ; 2 ∗ x

Equivalence versus Equality

= and ≡ differ in terms of

- undefinedness (**chaos**)
- non-determinism
- effects (variables and communication)

otherwise they are the same.

For example, we can say

factorial(3) = 6
or
factorial(3) ≡ 6

They are both true

When equivalence and equality differ

Assume the variable x currently holds the value 0.

Expression	**Evaluation**
$1 \lceil\rceil 2 = 1 \lceil\rceil 2$	**true** $\lceil\rceil$ **false**
$1 \lceil\rceil 2 \equiv 1 \lceil\rceil 2$	**true**
while true do skip end = **chaos**	**chaos**
while true do skip end ≡ **chaos**	**true**
$((x := x + 1 ; 1) = (x := x + 1 ; x))$	$x := 2$; **false**
$((x := x + 1 ; 1) \equiv (x := x + 1 ; x))$	**true**

- ≡ and = are the same if the arguments are convergent and pure.
- ≡ is always defined.
- ≡ compares effects as well as results; = only compares results
- ≡ has hypothetical evaluation; = has left-to-right evaluation.
- ≡ gives no effects; = may give effects.

Imperative Preconditions

Example:

DECREASE =
  extend COUNTER **with**
  class
    value
      decrease : **Unit** $\stackrel{\sim}{\to}$ **write** counter **Nat**
      decrease() ≡ counter := counter − 1 ; counter
      pre counter > 0
  end

preconditions must be read-only.

Some Examples

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

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

is wrong

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

is correct


```
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

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


Applicative to imperative transformation

- Remove definition of type of interest.
- Add variable(s) to model concrete type of interest.
- Remove type of interest from function signatures; fill holes with **Unit**.
- Give type "**Unit** → **write** ... **Unit**" to each constant "c" of type of interest, and replace "=" by "c() ≡".
- Insert write accesses to generators.
- Insert read accesses to observers.
- Remove formal parameters representing type of interest.
- Replace occurrences of type of interest parameters with references to variable(s).
- For generators, insert assignments.

This is for "leaf" modules in a hierarchy. We will see later an example of how to transform non-leaf modules.


```
I_DATABASE =
  class
    type
      Key, Data
    variable
      database : Key →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 ~→ read database Data
      lookup(k) ≡ database(k)
      pre defined(k)
   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 $\equiv$ ()

Example:

variable counter : **Nat**
value
  decrease : **Unit** $\rightarrow$ **write** counter **Unit**
  decrease() $\equiv$
    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**

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

FRACTION_SUM =
  class
    variable
      counter : **Nat**,
      result : **Real**
    value
      fraction_sum : **Nat** $\xrightarrow{\sim}$ **write** counter, result **Unit**
      fraction_sum(n) $\equiv$
        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**

```
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
 readonly_list–value_expr$_1$ · *readonly_logical*–value_expr$_p$
do
 unit-value_expr$_2$
end

has type **Unit**


```
FRACTION_SUM =
  class
    variable
      result : Real
    value
      fraction_sum : Nat ∼→ 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
```


Local Expressions

General form:

local
 declaration$_1$ … declaration$_n$
in value_expr **end**

```
FRACTION_SUM =
  class
    value
      fraction_sum : Nat ∼→ Real
      fraction_sum(n) ≡
        local
          variable
            counter : Nat := n,
            result : Real := 0.0
          value
            calc_fraction :
              Unit → write counter, result Unit
            calc_fraction() ≡
              result := result + 1.0/(real counter) ;
              counter := counter − 1
        in
          while counter > 0 do calc_fraction() end ; result
        end
      pre n > 0
  end
```


Local versus Let

```
local
  declaration₁ ... declarationₙ
in value_expr end
```

```
let
  let_def₁, ... ,let_defₙ
in value_expr end
```

In local expressions:

- local declarations of:
  all kinds (not just of values)
- scope of declaration$_i$:
  all the declarations
  (not just the declaration$_j$, $j > i$)

Pre-names

```
CHOOSE =
  class
    variable
      set : Int-set
    value
      choose : Unit ∼→ write set Int
      choose() as i post i ∈ set` ∧ set = set`\{i}
      pre set ≠ {}
  end
```

Avoiding pre-names:

```
value
  choose : Unit ∼→ write set Int
axiom
  let s = set in
    choose() as i post i ∈ s ∧ set = s\{i}
    pre set ≠ {}
  end
```



```
INSERT_SORTED =
  class
    variable
      list : Int* := ⟨⟩
    value
      insert : Int → write list Unit
      insert(i)
        post is_permutation(list, ⟨i⟩ ⌢ list`) ∧ is_sorted(list)

      /* auxiliary functions */
      is_sorted : Int* → Bool,
      is_sorted(l) ≡
        (∀ idx1,idx2 : Nat ·
          ({idx1,idx2} ⊆ inds l ∧ idx1 < idx2) ⇒
            l(idx1) ≤ l(idx2)),

      is_permutation : Int* × Int* → Bool
      is_permutation(l1,l2) ≡
        (∀ i : Int · count(i, l1) = count(i, l2)),

      count : Int × Int* → Nat
      count(i, l) ≡
        card {idx | idx : Nat ·
                 idx ∈ inds l1 ∧ l1(idx) = i}

  end
```

Concurrency

Agenda

- channel definitions (**channel** c : T)
- (process) expressions:
  - concurrency (e1 ∥ e2)
  - communication (c?, c!e)
  - choice (e1 ⌷ e2, e1 ⌈⌉ e2)
  - **stop** (deadlock)
- functions with channel access (value f : **Unit** $\stackrel{\sim}{\rightarrow}$ **in** c **out** c **Unit**)
  - applicative versus imperative

Composition of Expressions

Composition:

- sequential:

  value_expr_1 ; value_expr_2

- concurrent:

  value_expr_1 ∥ value_expr_2

1. has type **Unit**
2. value_expr_1 and value_expr_2 must have type **Unit**
3. value_expr_1 and value_expr_2 recommended to be assignment-disjoint

Concurrency

Concurrency is necessary in particular for describing distributed systems.

Concurrent systems in general may communicate through

- shared variables
- message passing

RSL uses message passing.

Channel Definitions

channel
channel_definition$_1$,
⋮
channel_definition$_n$

single channel definition:

id : type_expr

multiple channel definition:

id$_1$,...,id$_n$: type_expr

Communication Expressions

channel id : type_expr

Communication expressions:

- input expressions: id ?
- output expressions: id ! value_expr

Example

channel c : **Int**
variable x : **Int**

... x := c? ∥ c!5 ...

communication may

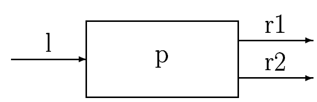
1. be synchronized: x := 5
2. be interleaved, if other behaviours are possible

channel c : **Int**
variable x : **Int**

... (x := c? ∥ c!5) ∥ c!7 ...

may give: x := 7; c!5 or ...

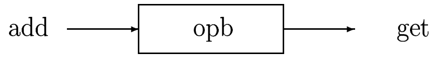
Example



channel
l, r1, r2: **Int**

value
p : **Unit** → **in** l **out** r1, r2 **Unit**
p() ≡
let e = l? **in** (r1!e ∥ r2!e) **end**; p()

Example



ONE_PLACE_BUFFER =
 class
  type Elem
  channel add, get : Elem
  value
   opb : **Unit** → **in** add **out** get **Unit**
   opb() ≡ **let** v = add? **in** get!v **end** ; opb()
 end

Functions with Channel Access

Function types revisited:

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

access_desc_i:

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

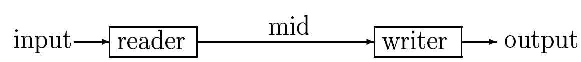
Function definitions:

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

Pure and Read-only Revisited

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
- *input* from a channel (e.g c?)
- *output* to a channel (e.g. c!e)
- *access* a channel: input from or output to a channel
- *pure*: does not statically access any variable or channel
- *read-only*: does not statically write to any variable and does not statically access any channel



READER_WRITER =
 class
  type Elem
  channel input, output, mid : Elem
  value
   reader : **Unit** → **in** input **out** mid **Unit**
   reader() ≡
    let v = input? **in** mid ! v **end** ; reader(),
   writer : **Unit** → **in** mid **out** output **Unit**
   writer() ≡
    let v = mid? **in** output ! v **end** ; writer()
 end

SYSTEM = **extend** READER_WRITER **with**
 class
  value
   system : **Unit** →
      in input, mid **out** output, mid **Unit**
   system() ≡ reader() ∥ writer()
 end

```
  system() ≡
    let v = input? in mid ! v end ; reader()
    ||
    let v = mid? in output ! v end ; writer()
```

We should make the channel *mid* unavailable to any other processes.

Restricting channel access

```
 local
   channel c : Int
 in x := c? || c!5 end
≡
 x := 5
```


Deadlock

```
 local
   channel c : Int
 in  c!2 || c!5 end
≡
 stop
```

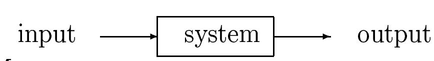
Note the difference between **stop** and **skip**.

skip *terminates*, which means the next expression in sequence may execute.

stop does not terminate.

So

```
 skip ; e ≡ e
 stop ; e ≡ stop
```


```
SYSTEM =
 class
   type Elem
   channel input, output : Elem
   value
     system : Unit → in input out output Unit
     system() ≡
       local
         channel mid : Elem
         value
           reader : Unit → in input out mid Unit
           reader() ≡
             let v = input? in mid ! v end ; reader(),
           writer : Unit → in mid out output Unit
           writer() ≡
             let v = mid? in output ! v end ; writer()
       in reader() || writer() end
 end
```

External choice

General form:

$\text{value\_expr}_1 \; [] \; \text{value\_expr}_2$

Example:

- The value expression

  v:=c1? [] c2!e

  will:

  - input from c_1 if a value expression is willing to output to c_1 but no value expression is willing to input from c_2;
  - output to c_2 if a value expression is willing to input from c_2 but no value expression is willing to output to c_1;
  - either input from c_1 or output to c_2 if a value expression is willing to output to c_1 and a value expression is willing to input from c_2;
  - deadlock if no value expression is ever willing to output to c_1 and no value expression is ever willing to input from c_2.

Internal choice

General form:

$\text{value\_expr}_1 \; \sqcap \; \text{value\_expr}_2$

Example:

- The value expression

  v:=c1? ⊓ c2!e

  will:

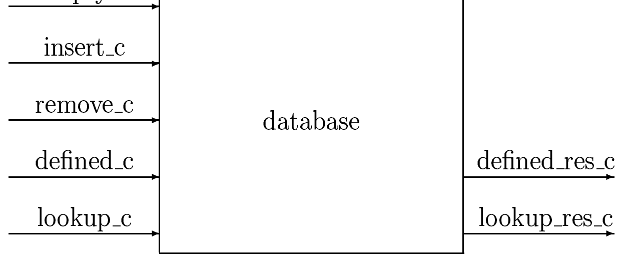
  - either deadlock or input from c_1 if a value expression is willing to output to c_1 but no value expression is willing to input from c_2;
  - either deadlock or output to c_2 if a value expression is willing to input from c_2 but no value expression is willing to output to c_1;
  - either input from c_1 or output to c_2 if a value expression is willing to output to c_1 and a value expression is willing to input from c_2;
  - deadlock if no value expression is ever willing to output to c_1 and no value expression is ever willing to input from c_2.


```
MANY_PLACE_BUFFER =
  class
    type
      Elem,
      Buffer = Elem*
    channel
      empty : Unit,
      add, get : Elem
    value
      mpb : Buffer → in empty, add out get Unit
      mpb(b) ≡
        empty? ; mpb(⟨⟩)
        []
        let v = add? in mpb(b ⌢ ⟨v⟩) end
        []
        if b ≠ ⟨⟩
          then get ! hd b ; mpb(tl b)
          else stop end
  end
```



```
  mpb(⟨⟩)
≡
  empty? ; mpb(⟨⟩)
  []
  let v = add? in mpb(⟨⟩ ⌢ ⟨v⟩) end
  []
  if ⟨⟩ ≠ ⟨⟩ then get ! hd ⟨⟩ ; mpb(tl ⟨⟩) else stop end
≡
  empty? ; mpb(⟨⟩)
  []
  let v = add? in mpb(⟨v⟩) end
  []
  stop
≡
  empty? ; mpb(⟨⟩)
  []
  let v = add? in mpb(⟨v⟩) end
```

```
MANY_PLACE_BUFFER =
  class
    type
      Elem
    channel
      empty : Unit,
      add, get : Elem
    value
      mpb : Unit → in empty, add out get Unit
      mpb() ≡
        local
          type Buffer = Elem*
          variable buffer : Buffer := ⟨⟩
        in
          while true do
            empty? ; buffer := ⟨⟩
            ⌷
            let v = add? in buffer := buffer ^ ⟨v⟩ end
            ⌷
            if buffer ≠ ⟨⟩
            then get ! hd buffer ; buffer := tl buffer
            else stop
            end
          end
        end
  end
```

Imperative to concurrent transformation

- Insert an object instantiating the imperative sequential module, and hide it.
- Define channels for each observer and generator; at least one channel for each. Hide them.
- Define a "server" process:
  - type "**Unit** → **in** ... **out** ... **write** I.**any** **Unit**"
  - body is a **while true do** loop
  - loop body is an external choice between clauses, one clause for each observer and each generator
  - each clause inputs parameters (if any); calls corresponding function I.f; outputs result (if any). Must do at least one communication.

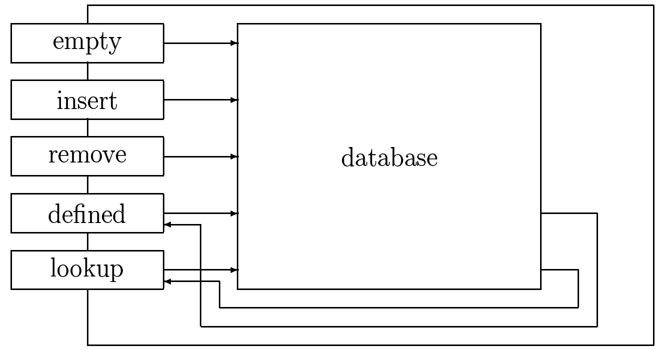
  Hide it.
- Define an "init" process with the same type as the server that initialises the imperative object and calls the server.
- Define "interface functions" mirroring clauses in server. These *have no accesses to the imperative object.*

This is for "leaf" modules in a hierarchy. We will see later an example of how to transform non-leaf modules.


```
C_DATABASE =
  hide I, database in
  class
    object I : I_DATABASE
    type
      Key = I.Key,
      Data = I.Data,
      Result == not_found | res(Data)
    channel
      empty_c : Unit,
      insert_c : Key × Data,
      remove_c, defined_c, lookup_c : Key,
      defined_res_c : Bool,
      lookup_res_c : Result
    value
      init : Unit →
        in empty_c, insert_c, remove_c, defined_c, lookup_c
        out defined_res_c, lookup_res_c write I.any Unit
      init() ≡ I.empty() ; database(),

      database : Unit →
        in empty_c, insert_c, remove_c, defined_c, lookup_c
        out defined_res_c, lookup_res_c write I.any Unit
```

```
  database() ≡
    while true do
      empty_c? ; I.empty()
      []
      let (k,d) = insert_c? in I.insert(k,d) end
      []
      let k = remove_c? in I.remove(k) end
      []
      let k = defined_c? in
        defined_res_c ! I.defined(k)
      end
      []
      let k = lookup_c? in
        if I.defined(k)
        then lookup_res_c ! res(I.lookup(k))
        else lookup_res_c ! not_found
        end
      end
    end
end
```


Encapsulation with Interface Functions




```
INTERFACED_DATABASE =
  hide empty_c, insert_c, remove_c, defined_c, lookup_c,
      defined_res_c, lookup_res_c in
  extend C_DATABASE with
  class
    value
      empty : Unit → out any Unit
      empty() ≡ empty_c ! (),

      insert : Key × Data → out any Unit
      insert(k,d) ≡ insert_c ! (k,d),

      remove : Key → out any Unit
      remove(k) ≡ remove_c ! k,

      defined : Key → in any out any Bool
      defined(k) ≡ defined_c ! k ; defined_res_c?,

      lookup : Key → in any out any Result
      lookup(k) ≡ lookup_c ! k ; lookup_res_c?
  end
```


Modularity

Specifications

An RSL specification consists of

- module definitions

A module contains definitions of

- types
- values
- variables
- channels
- modules
- axioms

Modularity

Modules are the building blocks.

Purposes:

- Readability
- Separate development
- Reuse

Schemes and Objects

Modules are either schemes or objects.

A scheme denotes a class of models

scheme id = class_expr

An object denotes a single model

object id : class_expr

Class Expressions

- basic
- extending
- renaming
- hiding
- instantiation

Extension

General form:

extend class_expr_1 **with** class_expr_2

class_expr_1 and class_expr_2 must be compatible

Context dependent expansion:

extend
  class decl_string_1 **end**
with
  class decl_string_2 **end**

expands to:

class
  decl_string_1
  decl_string_2
end

Renaming

General form:

use
  id_{new_1} **for** id_{old_1}, ... ,id_{new_n} **for** id_{old_n}
in class_expr

For example

scheme BUFFER =
  use
    add **for** enq, get **for** deq, Buffer **for** Queue
  in QUEUE

Hiding

General form:

hide id_1,...,id_n **in** class_expr

Hidden entities

1. are not visible outside
2. need not be implemented

Typically use:

1. prevention of unintended access to variables and/or channels
2. hiding of auxiliary functions

Objects

scheme BUFFER =
  class
    variable buff : $\textbf{Int}^*$
    value
      is_empty : **Unit** $\rightarrow$ **read** buff **Bool**
      ...
  end

object
  B1 : BUFFER,
  B2 : BUFFER,

scheme SYS =
  class
    value
      one_is_empty :
        Unit $\rightarrow$ **read** B1.buff B2.buff **Bool**
      one_is_empty() $\equiv$
        B1.is_empty() $\vee$ B2.is_empty()
  end

B1.buff and B2.buff are distinct

Module Nesting

scheme
SYS =
class
object
B1 : BUFFER,
B2 : BUFFER
value
one_is_empty :
Unit → **read** B1.buff B2.buff **Bool**
one_is_empty() ≡
B1.is_empty() ∨ B2.is_empty()
end

Building hierarchies

Suppose we have a system that needs a database component.

There are several ways we can construct the specification:

- merging the system and database definitions in one class
- extending the database class with the system class
- making a hierarchy with a database object

Merging the definitions in one class

scheme SYSTEM =
class
/∗ database ∗/
⋮
/∗ system ∗/
⋮
end

- Hard to read
- Database cannot be reused
- Hard to make database private to system
- Problem of name clashes between two parts

Extending the database

scheme DATABASE = ...

scheme SYSTEM =
extend DATABASE **with** ...

- Easier to read
- Database can be reused
- Hard to make database private to system
- Problem of name clashes between two parts

Making a hierarchy with a database object

scheme DATABASE = ...

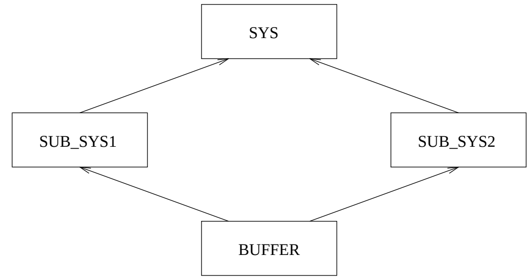
scheme SYSTEM =
  class
    object DB : DATABASE
    ⋮
  end

- Easier to read
- Database can be reused
- Easy to make database private to system

  scheme SYSTEM =
    hide DB **in**
    class
      object DB : DATABASE
      ⋮
    end

- No problem of name clashes between two parts

Sharing



scheme BUFFER = ...

scheme SUB_SYS1 =
  class object B : BUFFER ... **end**
scheme SUB_SYS2 =
  class object B : BUFFER ... **end**

scheme SYS =
  class
    object
      O1 : SUB_SYS1,
      O2 : SUB_SYS2
    ⋮
  end

We get two buffer variables (O1.B.buff and O2.B.buff)

Sharing using Global Objects

object
  B : BUFFER

scheme
  SUB_SYS1 = **class** ... B.buff ... **end**,
  SUB_SYS2 = **class** ... B.buff ... **end**,

  SYS =
    class
      object
        O1 : SUB_SYS1,
        O2 : SUB_SYS2
    end

We get only one buffer: B.buff

Parameterization - Example

scheme BUFFER =
  class
    type Elem
    variable buff : Elem*
    value
      empty : **Unit** → **write** buff **Unit**
      empty() ≡ buff := ⟨⟩,

      add : Elem → **write** buff **Unit**
      add(e) ≡ buff := buff ⌢ ⟨e⟩
  end

is better expressed using parameterization

scheme ELEM = **class type** Elem **end**

scheme BUFFER(E : ELEM) =
  class
    variable buff : E.Elem*
    value
      empty : **Unit** → **write** buff **Unit**
      empty() ≡ buff := ⟨⟩,

      add : E.Elem → **write** buff **Unit**
      add(e) ≡ buff := buff ⌢ ⟨e⟩
  end

Instantiation - Example

```
object
  INTEGER :
    class
      type Elem = Int
    end,

  INTEGER_BUFFER : BUFFER(INTEGER)
```

If we expand INTEGER_BUFFER:

```
  INTEGER_BUFFER :
    class
      variable buff : INTEGER.Elem*
      value
        empty : Unit → write buff Unit
        empty() ≡ buff := ⟨⟩,

        add : INTEGER.Elem → write buff Unit
        add(e) ≡ buff := buff ⁀ ⟨e⟩
    end
```


Actual versus Formal Parameters

```
scheme S(X : FC)
object A : AC,
```

... S(A) ...

Context condition: AC must statically implement FC

Sharing by Parameterization

```
scheme
  SUB_SYS1(B : BUFFER) =
    class ... B.buff ... end,

  SUB_SYS2(B : BUFFER) =
    class ... B.buff ... end,


  SYS =
    class
      object
        B : BUFFER,
        O1 : SUB_SYS1(B),
        O2 : SUB_SYS2(B)
    end
```


Summary

Modules:

- schemes
- objects

Class expressions:

- basic
- extending
- hiding
- renaming
- instantiation

RAISE Method

Agenda

1. Why formal methods?
2. Characteristics of formal methods
3. RAISE method
4. Implementation relation: requirements and definition
5. Example

Why formal methods?

To produce software that is

- more likely to be correct
- more reliable
- better documented
- more easily maintainable

What is formality?

- a language — symbols and grammar rules for constructing terms
- (usually) rules for deciding if terms are well formed (e.g. scope, typing rules)
- a semantics — a description of what terms mean
- a logic — a set of rules for determining if predicates about terms are true

Programming languages are not formal according to this definition because they lack a logic.

Characteristics of formal methods

- Precise notation
- Abstraction (*what* rather than *how*)
- Stepwise development (gradual commitment)
- Proof opportunities and justifications
- Structuring based on compositionality
- Guidelines for quality assurance

Rigorous methods

Choice of level of formality. E.g.

1. No proof opportunities generated or checked
2. Proof opportunities generated and inspected but not proved
3. Proof opportunities generated and proved with some informal steps — "it follows immediately that ..."
4. Proof opportunities generated and proved formally

All formal methods are in fact rigorous. But only a method with a formal basis can be rigorous, because it must always be possible to say "I am not sure if it does follow. Please prove it."

Current state of the art is the first three levels.

Why formality?

When we try to be formal, what are our objectives?

- To provide unambiguous specification techniques
- To provide a theory to support reliable reasoning
- To provide descriptions in precise and machine independent terms

Examples of formal methods

- "Model based" approaches like Z and VDM
- "Algebraic" like Larch and OBJ
- "Process algebras" like CSP, CCS and Lotos
- RAISE
- "Modal" logics, especially temporal logics for concurrency

RAISE contains most of the features of the first three

Validation

Are we building the "right" system?

- the specification is *validated* against the requirements
- a specification may be viewed as a *theory* about the requirements. Viewed in this way we can:
  - look at the requirements, identify a "requirements issue", and see if the specification "predicted" this requirement
  - look at the specification, deduce a property from the specification, and check the requirements to see if it is a valid prediction.

Verification

Are we building the system "right"?

The *implementation relation* captures the notion of correctness of a development step.

An assertion of implementation may be *statically* checked by tools. This shows that the *signature* has been maintained, e.g.

- no entities (types, values etc.) have been omitted
- the types of values, variables etc. have not been changed

If statically correct, the assertion may be *dynamically* checked by proof (or "justification"). This shows that *properties* have been maintained, e.g.

- subtypes have not changed
- axioms are true
- postconditions are established by concrete definitions

An implementation may have more entities and/or properties; it may not have fewer or weaker or different ones.

The RAISE method

The method is based on 2 key notions:

Separate development

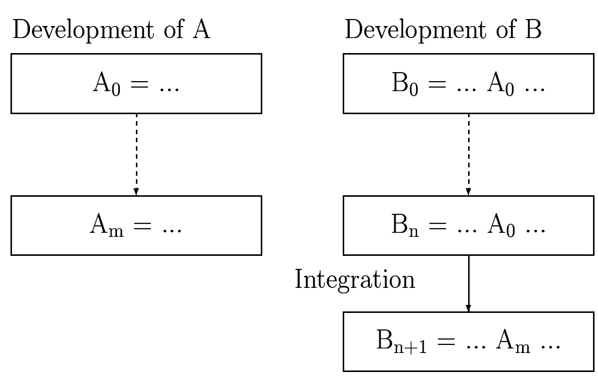
- Divide systems into subsystems
- Take the initial specification of the subsystem as a "contract" between its developers and the system developers
- Develop parts separately. Implementation rules mean that as long as contracts are not changed integration will preserve the original properties.

Stepwise development Repeatedly:

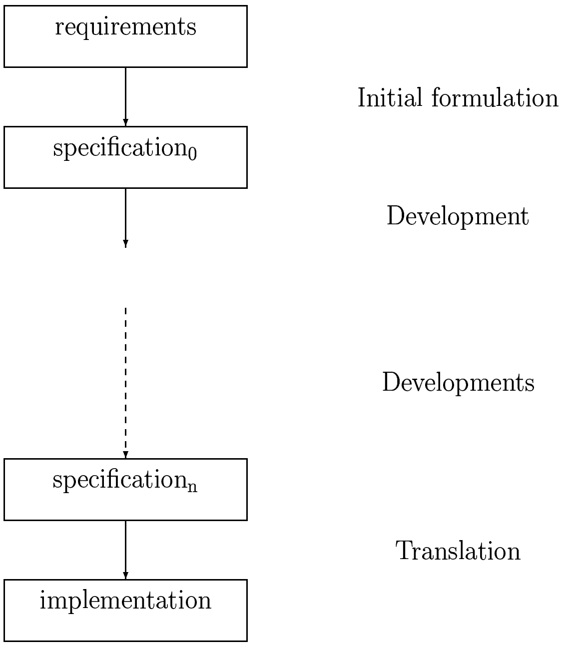
- Specify a new, more detailed version
- Assert the relation (if possible implementation) between this and the previous version, and justify that it holds

This follows the "invent and verify" paradigm.

Separate Development



Stepwise development



Implementation relation: requirements

- Property preservation
- Substitutivity
- Transitivity

These are needed for separate and stepwise development.

Implementation relation

- new signature includes the old one (statically decidable)
- old properties preserved by the new one ($\Rightarrow$ implementation conditions)

Underspecification

value x : **Int**
axiom x > 0

x is underspecified, may be implemented by:

value x : **Int** = 1

or

value x : **Int** = 2

or ...

Example 1

scheme S0 =
  class
    value x : **Int**
    axiom x ≥ 0
  end

scheme S1 =
  class
    value
      x : **Int** = 2
  end

scheme S2 =
  class
    value
      x : **Int** = 2
      y : **Int** = 0
  end

Does S1 or S2 implement S0?

Does S2 implement S1?

Example 2

scheme S0 =
  hide z **in class**
    value x, y, z : **Int**
    axiom x > z ∧ z > y
  end

scheme S1 =
  class
    value
      x : **Int** = 1
      y : **Int** = 0
  end

scheme S2 =
  class
    value
      x : **Int** = 2
      y : **Int** = 0
  end

Does S1 or S2 implement S0?

Showing implementation with hidden entities

1. Make an extension defining the hidden entities.
2. Show the extension is conservative.
3. Show the extension implements the old.

For example:

1. Define

   scheme S2′ =
     extend S2 **with**
     hide z **in class value** z : **Int** = 1 **end**

2. Extension is conservative since definition of z cannot affect x or y.
3. We can prove S2′ implements S0.

Example RSL specification

scheme REGISTRATION1 =
  class
    type
      Register = Name-**set**,
      Name = **Text**

    value
      enrol : Name × Register → Register
      enrol(n,r) ≡ r ∪ {n},

      leave : Name × Register → Register
      leave(n,r) ≡ r \ {n},

      registered : Name × Register → **Bool**
      registered(n,r) ≡ n ∈ r
  end

An abstract version of REGISTRATION

scheme REGISTRATION0 =
  class
    type Register, Name
    value
      /* generators */
      empty : Register,
      enrol : Name × Register → Register,
      leave : Name × Register → Register
      /* observers */
      registered : Name × Register → **Bool**

REGISTRATION0 continued

axiom
  [registered_empty]
    ∀ n : Name ·
      registered(n, empty) ≡ **false**,

  [registered_enrol]
    ∀ n, n′ : Name, r : Register ·
      registered(n′, enrol(n, r)) ≡
        n′ = n ∨ registered(n′ , r),

  [registered_leave]
    ∀ n, n′ : Name, r : Register ·
      registered(n′, leave(n, r)) ≡
        n′ ≠ n ∧ registered(n′ , r)
end

Each axiom relates an observer to a generator.

Showing implementation for REGISTRATION

Assert the relation:

"REGISTRATION1 ⪯ REGISTRATION0"
i.e.
"REGISTRATION1 implements REGISTRATION0"

Static check

Signature of REGISTRATION0 included in REGISTRATION1 Checked by tool. False!

Dynamic check

Properties of REGISTRATION0 hold in REGISTRATION1. Proof opportunities. E.g.

⌞ **in** REGISTRATION1 ⊢
    ∀ n, n′ : Name, r : Register ·
      registered(n′, enrol(n, r)) ≡
        n′ = n ∨ registered(n′ , r) ⌟

Reduces to

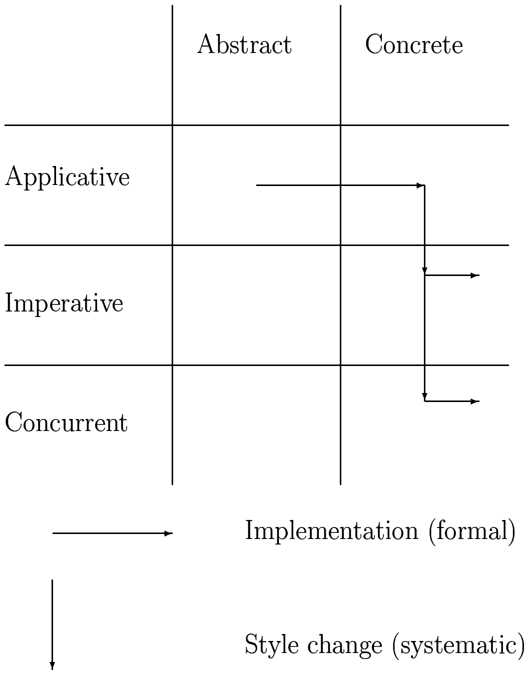
⌞ $n' \in (r \cup \{n\}) \equiv n' = n \vee n' \in r$ ⌟

This can be proved.

Design

- removing underspecification
  - abstract types to concrete types
  - more explicit value definitions
- changing style
  - applicative/imperative
  - sequential/concurrent
- providing more efficient algorithms

Typical Development



A concrete, imperative version

```
scheme I_REGISTRATION1 =
  hide register, Register in class
    type
      Register = Name-set,
      Name = Text

    variable register : Register := {}

    value
      empty : Unit → write any Unit
      empty() ≡ register := {},

      enrol : Name → write any Unit
      enrol(n) ≡ register := register ∪ {n},

      leave : Name → write any Unit
      leave(n) ≡ register := register \ {n},

      registered : Name → read any Bool
      registered(n) ≡ n ∈ register
  end
```


A concrete, concurrent version

```
scheme C_REGISTRATION1 =
  hide main, CH, I in class
    type
      Name = Text
    object
      CH :
        class
          channel
            empty : Unit,
            enrol, leave, registered : Name,
            registered_res : Bool
        end,
        I : I_REGISTRATION1
```

We first define some channels and an instantiation of the sequential imperative version.

The concurrent version will act as a communication shell around the sequential version.

Concurrent version continued

```
value
  /* initial */
  init : Unit → in any out any write any Unit
  init() ≡ I.empty() ; main(),

  /* main */
  main : Unit → in any out any write any Unit
  main() ≡
    while true do
      CH.empty? ; I.empty()
      []
      let n = CH.enrol? in I.enrol(n) end
      []
      let n = CH.leave? in I.leave(n) end
      []
      let n = CH.registered? in
        CH.registered_res ! I.registered(n) end
    end
```

init is used to start the *main* "server" process.

Concurrent version continued

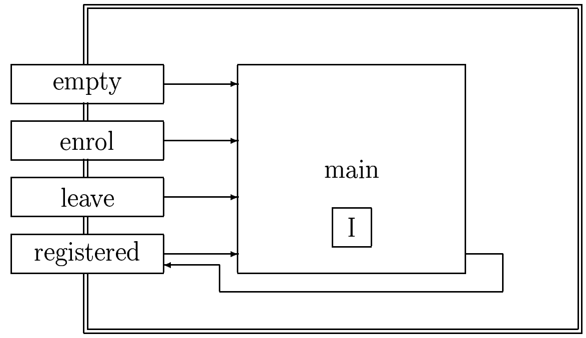
value
  empty : **Unit** → **in any out any Unit**
  empty() ≡ CH.empty!(),

  enrol : Name → **in any out any Unit**
  enrol(n) ≡ CH.enrol!n,

  leave : Name → **in any out any Unit**
  leave(n) ≡ CH.leave!n,

  registered : Name → **in any out any Bool**
  registered(n) ≡ CH.register!n ; CH.register_res?
end

Lastly add the 'interface' functions which provide the user interface to the registration system.



Translation

- manual translation
- automatic translation (to Ada and C++)

of low-level RSL (e.g. concrete types and explicit value definitions)

RAISE Method Summary

- separate development
- stepwise development
- rigorous
- invent and verify
- implementation relation

Harbour example

Requirements

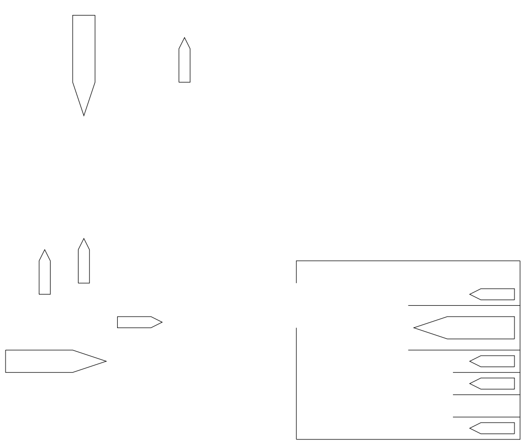
Ships arriving at a harbour have to be allocated berths in the harbour which are vacant and which they will fit, or wait in a "pool" until a suitable berth is available.

Develop a system providing the following functions to allow the harbour master to control the movement of ships in and out of the harbour:

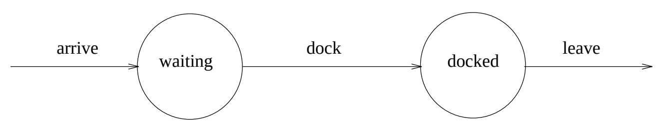
arrive: to register the arrival of a ship

dock: to register a ship docking in a berth

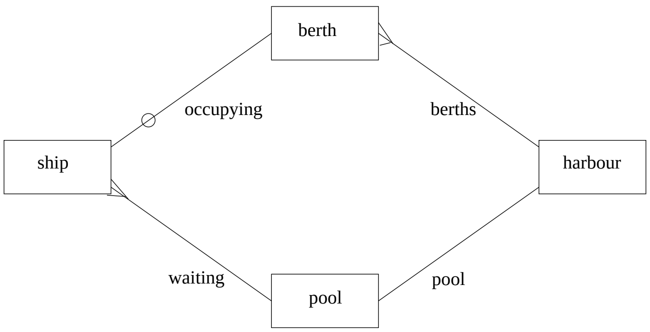
leave: to register a ship leaving a berth



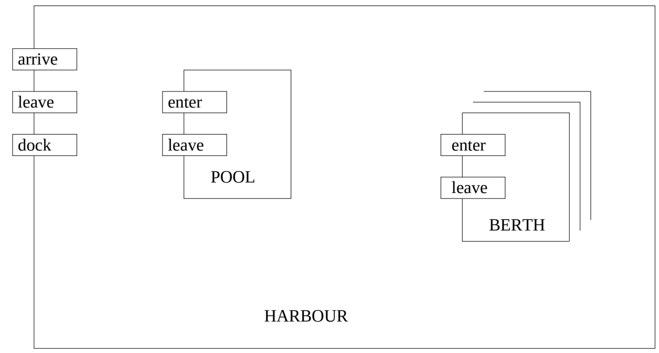
State transitions for ships



Entity relationship diagram



Harbour objects



Possible attributes

- Harbour
  - Pool (S)
  - (Set of) berths (S)
- Pool
  - (Set of) ships (D)
- Berth
  - Occupancy (D)
  - Size (S)
- Ship
  - Location (D)
  - Name (S)
  - Size (S)

"S" indicates a static attribute
"D" indicates a dynamic (state-dependent) attribute

Design decisions

- Don't know components of "size" — length, width, depth/draught etc. So define

  value
  fits : Ship × Berth → **Bool**

  and leave underspecified.
- Name of ship unnecessary
- Location of ship can be calculated (to avoid duplication)

TYPES module

```
scheme TYPES =
  class
    type
      Ship, Berth,
      Occupancy == vacant | occupied_by(occupant : Ship)
    value
      fits : Ship × Berth → Bool
  end
```

We then make a global object from TYPES:

```
object T : TYPES
```


Consistency

1. a ship can't be in two places at once
2. at most one ship can be in any one berth
3. a ship can only be in a berth it fits

Two possibilities:

- build into model
- express as a predicate

2nd consistency condition in *Occupancy*; for 1st and 3rd we will use a predicate.

Abstract applicative specification

```
scheme A_HARBOUR0 =
hide consistent in
class
  type Harbour
  value
    /∗ generators ∗/
    arrives : T.Ship × Harbour ~→ Harbour,
    docks : T.Ship × T.Berth × Harbour ~→ Harbour,
    leaves : T.Ship × T.Berth × Harbour ~→ Harbour,

    /∗ observers ∗/
    waiting : T.Ship × Harbour → Bool,
    occupancy : T.Berth × Harbour → T.Occupancy,
```



```
    /∗ derived ∗/
    consistent : Harbour → Bool
    consistent(h) ≡
      (∀ s : T.Ship •
         ∼ (waiting(s, h) ∧ is_docked(s, h)) ∧
         (∀ b1, b2 : T.Berth •
           occupancy(b1, h) = T.occupied_by(s) ∧
           occupancy(b2, h) = T.occupied_by(s) ⇒
             b1 = b2) ∧
         (∀ b : T.Berth •
           occupancy(b, h) = T.occupied_by(s) ⇒
             T.fits(s, b))),
```

```
is_docked : T.Ship × Harbour → Bool
is_docked(s, h) ≡
  (∃ b : T.Berth ·
    occupancy(b, h) = T.occupied_by(s)),

/* guards */
can_arrive : T.Ship × Harbour → Bool
can_arrive(s, h) ≡
  ∼waiting(s, h) ∧ ∼is_docked(s, h),

can_dock : T.Ship × T.Berth × Harbour → Bool
can_dock(s, b, h) ≡
  waiting(s, h) ∧ ∼ is_docked(s, h) ∧
  occupancy(b, h) = T.vacant ∧ T.fits(s, b),

can_leave : T.Ship × T.Berth × Harbour → Bool
can_leave(s, b, h) ≡
  occupancy(b, h) = T.occupied_by(s)
```



```
axiom
  [waiting_arrives]
    ∀ h : Harbour, s1, s2 : T.Ship ·
      waiting(s2, arrives(s1, h)) ≡
        s1 = s2 ∨ waiting(s2, h)
      pre can_arrive(s1, h),

  [waiting_docks]
    ∀ h : Harbour, s1, s2 : T.Ship, b : T.Berth ·
      waiting(s2, docks(s1, b, h)) ≡
        s1 ≠ s2 ∧ waiting(s2, h)
      pre can_dock(s1, b, h),

  [waiting_leaves]
    ∀ h : Harbour, s1, s2 : T.Ship, b : T.Berth ·
      waiting(s2, leaves(s1, b, h)) ≡
        waiting(s2, h)
      pre can_leave(s1, b, h),
```



```
  [occupancy_arrives]
    ∀ h : Harbour, s : T.Ship, b : T.Berth ·
      occupancy(b, arrives(s, h)) ≡
        occupancy(b, h)
      pre can_arrive(s, h),

  [occupancy_docks]
    ∀ h : Harbour, s : T.Ship, b1, b2 : T.Berth ·
      occupancy(b2, docks(s, b1, h)) ≡
        if b1 = b2 then T.occupied_by(s)
        else occupancy(b2, h) end
      pre can_dock(s, b1, h),

  [occupancy_leaves]
    ∀ h : Harbour, s : T.Ship, b1, b2 : T.Berth ·
      occupancy(b2, leaves(s, b1, h)) ≡
        if b1 = b2 then T.vacant
        else occupancy(b2, h) end
      pre can_leave(s, b1, h),
```



```
  [arrives_consistent]
    ∀ h : Harbour, s : T.Ship ·
      arrives(s, h) as h′ post consistent(h′)
        pre consistent(h) ∧ can_arrive(s, h),

  [docks_consistent]
    ∀ h : Harbour, s : T.Ship, b : T.Berth ·
      docks(s, b, h) as h′ post consistent(h′)
        pre consistent(h) ∧ can_dock(s, b, h),

  [leaves_consistent]
    ∀ h : Harbour, s : T.Ship, b : T.Berth ·
      leaves(s, b, h) as h′ post consistent(h′)
        pre consistent(h) ∧ can_leave(s, b, h)
end
```

Validation

Have we met the main requirements?

1. Ships can arrive and will be registered
2. Ships can be docked when a suitable berth is free
3. Docked ships can leave
4. Ships can only be allocated to berths they fit
5. Any ship will eventually get a berth
6. Any ship waiting more than 2 days will be flagged
7. ...

Requirements might

- be met
- be deferred; be met later
- be removed; not be met
- make us rework the specification

Next steps

- Make the state more concrete; introduce modules for POOL and BERTHS
- Define HARBOUR functions in terms of functions from POOL and BERTHS

Decide to use standard modules A_SET for POOL and A_ARRAY for BERTHS.

A_SET

```
scheme ELEM = class type Elem end

scheme A_SET(E : ELEM) =
class
  type Set
  value
    /∗ generators ∗/
    empty : Set,
    add : E.Elem × Set → Set,
    remove : E.Elem × Set → Set,

    /∗ observer ∗/
    is_in : E.Elem × Set → Bool
  axiom
    [ is_in_empty ] ∀ e : E.Elem • ∼ is_in(e, empty),

    [ is_in_add ]
      ∀ s : Set, e, e′ : E.Elem •
        is_in(e′, add(e, s)) ≡ e = e′ ∨ is_in(e′, s),

    [ is_in_remove ]
      ∀ s : Set, e, e′ : E.Elem •
        is_in(e′, remove(e, s)) ≡ e ≠ e′ ∧ is_in(e′, s)
end
```


ARRAY_PARM

```
scheme ARRAY_PARM =
class
  type Elem
  value
    min, max : Int,
    init : Elem
  axiom [ array_not_empty ] max ≥ min
end
```

A_ARRAY

```
scheme A_ARRAY(P : ARRAY_PARM) =
class
  type
    Array,
    Index = {| i : Int • i ≥ P.min ∧ P.max ≥ i |}

  value
    /∗ generators ∗/
    init : Array,
    change : Index × P.Elem × Array → Array,

    /∗ observer ∗/
    apply : Index × Array → P.Elem
  axiom
    [ apply_init ]
    ∀ i : Index • apply(i, init) ≡ P.init,

    [ apply_change ]
      ∀ i, i′ : Index, e : P.Elem, a : Array •
        apply(i′, change(i, e, a)) ≡
          if i = i′ then e else apply(i′, a) end
end
```


Instantiation

- For the POOL we can use *Ship* for *Elem*
- For the BERTHS we can use *Occupancy* for *Elem*, *vacant* for *init*, but we need an integer index as an attribute (static) of *Berth*

We extend TYPES with

```
type
  Index = {| i : Int • i ≥ min ∧ max ≥ i |}
value
  min, max : Int,
  indx : Berth → Index
axiom
  [ index_not_empty ] max ≥ min,

  [ berths_indexable ]
  ∀ b1, b2 : Berth •
    indx(b1) = indx(b2) ⇒ b1 = b2
```


A_HARBOUR1

```
scheme A_HARBOUR1 =
hide P, B in
class
  object
    /∗ pool of waiting ships ∗/
    P : A_SET(T{Ship for Elem}),

    /∗ berths ∗/
    B : A_ARRAY(T{Occupancy for Elem,
                          vacant for init})

  type
    Harbour = P.Set × B.Array
```

value

```
/* generators */
arrives : T.Ship × Harbour ~→ Harbour
arrives(s, (ws, bs)) ≡
  (P.add(s, ws), bs)
  pre can_arrive(s, (ws, bs)),

docks : T.Ship × T.Berth × Harbour ~→ Harbour
docks(s, b, (ws, bs)) ≡
  (P.remove(s, ws),
   B.change(T.indx(b), T.occupied_by(s), bs))
  pre can_dock(s, b, (ws, bs)),

leaves : T.Ship × T.Berth × Harbour ~→ Harbour
leaves(s, b, (ws, bs)) ≡
  (ws, B.change(T.indx(b), T.vacant, bs))
  pre can_leave(s, b, (ws, bs)),
```



```
/* observers */
waiting : T.Ship × Harbour → Bool
waiting(s, (ws, bs)) ≡ P.is_in(s, ws),

occupancy : T.Berth × Harbour → T.Occupancy
occupancy(b, (ws, bs)) ≡ B.apply(T.indx(b), bs),

is_docked : T.Ship × Harbour → Bool
is_docked(s, (ws, bs)) ≡
  (∃ b : T.Berth •
    B.apply(T.indx(b), bs) = T.occupied_by(s)),
```



```
/* guards */
can_arrive : T.Ship × Harbour → Bool
can_arrive(s, (ws, bs)) ≡
  ∼ P.is_in(s, ws) ∧ ∼ is_docked(s, (ws, bs)),

can_dock : T.Ship × T.Berth × Harbour → Bool
can_dock(s, b, (ws, bs)) ≡
  P.is_in(s, ws) ∧ ∼ is_docked(s, (ws, bs)) ∧
  B.apply(T.indx(b), bs) = T.vacant ∧ T.fits(s, b),

can_leave : T.Ship × T.Berth × Harbour → Bool
can_leave(s, b, (ws, bs)) ≡
  B.apply(T.indx(b), bs) = T.occupied_by(s)
end
```


Validation and verification

Validation :

- Have we taken any more requirements into account?
- If so, are they satisfied?

Verification :

- Justification that
  A_HARBOUR1 $\preceq$ A_HARBOUR0

Finished applicative development?

- All functions explicit
  (though *is_docked* not translatable)
- Standard modules A_SET and A_ARRAY can be ignored

So ready for next step — to imperative style.

I_HARBOUR1

```
scheme I_HARBOUR1 =
hide P, B in
class
  object
    /∗ pool of waiting ships ∗/
    P : I_SET(T{Ship for Elem}),

    /∗ berths ∗/
    B : I_ARRAY(T{Occupancy for Elem,
                  vacant for init})
```



```
  value
    /∗ generators ∗/
    arrives : T.Ship ∼→ write any Unit
    arrives(s) ≡ P.add(s) pre can_arrive(s),

    docks : T.Ship × T.Berth ∼→ write any Unit
    docks(s, b) ≡
      P.remove(s) ; B.change(T.indx(b), T.occupied_by(s))
      pre can_dock(s, b),

    leaves : T.Ship × T.Berth ∼→ write any Unit
    leaves(s, b) ≡
      B.change(T.indx(b), T.vacant)
      pre can_leave(s, b),

    /∗ observers ∗/
    waiting : T.Ship → read any Bool
    waiting(s) ≡ P.is_in(s),

    occupancy : T.Berth → read any T.Occupancy
    occupancy(b) ≡ B.apply(T.indx(b)),

    is_docked : T.Ship → read any Bool
    is_docked(s) ≡
      (∃ b : T.Berth •
        B.apply(T.indx(b)) = T.occupied_by(s)),
```



```
    /∗ guards ∗/
    can_arrive : T.Ship → read any Bool
    can_arrive(s) ≡ ∼ P.is_in(s) ∧ ∼ is_docked(s),

    can_dock : T.Ship × T.Berth → read any Bool
    can_dock(s, b) ≡
      P.is_in(s) ∧ ∼ is_docked(s) ∧
      B.apply(T.indx(b)) = T.vacant ∧ T.fits(s, b),

    can_leave : T.Ship × T.Berth → read any Bool
    can_leave(s, b) ≡
      B.apply(T.indx(b)) = T.occupied_by(s)
end
```

Validation and Verification

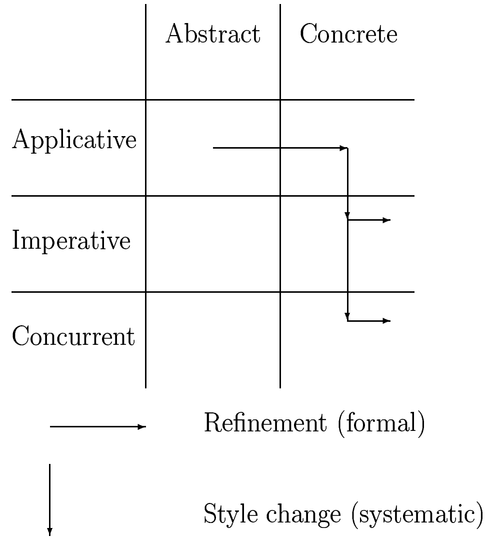
Validation :

- Have we taken any more requirements into account?
- If so, are they satisfied?

Verification :

Idea of method is that this is not done; we have no abstract imperative version for which to show implementation. Instead we argue for "correctness by construction".

Typical Development



Only non-translatable function is *is_docked*. Develop to I_HARBOUR2 with *is_docked* defined by

```
value
  is_docked : T.Ship → read any Bool
  is_docked(s) ≡
    local
      variable
        found : Bool := false,
        indx : Int := T.min
    in
      while ∼ found ∧ indx ≤ T.max do
        found := B.apply(indx) = T.occupied_by(s) ;
        indx := indx + 1
      end ;
      found
    end
```

Verify implementation by showing this satisfies the definition in I_HARBOUR1.

Completion

- translation of I_HARBOUR2
- translation (if necessary) of standard modules I_SET and I_ARRAY
- unit testing
- installation
- testing