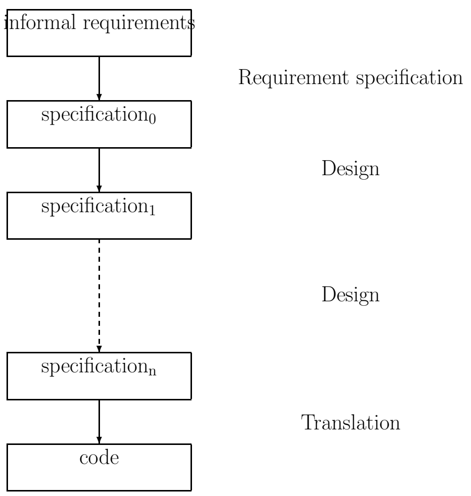
The RAISE Method
Contents

A Survey of the RAISE Method

RAISE Method Characteristics

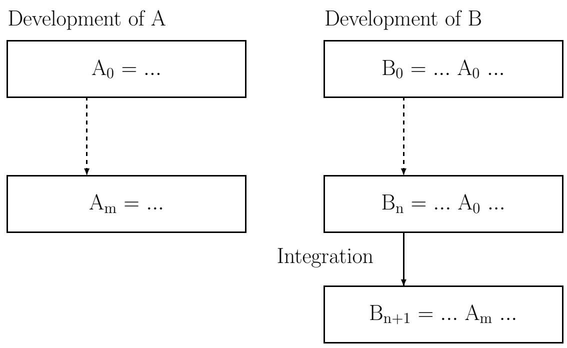
- stepwise development
- separate development
- invent and verify
- rigorous verification

Stepwise Development



specification$_0$ might build on a domain specification

Separate Development



Initial Specification

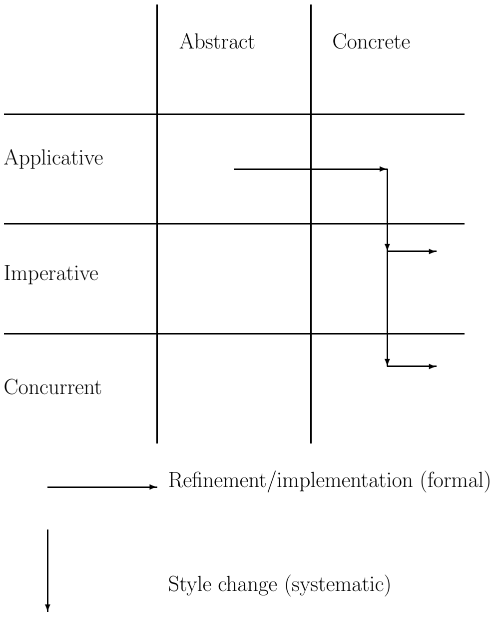
Should

- be modular (allows separate development)
- be as general as possible
- be abstract rather than detailed using sorts and important axioms and hiding what does not need to be exported

Design

- removing underspecification
  - replacing abstract types with concrete types
  - replacing signatures and axioms with explicit value definitions
- changing style
  - from applicative to imperative
  - from sequential to concurrent

Typical Development



Translation

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

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

Invent and Verify

$$\begin{tikzcd} \boxed{\text{specification}_1} \arrow[d, "\text{development relation}"] \\ \boxed{\text{specification}_2} \end{tikzcd}$$

Possible development relations:

- implementation relation
- user-defined relation

Implementation Relation

$$\begin{tikzcd} \boxed{\text{class\_expr}_1} \arrow[d, "\text{implemented by}"] \\ \boxed{\text{class\_expr}_2} \end{tikzcd}$$

class_expr_2 implements class_expr_1

$\text{class\_expr}_2 \preceq \text{class\_expr}_1$

iff

1. class_expr_2 *statically implements* class_expr_1, i.e.: class_expr_2 defines at least all the names defined in class_expr_1 with same kinds and maximal types (statically decidable)
2. theory(class_expr_1) holds in class_expr_2 ($\Rightarrow$ implementation conditions)

theory(class_expr_1) is the collection of (explicit and implicit) axioms in class_expr_1.

Example: abstract specification

```
scheme REGISTRATION0 =
  class
    type
      Register,
      Name
    value
      enrol : Name × Register → Register,
      leave : Name × Register → Register,
      registered : Name × Register → Bool
    axiom
      [reg_enrol]
        ∀ n, n1 : Name, r : Register •
          registered(n, enrol(n1, r)) ≡
            n = n1 ∨ registered(n, r),
      [reg_leave]
        ∀ n, n1 : Name, r : Register •
          registered(n, leave(n1, r)) ≡
            n ≠ n1 ∧ registered(n, r)
  end
```


Example: Concrete Specification

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

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

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

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

Example of a Development Relation

development_relation [REGISTRATION0_1]
REGISTRATION1 $\preceq$ REGISTRATION0

Static implementation can be checked by a tool.

Implementation conditions generated by a tool:

$\bullet \llcorner \forall$ n, n1 : Name, r : register $\bullet$
registered(n, enrol(n1, r)) $\equiv$
n = n1 $\vee$ registered(n, r)$\lrcorner$

$\bullet \llcorner \forall$ n, n1 : Name, r : register $\bullet$
registered(n, leave(n1, r)) $\equiv$
n $\neq$ n1 $\wedge$ registered(n, r)$\lrcorner$

These should be verified in the context of REGISTRATION1.

Rigorous Verification

justification = justification obligation + argument

Justification obligations:

- development relations between modules
- theorems about modules
- confidence conditions

Arguments are mixture of:

- formal arguments (proof rule applications)
- informal arguments

Examples of Proof Rules

Always available:

[isin_union]
e $\in$ (es1 $\cup$ es2) $\simeq$ e $\in$ es1 $\vee$ e $\in$ es2 **when** ...
[isin_singleton]
e $\in$ {e′} $\simeq$ e = e′
[is_annihilation]
e $\equiv$ e $\simeq$ **true**

Available in the context of REGISTRATION1:

[enrol_def] enrol(n,r) $\simeq$ {n} $\cup$ r **when** ...
[leave_def] leave(n,r) $\simeq$ r \ {n} **when** ...
[registered_def] registered(n,r) $\simeq$ n $\in$ r **when** ...

Example of a Justification

$\llcorner \forall$ n, n1 : Name, r : Register $\bullet$
registered(n, enrol(n1,r)) $\equiv$ n = n1 $\vee$ registered(n, r)$\lrcorner$
all_assumption_inf:
$\llcorner$registered(n, enrol(n1,r)) $\equiv$ n = n1 $\vee$ registered(n, r)$\lrcorner$
enrol_def:
$\llcorner$registered(n,{n1} $\cup$ r) $\equiv$ n = n1 $\vee$ registered(n, r)$\lrcorner$
registered_def:
$\llcorner$n $\in$ ({n1} $\cup$ r) $\equiv$ n = n1 $\vee$ registered(n, r)$\lrcorner$
registered_def:
$\llcorner$n $\in$ ({n1} $\cup$ r) $\equiv$ n = n1 $\vee$ n $\in$ r$\lrcorner$
isin_union:
$\llcorner$n $\in$ {n1} $\vee$ n $\in$ r $\equiv$ n = n1 $\vee$ n $\in$ r$\lrcorner$
isin_singleton:
$\llcorner$n = n1 $\vee$ n $\in$ r $\equiv$ n = n1 $\vee$ n $\in$ r$\lrcorner$
is_annihilation:
$\llcorner$**true**$\lrcorner$
qed

Implementation Relation

1. definition
2. use
3. proving static implementation using maximal signatures
4. proving (dynamic) implementation
5. compositionality properties

Implementation Relation

$$\begin{array}{c}\boxed{\text{class\_expr}_1}\\ \Big\downarrow \text{ implemented by}\\ \boxed{\text{class\_expr}_2}\end{array}$$

class_expr_2 implements class_expr_1

$\text{class\_expr}_2 \preceq \text{class\_expr}_1$

iff

1. class_expr_2 *statically implements* class_expr_1, i.e.: class_expr_2 defines at least all the names defined in class_expr_1 with same kinds and maximal types (statically decidable)
2. theory(class_expr_1) holds in class_expr_2 ($\Rightarrow$ implementation conditions)

theory(class_expr_1) is the collection of (explicit and implicit) axioms in class_expr_1.

Use of Implementation Relation

1. In development relations
   (a) between unparameterized schemes, S1 and S2

   S2 $\preceq$ S1

   (b) between parameterized schemes, S1 and S2

   in class object A : PAR **end** $\vdash$ S2(A) $\preceq$ S1(A)

   where PAR is the parameter requirement
2. Between formal and actual scheme parameters

   scheme ELEM = ...
   scheme S(X : ELEM) = ...
   scheme ACTUAL
   object A : ACTUAL

   Confidence condition for S(A): ACTUAL $\preceq$ ELEM

Showing implementation

1. max-sig(class_expr_1) $\subseteq$ max-sig(class_expr_2)
2. $\text{class_expr}_2 \vdash$theory($\text{class\_expr}_1$)

assuming no hiding (and local scheme defs) in class_expr1.

Maximal Signature

max−sig(**class** d1 ... dn **end**) ≃
  class max−sig(d1) ... max−sig(dn) **end**

max−sig(**type** id) ≃ **type** id

max−sig(**type** id = te) ≃ **type** id = max(te)

max−sig(**value** v : te) ≃ **value** v : max(te)

max−sig(**variable** v : te := ve) ≃ **variable** v : max(te)

max−sig(**channel** c : te) ≃ **channel** c : max(te)

max−sig(**object** O : ce) ≃ **object** O : max−sig(ce)

max−sig(**axiom** ax) ≃

Example

max−sig(
  class type T = **Nat value** v : T **axiom** v > 3 **end**
)
≃
  class type T = **Int value** v : **Int end**

Theory

theory(**class** d1 ... dn **end**) ≃
  theory(d1) ∧ ... ∧ theory(dn)

theory(**type** id) ≃ **true**

theory(**type** id = te) ≃ { x | x : id } = { x | x : te }

theory(**value** id : tmax) ≃ **true**

theory(**value** id : {| x : tmax • p(x) |}) ≃ p(id) ≡ **true**

...

theory(**axiom** ax) ≃ □ ax ≡ **true**

Example

theory(
  class type T = **Nat value** v : T **axiom** v > 3 **end**
)
≃
  { x | x : T } = { x | x : **Nat** } ∧ v ≥ 0 ∧ v > 3

Generation of Implementation Conditions

$\llcorner$ce2 ⪯ ce1$\lrcorner$
implementation_relation_expansion_inf :
- $\llcorner$p1$\lrcorner$
...
- $\llcorner$pn$\lrcorner$

where

theory(ce1) = p1 ∧ ... ∧ pn

If ce2 ⊢ pi is statically decidable, it is not generated.

Implementation Examples

S1 = **class value** x, y : **Int end**

S2 = **class value** x, y : **Int axiom** x > y **end**

S3 = **class value** x : **Int** = 1, y : **Int** = 0 **end**

S4 = **class value** x, y, z : **Int end**

S5 =
  class
    value x, y, z : **Int axiom** x > z ∧ z > y
  end

S3 ⪯ S2 ⪯ S1

S5 ⪯ S4 ⪯ S1

S5 ⪯ S2

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Implementation of type definitions

type id $\preceq$ **type** id

type id = te $\preceq$ **type** id

$$\frac{\mathrm{te2} \equiv \mathrm{te1}}{\textbf{type}\ \mathrm{id} = \mathrm{te2} \preceq \textbf{type}\ \mathrm{id} = \mathrm{te1}}$$

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Implementation of value definitions

There are a number of rules, e.g.

$$\frac{\mathrm{te2} \preceq \mathrm{te1}}{\textbf{value}\ \mathrm{id} : \mathrm{te2}\ \preceq\ \textbf{value}\ \mathrm{id} : \mathrm{te1}}$$

This is not the only possibility.

Example:

S1 = **class value** x : **Nat end**
S2 = **class value** x : **Int axiom** x = 2 **end**

S2 $\preceq$ S1

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Implementation of variable definitions

$$\frac{\mathrm{te2} \equiv \mathrm{te1}}{\textbf{variable}\ \mathrm{id} : \mathrm{te2}\ \preceq\ \textbf{variable}\ \mathrm{id} : \mathrm{te1}}$$

$$\frac{\mathrm{te2} \equiv \mathrm{te1}}{\textbf{variable}\ \mathrm{id} : \mathrm{te2} := \mathrm{e2}\ \preceq\ \textbf{variable}\ \mathrm{id} : \mathrm{te1}}$$

$$\frac{\mathrm{te2} \equiv \mathrm{te1},\ \mathrm{e2} \equiv \mathrm{e1}}{\textbf{variable}\ \mathrm{id} : \mathrm{te2} := \mathrm{e2}\ \preceq\ \textbf{variable}\ \mathrm{id} : \mathrm{te1} := \mathrm{e1}}$$

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Compositionality of Implementation Relation

- transitivity
  (allows for stepwise development)
- substitutivity
  (allows for separate development)

Requirements to Substitutions I

The result of a substitution must be well-formed.

Example

S1 = **class value** x : **Int end**
S2 = **class value** x, y : **Int end**

Substituting S1 with S2 in

extend S1 **with class value** y : **Int end**

gives an ill-formed class expression:

extend S2 **with class value** y : **Int end**

Requirements to Substitutions II

ce2 $\preceq$ ce1 $\Rightarrow$ F(ce2) $\preceq$ F(ce1) ?

Class expressions of local modules must only contain free names of parameter modules and global modules.

Example

Substitution of class(O) with

class value x : **Int** = y, y : **Int end**

in

class
  object O : **class value** x : **Int** = y **end**
  value y : **Int**
end

gives

class
  object O : **class value** x : **Int** = y, y : **Int end**
  value y : **Int**
end

which is not an implementation.

Development in the Large

Substitution with Implementations

Assume ce2 $\preceq$ ce1

Then

hide names **in** ce2 $\preceq$ **hide** names **in** ce1

use renamings **in** ce2 $\preceq$ **use** renamings **in** ce1

extend ce2 **with** ce $\preceq$ **extend** ce1 **with** ce

extend ce **with** ce2 $\preceq$ **extend** ce **with** ce1

class object O : ce2 **end** $\preceq$ **class object** O : ce1 **end**

Development in the Large

Substitution with Implementations

Development of unparameterized schemes:

Given

scheme S1 = ce1, S2 = ce2

If ce2 $\preceq$ ce1 then

S2 $\preceq$ S1

Development of parameterized schemes:

Given

scheme S1(X : ceX) = ce1, S2(X : ceX) = ce2
object A : ceA

If

ceA $\preceq$ ceX   and   **object** X : ceX $\vdash$ ce2 $\preceq$ ce1

then

S2(A) $\preceq$ S1(A)

Verification

- What to verify (justification obligations)
- How to verify (arguments)
- Proof rules and their application
- Available proof rules
- Informal arguments

Verification and Justifications

Justification obligations

A *justification obligation* is a logical expression that should be verified to be true. Kinds of justification conditions:

- development relations between modules, e.g.
  REGISTRATION1 $\preceq$ REGISTRATION0
- theorems about modules, e.g.
  REGISTRATION1 $\vdash$
  $\forall$ n : Name • registered(n, {}) = **false**

Justifications

A *justification* is a piece of syntax, consisting of
a justification obligation + an argument for its truth.

The RAISE tools comprise a justification editor.

Arguments

are mixtures of:

- formal arguments (proof rule applications)
- informal arguments

Verification

verification is the act of making a justification.

Example of a Justification

(Foil 16 once again.)

$\llcorner\forall$ n, n1 : Name, r : Register •
registered(n, enrol(n1,r)) $\equiv$ n = n1 $\vee$ registered(n, r)$\lrcorner$
all_assumption_inf:
$\llcorner$registered(n, enrol(n1,r)) $\equiv$ n = n1 $\vee$ registered(n, r)$\lrcorner$
enrol_def:
$\llcorner$registered(n,{n1} $\cup$ r) $\equiv$ n = n1 $\vee$ registered(n, r)$\lrcorner$
registered_def:
$\llcorner$n $\in$ ({n1} $\cup$ r) $\equiv$ n = n1 $\vee$ registered(n, r)$\lrcorner$
registered_def:
$\llcorner$n $\in$ ({n1} $\cup$ r) $\equiv$ n = n1 $\vee$ n $\in$ r$\lrcorner$
isin_union:
$\llcorner$n $\in$ {n1} $\vee$ n $\in$ r $\equiv$ n = n1 $\vee$ n $\in$ r$\lrcorner$
isin_singleton:
$\llcorner$n = n1 $\vee$ n $\in$ r $\equiv$ n = n1 $\vee$ n $\in$ r$\lrcorner$
is_annihilation:
$\llcorner$**true**$\lrcorner$
qed

The context of this justification is REGISTRATION1.

Equivalence Rules

Simple form:

[name_of_rule]
term1 $\simeq$ term2

Examples:

[isin_singleton]
e $\in$ {e′} $\simeq$ e = e′
[is_annihilation]
e $\equiv$ e $\simeq$ **true**

state that two terms are equivalent

$\simeq$ is a generalisation of $\equiv$

Application of Equivalence Rules

can be applied in either direction
can be applied to subterms

⌞condition⌟
name_of_rule:
⌞resulting_condition⌟

Examples: see page 35

Equivalence Rules with Applicability Conditions

Form:

[name_of_rule]
term1 $\simeq$ term2 **when** condition

Application:

⌞condition⌟
name_of_rule:
⌞resulting_condition⌟
since
⌞side_condition⌟
argument_for_side_condition
end
argument_for_resulting_condition

Equivalence Rules with Applicability Conditions

Example

[equality_annihilation]
e = e $\simeq$ **true**
when convergent(e) $\wedge$ **readonly**(e)

Application:

⌞(5 = 5) $\Rightarrow$ ... ⌟
equality_annihilation:
⌞(**true**) $\Rightarrow$ ... ⌟
since
⌞ **convergent**(5) $\wedge$ **readonly**(5)⌟
<:*argument*:>
end
<:*argument*:>

Inference Rules

example

[and_split_inf]
$\dfrac{\text{ro\_eb, ro\_eb}'}{\text{ro\_eb} \wedge \text{ro\_eb}'}$

Application:

⌞x $\neq$ y $\wedge$ x = z⌟
and_split_inf:
- ⌞x $\neq$ y⌟
  <:*argument*:>
- ⌞x = z⌟
  <:*argument*:>

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Inference Rules

example

$$\frac{[\,\text{imply\_deduction\_inf}\,] \quad [\,\text{id}\,]\ \text{ro\_eb} \vdash \text{ro\_eb}'}{\text{ro\_eb} \Rightarrow \text{ro\_eb}'}$$

when convergent(ro_eb) $\wedge$ **pure**(ro_eb)

Application:

⌞x = 0 ⇒ x ≥ 0⌟
imply_deduction_inf :
[x_zero] x = 0 ⊢ ⌞x ≥ 0⌟
x_zero :
⌞0 ≥ 0⌟
<:*argument*:>

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Available Proof Rules

1. rule base (always available)
2. context rules (depend on context)

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Context Rules

Value definitions

Context:

value max : Int = 3

gives rise to the context rule

[max_def] max ≃ 3

Example of application:

⌞ ... max ... ⌟
max_def :
⌞ ... 3 ... ⌟

In the tools:

⌞ ... max ... ⌟
value_name_unfold :
⌞ ... 3 ... ⌟

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Context Rules

Function definitions without pre conditions

Context:

value
f : **Nat** → **Nat**
f(x) ≡ x + 1

gives rise to the context rule

[f_def] f(e) ≃ e + 1
when convergent(e) ∧ **pure**(e) ∧ **isin_subtype**(e, **Nat**)

Example of application:

⌞ ... f(7) ... ⌟
f_def :
⌞ ... 7 + 1 ... ⌟

In the tools:

⌞ ... f(7) ... ⌟
application_expr_unfold1 :
⌞ ... 7 + 1 ... ⌟

Context Rules
Function definitions with pre conditions

Context:

value
  f : **Int** $\xrightarrow{\sim}$ **Int**
  f(x) ≡ x + 1
  pre x > 0

gives rise to the context rule

[f_def] f(e) ≃ e + 1
when convergent(e) ∧ **pure**(e) ∧ e > 0

Example of application:

⌞ ... f(7) ... ⌟
f_def :
⌞ ... 7 + 1 ... ⌟
  since
    ⌞7 > 0⌟

  end

In the tools the rule application_expr_unfold2 is used instead of f_def.

Context Rules

Available in the context of REGISTRATION1:

[enrol_def] enrol(n,r) ≃ {n} ∪ r
when convergent(n, r) ∧ **pure**(n) ∧ **pure**(r) ∧
  isin_subtype(n, Name) ∧ **isin_subtype**(r, Register)

[leave_def] leave(n,r) ≃ r \ {n}
when ...

[registered_def] registered(n,r) ≃ n ∈ r
when ...

Informal Arguments

Explanation argument:

⌞⟨5⟩ ⌢ ⟨7⟩ ≡ ⟨5, 7⟩⌟
/∗ which is obviously true ∗/
qed

Replacement argument:

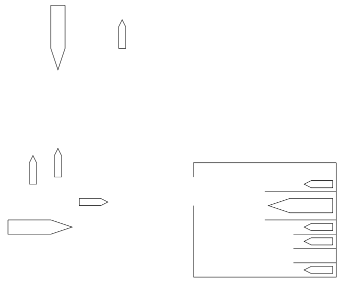
⌞7 ∈ {(⟨5,7,3⟩ ⌢ ⟨⟩)(2)}⌟
/∗ as ..., this is equivalent to ∗/
⌞7 ∈ {7}⌟

Commented Arguments

⌞x ≠ y ∧ x = z⌟
/∗ next we want to bring the left conjunct
  into an appropriate form ∗/
inequality_expansion:
⌞∼(x = y) ∧ x = z⌟

A case study: harbour

Example: a Harbour Master



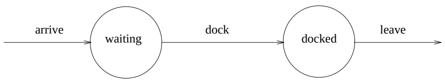
to control movement ships:

arrive
dock
leave

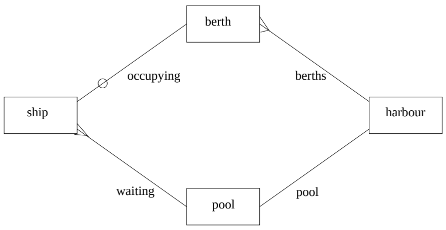
Some requirements:

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 docked in berths they fit.

State Transition Diagram



Entity Relationships



Initial Specification

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

object T : TYPES

scheme A_HARBOUR0 = ...
```

Abstract Applicative Modules

1. Abstract type of interest T (**type** T)
2. Observers with range types not dependent on T (**value** obs : ... T → T2)
3. Constant(s) of types dependent on T (**value** c : T)
4. Generators with range types dependent on T (**value** gen : ... → T)
5. Guards for partial generators (**value** can_gen : ... → Bool)
6. Derived functions: defined from others
7. Axioms:
   (a) Observer-constant axioms
   (b) Observer-generator axioms
   (∀ ...• obs(..., gen(...)) ≡ ...)
   (c) Generator-result axioms
   (d) System axioms: preservation of invariants etc.

A_HARBOUR0

```
type
  /∗ type of interest ∗/
  Harbour

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

  /∗ derived observer ∗/
  is_docked : T.Ship × Harbour → Bool
  is_docked(s, h) ≡
    (∃ b : T.Berth • occupancy(b, h) = T.occupied_by(s))
```


Invariants

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

invariant no. 1 is ensured by:

value occupancy : T.Berth × Harbour → T.Occupancy

A_HARBOUR0

```
value
  /∗ invariant ∗/
  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))
    )
```

A_HARBOUR0

```
value
  /∗ generators ∗/
  arrives : T.Ship × Harbour ~→ Harbour,
  docks : T.Ship × T.Berth × Harbour ~→ Harbour,
  leaves : T.Ship × T.Berth × Harbour ~→ Harbour,

  /∗ guards to express preconditions ∗/
  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)
```

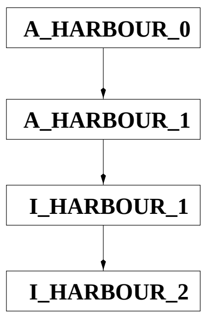

A_HARBOUR0

```
axiom
/∗ observer−generator axioms ∗/
  [waiting_arrives]
    ∀ h : Harbour, s1, s2 : T.Ship •
      waiting(s2, arrives(s1, h)) ≡
      s1 = s2 ∨ waiting(s2, h)
        pre can_arrive(s1, h),

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

  [occupancy_docks] ...,

  [waiting_leaves] ...,

  [occupancy_leaves] ...
```


A_HARBOUR0

```
axiom
  /∗ invariant preservation by generators ∗/
  [consistent_arrives]
    ∀ h : Harbour, s : T.Ship •
      arrives(s, h) as h′ post consistent(h′)
        pre consistent(h) ∧ can_arrive(s, h),
  [consistent_docks]
    ∀ 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),
  [consistent_leaves]
    ∀ 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)
```


Development Plan



Development to Concrete Applicative

1. Replace sort definition with concrete type definition.
2. Give explicit definitions of constants and functions.
3. Remove axioms.

Extension of TYPES

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 =
  class
    type Harbour =
        T.Ship-**set** ×
        {| bs : T.Index $\overrightarrow{m}$ T.Occupancy •
            (∀ idx : T.Index • idx ∈ **dom** bs) |}

    ... (see the following foils)
  end

A_HARBOUR1

value
  /* observers */
  waiting : T.Ship × Harbour → **Bool**
  waiting(s, (ws, bs)) ≡ s ∈ ws,

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

A_HARBOUR1

```
/∗ generators ∗/
arrives : T.Ship × Harbour ∼→ Harbour
arrives(s, (ws, bs)) ≡
  (ws ∪ {s}, bs)
  pre can_arrive(s, (ws, bs)),

docks : T.Ship × T.Berth × Harbour ∼→ Harbour
docks(s, b, (ws, bs)) ≡
  (ws \ {s}, bs † [ T.indx(b) ↦ T.occupied_by(s) ])
  pre can_dock(s, b, (ws, bs)),

leaves : T.Ship × T.Berth × Harbour ∼→ Harbour
leaves(s, b, (ws, bs)) ≡
  (ws, bs † [ T.indx(b) ↦ T.vacant ])
   pre can_leave(s, b, (ws, bs)),
```


A_HARBOUR1

```
/∗ derived observer ∗/
is_docked : T.Ship × Harbour → Bool
is_docked(s, (ws, bs)) ≡ ...

/∗ invariant ∗/
consistent : Harbour → Bool
consistent((ws, bs)) ≡ ...

/∗ guards to express preconditions ∗/
...
```


A_HARBOUR0_1

```
development_relation [ A_HARBOUR0_1 ]
  A_HARBOUR1 ⪯ A_HARBOUR0
```


Part of proof

```
  ⌞∀ h : Harbour, s1, s2 : T.Ship •
    waiting(s2, arrives(s1, h)) ≡ s1 = s2 ∨ waiting(s2, h)
      pre can_arrive(s1, h)⌟
all_name_change :
  ⌞∀ (ws, bs) : Harbour, s1, s2 : T.Ship •
    waiting(s2, arrives(s1, (ws, bs))) ≡ s1 = s2 ∨ waiting(s2, (ws, bs))
      pre can_arrive(s1, (ws, bs))⌟
all_assumption_inf :
  ⌞waiting(s2, arrives(s1, (ws, bs))) ≡ s1 = s2 ∨ waiting(s2, (ws, bs))
    pre can_arrive(s1, (ws, bs))⌟
pre_deduction_inf :
[ assump ] can_arrive(s1, (ws, bs)) ⊢
  ⌞waiting(s2, arrives(s1, (ws, bs))) ≡ s1 = s2 ∨ waiting(s2, (ws, bs))⌟
arrives_def :
  ⌞waiting(s2, (ws ∪ {s1}, bs)) ≡ s1 = s2 ∨ waiting(s2, (ws, bs))⌟
  since
      ⌞can_arrive(s1, (ws, bs))⌟
    assump :
      ⌞true⌟
    qed
  end
```

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

waiting_def :
  ⌞s2 ∈ ws ∪ {s1} ≡ s1 = s2 ∨ waiting(s2, (ws, bs))⌟
waiting_def :
  ⌞s2 ∈ ws ∪ {s1} ≡ s1 = s2 ∨ s2 ∈ ws⌟
isin_union :
  ⌞s2 ∈ ws ∨ s2 ∈ {s1} ≡ s1 = s2 ∨ s2 ∈ ws⌟
isin_singleton :
  ⌞s2 ∈ ws ∨ s2 = s1 ≡ s1 = s2 ∨ s2 ∈ ws⌟
or_commutativity :
  ⌞s2 = s1 ∨ s2 ∈ ws ≡ s1 = s2 ∨ s2 ∈ ws⌟
equality_commutativity :
  ⌞s1 = s2 ∨ s2 ∈ ws ≡ s1 = s2 ∨ s2 ∈ ws⌟
is_annihilation :
  ⌞**true**⌟
qed

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

I_HARBOUR1

I_HARBOUR1 is as A_HARBOUR1 except:

- variables are introduced

  variable
    ws : T.Ship-**set**,
    bs : {| bs : T.Index $\overrightarrow{m}$ T.Occupancy •
        (∀ idx : T.Index • idx ∈ **dom** bs) |}

- type Harbour is removed
- Function signatures:
  Harbour removed and **read/write any** added
- Function bodies changed accordingly

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Examples:

  waiting : T.Ship × Harbour → **Bool**
  waiting(s, (ws, bs)) ≡ s ∈ ws,

  arrives : T.Ship × Harbour $\xrightarrow{\sim}$ Harbour
  arrives(s, (ws, bs)) ≡ (ws ∪ {s}, bs)
    pre can_arrive(s, (ws, bs))

changed to:

  waiting : T.Ship → **read any Bool**
  waiting(s) ≡ s ∈ ws,

  arrives : T.Ship $\xrightarrow{\sim}$ **write any Unit**
  arrives(s) ≡ ws := ws ∪ {s} **pre** can_arrive(s)

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

I_HARBOUR2

I_HARBOUR2 is as I_HARBOUR1 except:

  is_docked : T.Ship → **read any Bool**
  is_docked(s) ≡
    (∃ b : T.Berth • bs(T.indx(b)) = T.occupied_by(s))

is changed to

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

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

I_HARBOUR1_2

development_relation [I_HARBOUR1_2]
I_HARBOUR2 $\preceq$ I_HARBOUR1

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Further development of TYPES

- concrete types for Ship, Berth
- explicit definition of fits, min, max and indx

left as an exercise for you!