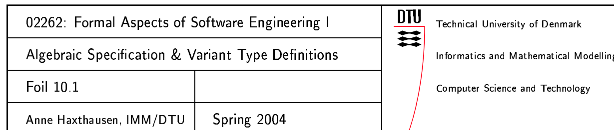
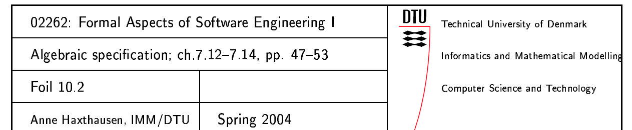


Algebraic Specification & Variant Type Definitions

Contents

- Model-oriented vs. property-oriented 2
- Examples of algebraic specification 3
- Semantics of RSL specifications 6
- Pitfalls of specifications 9
- Variant type definitions 12



Specification styles
Model-oriented versus Property-oriented

Characterization:

- **algebraic/property-oriented**:
  (main) types are *abstract*,
  typically declared as *sorts* and implicitly specified by declared values and axioms.

  Example: type Database

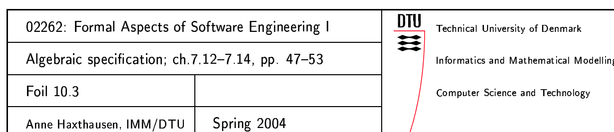
- **model-oriented**:
  (main) types are *concrete* types that are constructed explicitly, typically from basic types and type constructors in *abbreviation* definitions.

  Example: type Database = Person-**set**

Pragmatics:

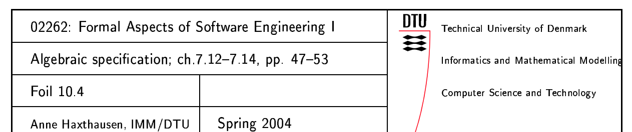
- **algebraic/property-oriented**:
  normally used in early development phases
- **model-oriented**:
  normally used in later development phases

This lecture concerns algebraic specifications and some short-hands (variant type definitions) for these.



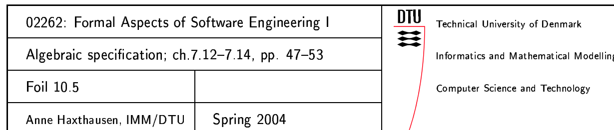
Algebraic Specification: Example 1

```
scheme DATABASE =
  class
    type
      Database, Key, Data
    value
      empty : Database,
      insert : Key × Data × Database → Database,
      remove : Key × Database → Database,
      defined : Key × Database → Bool,
      lookup : Key × Database ∼→ Data
```

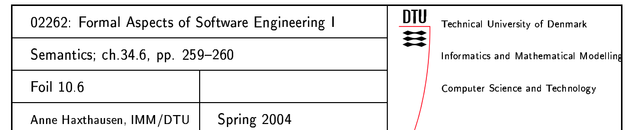
Example 1 contd.

```
    axiom
      [remove_empty]
        ∀ k : Key • remove(k,empty) ≡ empty,
      [remove_insert] ∀ k, k1 : Key, d : Data, db : Database •
        remove(k,insert(k1,d,db)) ≡
          if k = k1 then remove(k,db)
          else insert(k1,d,remove(k,db)) end,
      [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),
      [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)
      ...
  end
```



Algebraic Specification: Example 2

scheme COLOUR =
class
  type
    Colour
  value
    black : Colour,
    white : Colour
  axiom
    [disjoint]
      black $\neq$ white
    [gen]
      $\forall$ c : Colour • (c = black) $\vee$ (c = white)
end



Semantics of RSL Specifications

Semantics: syntactic objects are mapped to semantic (mathematical) objects.

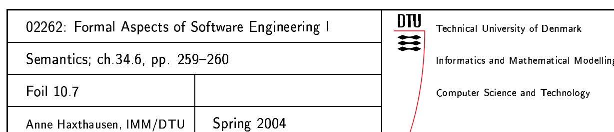
An *algebra* maps

- type names to types (i.e. sets of values)
- value names to values

A *model* of a specification is an algebra that satisfies all definitions and axioms of the specification.

The *semantics* of an RSL specification is the collection of all its models.

In course 02268 FASE II you will learn more about algebraic semantics.



Semantics of RSL Specifications
Example (algebraic)

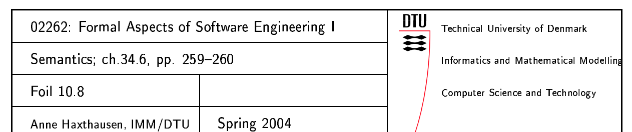
[Colour $\mapsto$ {•, ∘}, black $\mapsto$ •, white $\mapsto$ •]
is an algebra, but not a model for COLOUR.

[Colour $\mapsto$ {•, ∘}, black $\mapsto$ •, white $\mapsto$ ∘]
is a model for COLOUR

Semantics of COLOUR:
{ [Colour $\mapsto$ {•, ∘}, black $\mapsto$ •, white $\mapsto$ ∘]
  ...
}

Are the following algebras models for COLOUR?

[Colour $\mapsto$ {•}, black $\mapsto$ •, white $\mapsto$ •]
[Colour $\mapsto$ {•, ∘, ⊗}, black $\mapsto$ •, white $\mapsto$ ∘]



Semantics of RSL specifications
Example (model-oriented)

scheme S =
  class
    type T = **Nat**
    value k : **Nat**
    axiom k > 2
  end

[T $\mapsto$ {0, 1, 2, ...}, k $\mapsto$ 1]
is an algebra, but not a model for S.

[T $\mapsto$ {0, 1, 2, ...}, k $\mapsto$ 3]
is a model for S

Semantics of S:
{[T $\mapsto$ {0, 1, 2, ...}, k $\mapsto$ 3],
 [T $\mapsto$ {0, 1, 2, ...}, k $\mapsto$ 4],
 ...
}

Pitfalls of Specifications

1. Syntax errors
2. Type errors
3. Inconsistency (i.e. there are no models)
4. Incompleteness (i.e. there are too many models)

Examples of Inconsistency

1. **class** ... **axiom false end**

2. **class**
   value n : **Nat** • n > 5
   axiom n < 3
   end

3. **class**
   value
   f : **Int** → **Int**
   f(x) ≡ **while true do** ... **end** ; x
   end

4. **class**
   value n : **Nat**
   axiom n ≡ **chaos**
   end

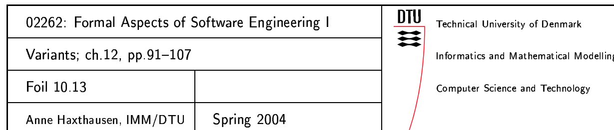
Example of Incompleteness

class
  value
    f : **Int** → **Int**
    f(x) ≡ f(x)
end

Variants

Contents

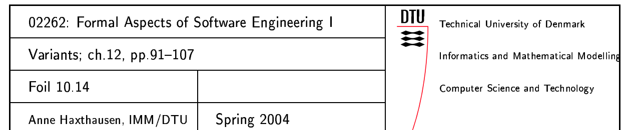
1. Introduction & Motivation 13
2. Constant Constructors 16
3. Record Constructors 20
4. Destructors 31
5. Reconstructors 33
6. Wildcard Constructors 34
7. Examples 40



Variant type definitions

A *variant type definition* is

- similar to ML's datatype declarations
- a convenient short-hand for an algebraic specification



Algebraic Specification

Example: Lists

```
scheme LIST =
  class
    type
      List
    value
      empty : List,
      add : Int × List → List,
      head : List ~→ Int,
      tail : List ~→ List
    axiom
      [head_add] ∀ i : Int, l : List • head(add(i,l)) ≡ i,
      [tail_add] ∀ i : Int, l : List •tail(add(i,l)) ≡ l
  end
```

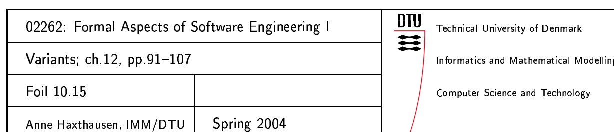
Problem

Have we chosen the right axioms?

For example, is

```
∀ i : Int, l : List • add(i,l) ≠ empty,
∀ l : List • add(1,l) ≠ add(2,l)
```

true in all models for LIST?



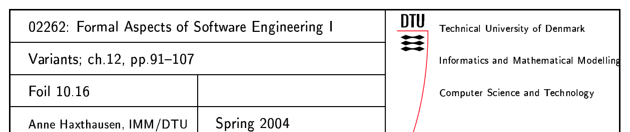
Algebraic Specification

Example: Colour

```
type
  Colour
value
  black : Colour,
  white : Colour
axiom
  [disjoint]
    black ≠ white,
  [gen]
    ∀ c : Colour • (c = black) ∨ (c = white)
```

Problem

Have we chosen the right axioms?



Variant definition with constant constructors

Example: Colour

```
type Colour == black | white
```

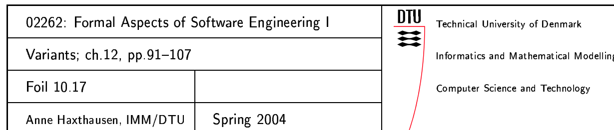
is a shorthand for

```
type
  Colour
value
  black : Colour,
  white : Colour
axiom
  [disjoint]
    black ≠ white,
  [induction]
    ∀ p : Colour → Bool •
     (p(black) ∧ p(white)) ⇒ (∀ c : Colour • p(c))
```

Is similar to ML's:

```
datatype Colour = black | white
```

Is it equivalent to the algebraic specification?

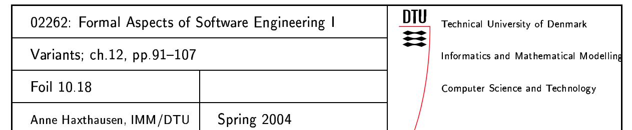


[induction] ⇒ [gen] ?

```
[induction]
  ∀ p : Colour → Bool •
    (p(black) ∧ p(white)) ⇒ (∀ c : Colour • p(c))

[gen]
  ∀ c : Colour • (c = black) ∨ (c = white)
```

Proof:

```
    p = λ c : Colour • (c = black) ∨ (c = white)

  ⌊∀ c : Colour • p(c)⌋
  induction:
    • ⌊p(black)⌋
    • ⌊p(white)⌋
```

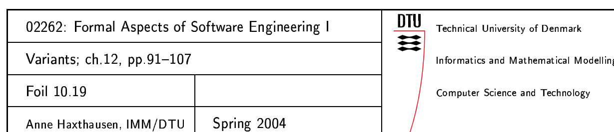



[gen] ⇒ [induction] ?

```
[induction]
  ∀ p : Colour → Bool •
    (p(black) ∧ p(white)) ⇒ (∀ c : Colour • p(c))

[gen]
  ∀ c : Colour • (c = black) ∨ (c = white)
```

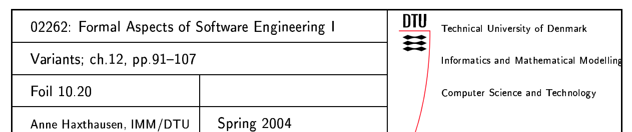
Proof:

```
    p : Colour → Bool,
    p(black),
    p(white),
    c : Colour

   ⌊p(c)⌋
  case
    c = black ⊢ ⌊p(c)⌋
    c = white ⊢ ⌊p(c)⌋
  end
  since
      ⌊(c = black) ∨ (c = white)⌋
    /* cf. [gen] */
  end
```

Colour example, continued

```
type
  Colour == black | white

value
  invert : Colour → Colour
axiom
  invert(black) ≡ white,
  invert(white) ≡ black
```

Variant definition with record constructors

Example: Collections

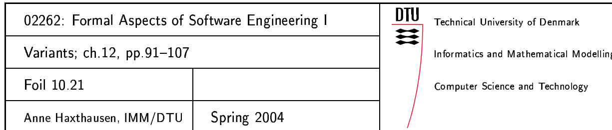
```
type Collection == empty | add(Elem,Collection)
```

is a shorthand for

```
type
  Collection
value
  empty : Collection,
  add : Elem × Collection → Collection
axiom
  [disjoint]
    ∀ e : Elem, c : Collection • empty ≠ add(e,c)
  [induction]
    ∀ p : Collection → Bool •
      (
        p(empty) ∧
        (∀ e : Elem, c : Collection • p(c) ⇒ p(add(e,c)))
      )
      ⇒
      (∀ c : Collection • p(c))
```

Is almost similar to ML's

```
datatype Collection = empty | add of Elem ∗ Collection
```



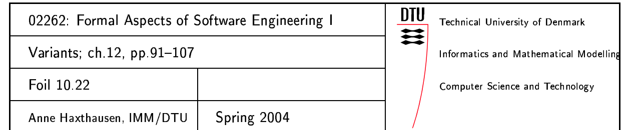
Collection example, continued
Generatedness

Collections are *generated* by the constructors iff

[gen] $\forall$ c : Collection • $\exists$ el : Elem* • c = addn(el)

where

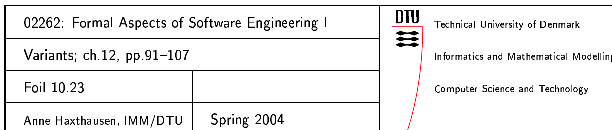
addn : Elem* $\to$ Collection
addn(el) $\equiv$
case el **of**
$\langle\rangle \to$ empty,
$\langle$e$\rangle$ ^ el′ $\to$ add(e,addn(el′))
end



Collection example, continued
[induction] $\Rightarrow$ [gen] ?

p = λ c : Collection • $\exists$ el : Elem* • c = addn(el)

$\llcorner$$\forall$ c : Collection • p(c)$\lrcorner$
induction:
- $\llcorner$p(empty)$\lrcorner$
  /$*$ empty = addn($\langle\rangle$) $*$/
- [I_hyp] p(c) $\vdash$ $\llcorner$p(add(e,c))$\lrcorner$
  /$*$ add(e,c) = addn($\langle$e$\rangle$ ^ el), where addn(el) = c $*$/



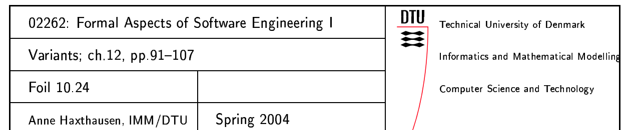
Collection example, continued
[gen] $\Rightarrow$ [induction] ?

Assume
p : Collection $\to$ **Bool**,
[1] p(empty),
[2] ($\forall$ e : Elem, c : Collection • p(c) $\Rightarrow$ p(add(e,c))),
c : Collection

$\llcorner$p(c)$\lrcorner$
according to [gen] we can find el s.t. c = addn(el):
$\llcorner$p(addn(el))$\lrcorner$
according to lemma:
$\llcorner$**true**$\lrcorner$

lemma $\forall$ el : Elem* • p(addn(el))

Proof of lemma by list_induction:
- $\llcorner$p(addn($\langle\rangle$))$\lrcorner$
  unfold addn:
  $\llcorner$p(empty)$\lrcorner$
  /$*$ which follows by assumption [1] $*$/
- [I_hyp] p(addn(el)) $\vdash$
  $\llcorner$p(addn($\langle$e$\rangle$ ^ el))$\lrcorner$
  unfold addn:
  $\llcorner$p(add(e,addn(el)))$\lrcorner$
  /$*$ which follows by assumption [2] (with c = addn(el)) $*$/



Collection example, continued
How many equivalences hold?

Assume $e_1 \neq e_2$.

How many of the following terms are equivalent in all models:

empty
add(e1, empty)
add(e2, empty)
⋮
add(e1(add(e1, empty)))
add(e2(add(e1, empty)))
add(e1(add(e2, empty)))
⋮

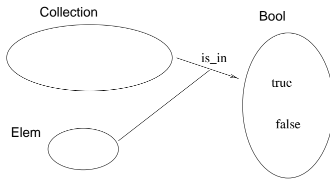
In all models: empty $\neq$ t, where t is not the term empty

In some models: add(e1, empty) = add(e2, empty)
In other models: add(e1, empty) $\neq$ add(e2, empty)

In some models:
add(e1, add(e2, empty)) = add(e2, add(e1, empty))
In other models:
add(e1, add(e2, empty)) $\neq$ add(e2, add(e1, empty))

etc.

Collection example, continued
Adding inequalities by adding observers



value
  is_in : Elem × Collection → **Bool**
axiom
  [is_in_empty]
    ∀ e : Elem • is_in(e,empty) ≡ **false**
  [is_in_add]
    ∀ e,e1 : Elem, c : Collection •
      is_in(e,add(e1,c)) ≡ e = e1 ∨ is_in(e,c)

Assume $e_1 \neq e_2$

is_in(e1,add(e1,empty)) ≠ is_in(e1,add(e2,empty))
⇒
add(e1,empty) ≠ add(e2,empty)

Adding inequalities by adding observers
Generally

Definition:

An *observer* of a sort type T is a function that takes a value of type T as argument and returns a value of another type

value obs : ... T ... → T1

Principle:

If obs(x, t1, y) ≠ obs(x, t2, y) then we can conclude t1 ≠ t2

Collections to denote sets

axiom
  [unordered]
    ∀ e1,e2 : Elem, c : Collection •
      add(e1,add(e2,c)) ≡ add(e2,add(e1,c)),
  [no_duplicates]
    ∀ e : Elem, c : Collection • add(e,add(e,c)) ≡ add(e,c)

alternatively:

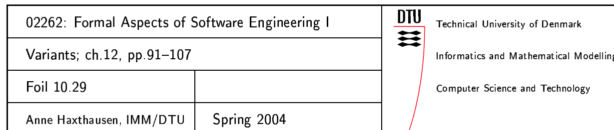
axiom
  [equality]
    ∀ c1, c2 : Collection •
      (c1 = c2) ≡ (∀ e : Elem • is_in(e,c1) = is_in(e,c2))

Exercise

1. Show that [equality] ⇒ [unordered]
2. Show that [equality] ⇒ [no_duplicates]

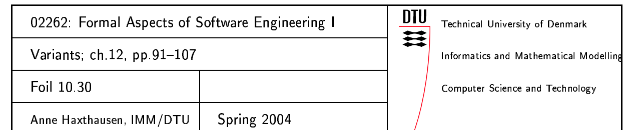
Example: Sets

scheme SET =
  class
    type
      Set == empty | add(Elem,Set),
      Elem
    value
      is_in : Elem × Set → **Bool**
    axiom
      [is_in_empty]
        ∀ e : Elem • is_in(e,empty) ≡ **false**,
      [is_in_add]
        ∀ e,e1 : Elem, s : Set •
          is_in(e,add(e1,s)) ≡ e = e1 ∨ is_in(e,s),
      [equality]
        ∀ s1,s2 : Set •
          (s1 = s2) ≡ (∀ e : Elem • is_in(e,s1) = is_in(e,s2))
  end



Collections to denote lists

type
  Collection == empty | add(Elem,Collection)
value
  head : Collection $\xrightarrow{\sim}$ Elem,
  tail : Collection $\xrightarrow{\sim}$ Collection
axiom
  [head_add]
    $\forall$ e : Elem, c : Collection • head(add(e,c)) $\equiv$ e,
  [tail_add]
    $\forall$ e : Elem, c : Collection • tail(add(e,c)) $\equiv$ c



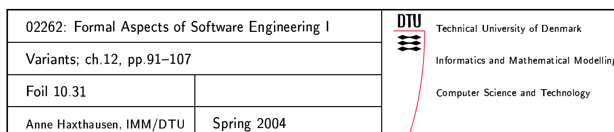
Consequences of head and tail

Assume e1 $\neq$ e2

⌊add(e1, add(e2, empty)) $\neq$ add(e2, add(e1, empty))⌋
observer_property:
⌊head(add(e1, add(e2, empty))) $\neq$
    head(add(e2, add(e1, empty)))⌋
head_add:
⌊e1 $\neq$ e2⌋
by assumption:
⌊**true**⌋

⌊add(e1, add(e1, empty)) $\neq$ add(e1,empty)⌋
observer_property:
⌊tail(add(e1, add(e1, empty))) $\neq$ tail(add(e1, empty))⌋
tail_add:
⌊add(e1, empty) $\neq$ empty⌋
disjointness:
⌊**true**⌋

Note:
never both [no_ordering] and head
never both [no_duplicates] and tail

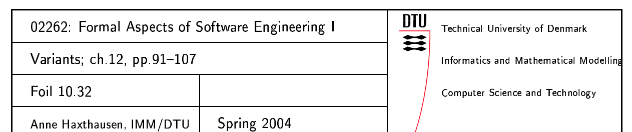


Destructors

type List == empty | add(head : Elem, tail : List)

expands to

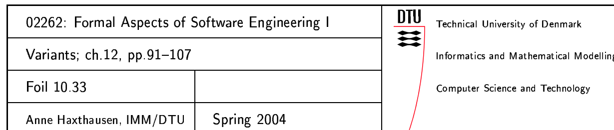
type
  List == empty | add(Elem,List)
value
  head : List $\xrightarrow{\sim}$ Elem,
  tail : List $\xrightarrow{\sim}$ List
axiom
  [head_add]
    $\forall$ e : Elem, l : List • head(add(e,l)) $\equiv$ e,
  [tail_add]
    $\forall$ e : Elem, l : List • tail(add(e,l)) $\equiv$ l



Consequences of destructors

If a constructor *con* has destructors for all its arguments then values produced by applications of this constructor to non equivalent argument values are not equivalent:

t1 $\neq$ t2 $\Rightarrow$ con(t1) $\neq$ con(t2)

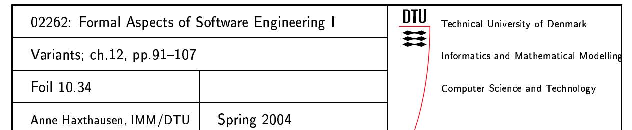


Reconstructors

```
type
  List ==
    empty |
    add(head : Elem ↔ replace_head, tail : List)
```

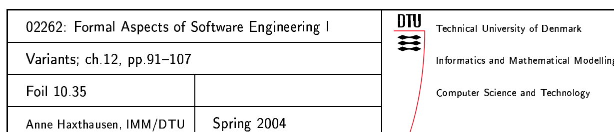
expands to

```
type
  List == empty | add(head : Elem, tail : List)
value
  replace_head : Elem × List ∼→ List
axiom
  [ head_replace_head ]
    ∀ e : Elem, l : List • head(replace_head(e,l)) ≡ e,
  [ tail_replace_head ]
    ∀ e : Elem, l : List • tail(replace_head(e,l)) ≡ tail(l)
```

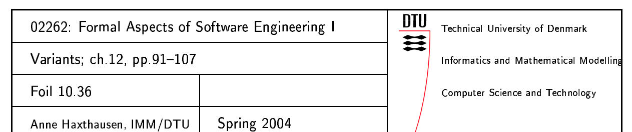
Wildcard Constructors

```
type PosReal = {| r : Real • r > 0.0 |}

type
  Figure ==
    box(length : PosReal, width : PosReal) |
    circle(radius : PosReal)

type
  Figure ==
    box(length : PosReal, width : PosReal) |
    circle(radius : PosReal) |
    _

type
  Figure ==
    box(length : PosReal, width : PosReal) |
    circle(radius : PosReal) |
    triangle(b_line : PosReal, l_angle : Angle, r_angle : Angle),

  Angle = {| r : Real • r > 0.0 ∧ r < 180.0 |}
```

Exercise

Is length, width and radius necessary?



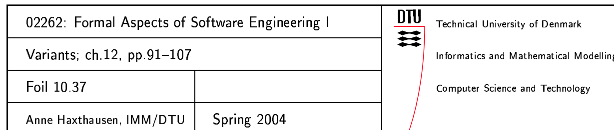
Exercise

Consider:

```
type
  Figure ==
    box(length : PosReal, width : PosReal) |
    circle(radius : PosReal) |
    triangle(b_line : PosReal, l_angle : Angle, r_angle : Angle),

  Angle = {| r : Real • r > 0.0 ∧ r < 180.0 |}
```

Complete the following definition

```
value
  is_wff : Figure → Bool
  is_wff(f) ≡
    case f of
      box(l, w) → ...,
      circle(r) → ...,
      triangle(b, l, r) → ...
    end
```

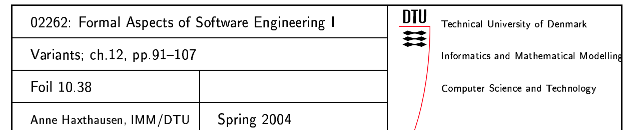


Wildcard Constructors

type
  Figure ==
    box(length : PosReal, width : PosReal) |
    circle(radius : PosReal) |
    triangle(b_line : PosReal, l_angle : Angle, r_angle : Angle)

type
  Figure ==
    box(length : PosReal, width : PosReal) |
    circle(radius : PosReal) |
    triangle(b_line : PosReal, l_line : PosReal, r_line : PosReal)

type
  Figure ==
    box(length : PosReal, width : PosReal) |
    circle(radius : PosReal) |
    _(b_line : PosReal)



Variant Type Definitions

Syntax Summary

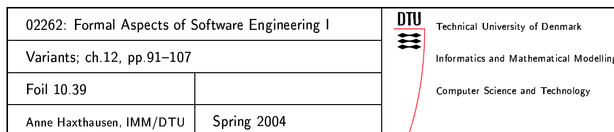
type id == variant_1 | ... | variant_n, $n \geq 1$

alternatives for variant_i:

- a constant constructor of the form:
  id
- a record having a constructor:

  id(
    destructor_1 : type_expr_1 ↔ reconstructor_1,
    ⋮
    destructor_m : type_expr_m ↔ reconstructor_m)
- a record without a constructor:

  _(
    destructor_1 : type_expr_1 ↔ reconstructor_1,
    ⋮
    destructor_m : type_expr_m ↔ reconstructor_m)
- a wildcard:
  _



Variant type definitions

Semantics Summary

Variant type definitions are short-hands for algebraic specifications.

The axioms imply always:

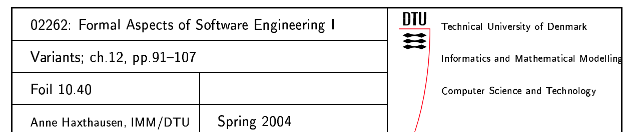
- generatedness by constructors (if no wildcards)
- disjointness of values produced by distinct constructors

If there are destructors present, then also:

- disjointness of values produced by applications of the same constructor to distinct (non equivalent) argument values.

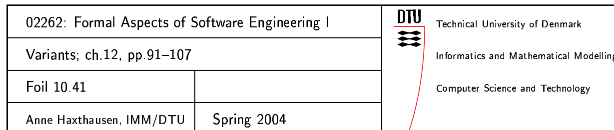
In the latter case the defined type is said to be a *free* type.

Note: Types defined by ML datatype declarations are free although they do not have destructors.

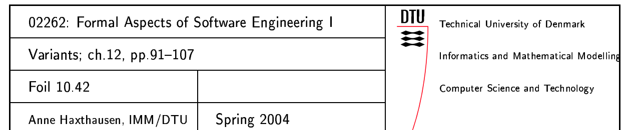


Example: Ordered Trees

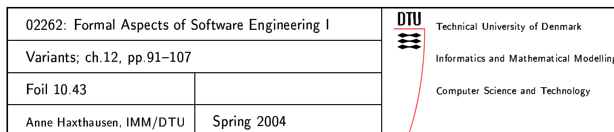
scheme ELEM =
  class
    type Elem
    value
      less_than : Elem × Elem → **Bool**
    axiom
      [anti_reflexive] ∀ e : Elem •
        ∼less_than(e,e),
      [transitive] ∀ e1,e2,e3 : Elem •
        less_than(e1,e2) ∧ less_than(e2,e3) ⇒ less_than(e1,e3),
      [total_order] ∀ e1,e2 : Elem •
        less_than(e1,e2) ∨ less_than(e2,e1) ∨ e1 = e2
  end




```
scheme ORDERED_TREE =
  extend ELEM with
  class
    type
      Tree ==
        empty | node(left : Tree, elem : Elem, right : Tree),
      Ordered_Tree = {| t : Tree • is_ordered(t) |}
    value
      is_ordered : Tree → Bool,
      extract_elems : Tree → Elem-set
    axiom
      [ is_ordered_empty ]
        is_ordered(empty) ≡ true,
      [ is_ordered_node ] ∀ e : Elem, t1,t2 : Tree •
        is_ordered(node(t1,e,t2)) ≡
          (∀ e1 : Elem • e1 ∈ extract_elems(t1) ⇒
              less_than(e1,e)) ∧
          (∀ e2 : Elem • e2 ∈ extract_elems(t2) ⇒
              less_than(e,e2)) ∧
          is_ordered(t1) ∧ is_ordered(t2),
      [ extract_elems_empty ]
        extract_elems(empty) ≡ {},
      [ extract_elems_node ] ∀ e : Elem, t1,t2 : Tree •
        extract_elems(node(t1,e,t2)) ≡
          extract_elems(t1) ∪ {e} ∪ extract_elems(t2)
  end
```


```
scheme SET_FUNCTIONS =
  extend ORDERED_TREE with
  class
    value
      add : Elem × Ordered_Tree → Ordered_Tree,
      is_in : Elem × Ordered_Tree → Bool
    axiom
      [ add_empty ]
        add(e,empty) ≡
          node(empty,e,empty),
      [ add_node ] ∀ e,e0 : Elem, t1,t2 : Ordered_Tree •
        add(e,node(t1,e0,t2)) ≡
          if e = e0 then node(t1,e0,t2)
          elsif less_than(e,e0) then node(add(e,t1),e0,t2)
          else node(t1,e0,add(e,t2))
          end,
      [ is_in_empty ]
        is_in(e,empty) ≡ false,
      [ is_in_node ] ∀ e,e0 : Elem, t1,t2 : Ordered_Tree •
        is_in(e,node(t1,e0,t2)) ≡
          if e = e0 then true
          elsif less_than(e,e0) then is_in(e,t1)
          else is_in(e,t2)
          end
  end
```


```
scheme BALANCED_SET_FUNCTIONS =
  extend SET_FUNCTIONS with class
    type
      Balanced_Tree = {| t : Ordered_Tree • is_balanced(t) |}
    value
      is_balanced : Ordered_Tree → Bool,
      depth : Ordered_Tree → Nat,
      add_balanced :
        Elem × Balanced_Tree → Balanced_Tree
    axiom
      [ is_balanced_empty ]
        is_balanced(empty) ≡ true,
      [ is_balanced_node ]
        ∀ e : Elem, t1,t2 : Ordered_Tree •
          is_balanced(node(t1,e,t2)) ≡
            abs(depth(t1) − depth(t2)) ≤ 1 ∧
            is_balanced(t1) ∧ is_balanced(t2),
      [ depth_empty ] depth(empty) ≡ 0,
      [ depth_node ] ∀ e : Elem, t1,t2 : Ordered_Tree •
        depth(node(t1,e,t2)) ≡
          1 + if depth(t1) > depth(t2) then depth(t1)
                else depth(t2) end,
      [ add_balanced_elem ] ∀ e : Elem, t : Balanced_Tree •
        add_balanced(e,t) as rt
        post extract_elems(rt) = extract_elems(t) ∪ {e}
  end
```