Chapter 13
Reification and Decomposition

Z Specifications

Class1

d:P Person

---

d <= Max

$\Delta Class1 \mathrel{\widehat{=}} Class1 \wedge Class1'$

$\Xi Class1 \mathrel{\widehat{=}} [\Delta Class1 \mid d' = d]$

$InitClass1 \mathrel{\widehat{=}} [Class1 \mid d = \varnothing]$

Operations:

Leave1

$\Delta Class1$

p?: Person

$p? \notin d$

$d' = d \setminus \{p?\}$

Specification, Design, Implementation

- Formal Specifications help us to write programs and enable us to prove that a particular program either does or does not meet its specification.
- Z: Many high level abstract mathematical data types and operations are specified by giving their preconditions and postconditions
- In moving towards the programming language constructs that are available to us, we often rewrite the specification using less abstract data types and more 'algorithmic' constructs.
- This second specification in an intermediate between the specification and the final program and it is often referred to as the design.
- Any program which is a correct implementation of the design must also be a correct implementation of the specification.

Data Reification

- Classroom Specification - could implement the set d of people as a linked list or as an array, but any proofs relating specification and proof would be more complicated that necessary.
- A design involving sequences rather than sets would help here, reducing the distance between specification and implementation and resulting is much less complicated proofs.
- **Data Reification:** the process of taking an abstract mathematical data type, like a set and representing it in another way , e.g. as a sequence.
  - Transforms one data type to another
  - Transforms an abstract data type into a more concrete one.

Modelling Sets by Sequences

- Let U: P Person e.g. {Paul, Tom, James}
- We can represent the set U by means of an appropriate sequence of people σ:seq:Person, where seq person is the seq of all finite sequences of elements drawn from the type Person e.g.

  σ =<Tom, Paul, James>
- The abstract data type (ADT) consists of the elements of P person and the operations defined on those sets and the concrete data type (CDT) consists of all the elements of seq Person and the operations defined on those sequences.

Retrieve Function

- Retrieve Function maps the concrete data type onto the abstract one:
  Ret: seq Person →P person
- In case of sets being modelled as sequences **ret σ = ran σ**
  The Retrieve Function has to be from the concrete data type to the abstract one as many sequences correspond to the same set.

Operation Modelling

- Relate operations defined on ADT to operations defined on the concrete data type.
- If an operation ♦ in the concrete world correctly models an operation ♥, in the abstract world then ret(σ ♦ r) = (ret σ) ♥ (ret r)
- Example: Model set union

[X]

Append:seq X x seq X → seq X

∀x: X;σ, r:seq X •(append (<>, r) = r) ∧
(x ∈ ran r
⇒ append(<x> ^ σ, r) = append (σ, r)) ∧
(x ∉ ran r
⇒ append(<x> ^ σ, r) = <x> ^ append (σ, r))

Prove Correctness of the Representation

- In order to show that the append function properly models the world of sets we must prove:
  - ret (append (σ, r)) = (ret σ) ∪ (ret r)
- The standard way to prove something about a sequence is by using sequence induction
- To show that P(σ) holds for for all sequences σ establish:
  - P(<>)
  - if P(σ) holds for any sequence σ, then so does P(<x> ^ σ)
    i.e. ∀x: ; σ:seq X • P(σ) ⇒ P(<x> ^ σ)

- Prove: ret (append (σ, r)) = (ret σ) ∪ (ret r)

Base Case:

ret (append (< >, r)) = (ret <>) ∪ (ret r)
≡ <Definition of append>
ret r = (ret <>) ∪ (ret r)
≡ <ret r = ran r and ran <> = ∅>
ret r = ret r
≡ True

Inductive Case:

Prove	ret (append (<x> ^ σ, r)) = (ret (<x>)^σ)) ∪ (ret r)
assuming	ret (append (σ, r)) = (ret σ) ∪ (ret r)

Case 1: **x ∈ ran r** implies append(<x> ^ σ, r) = append (σ, r)

ret (append (<x> ^ σ, r)) = (ret (<x>)^σ)) ∪ (ret r)
≡
ret (append(<x> ^ σ, r)) = **ret (append (σ, r))**
≡ <by Ind. Hyp>
ret (append(<x> ^ σ, r)) = **(ret σ) ∪ (ret r)**
≡ <ret σ = ran σ and ran(σ ^ r)= ran σ ∪ ran r>
ret <x> ∪ ret σ ∪ ret r = (ret σ) ∪ (ret r)
≡
{x} ∪ ret σ ∪ ret r = (ret σ) ∪ (ret r)
≡ <x ∈ ran r >
(ret σ) ∪ **(ret r)** = (ret σ) ∪ (ret r).

Case 2: **x ∉ ran r**

ret (append (<x> ^ σ, r)) = (ret (<x>)^σ)) ∪ (ret r)
≡
ret(append(<x> ^ σ, r)) = ret (<x> ^ append (σ, r))
≡ <by def of append>
ret (append(<x> ^ σ, r)) = **ret <x> ∪ ret(append (σ, r))**
≡ <ret σ = ran σ and ran(σ ^ r) = ran σ ∪ ran r>
ret (append(<x> ^ σ, r)) = **ret <x> ∪ ret σ ∪ ret r**
≡
ret <x> ∪ ret σ ∪ ret r = ret <x> ∪ ret σ ∪ ret r

We have proved that append correctly models the behaviour of ∪ in the concrete model.

Modelling Set Intersection

- Set Intersection and Set Difference can be modelled in the world of sequences by means of functions inter and subtract.
- Both have the property that if σ and r do not contain any duplicate elements then neither does inter(σ, r) nor substract(σ, r).

[X]

inter:seq X x seq X $\rightarrow$ seq X

$\forall$x: X;σ, r:seq X •(inter (<>, r) = <>) $\wedge$
(x $\in$ ran r
$\Rightarrow$ inter(<x> ^ σ, r) = <x> ^ inter (σ, r)) $\wedge$
(x $\notin$ ran r
$\Rightarrow$ inter(<x> ^ σ, r) = inter (σ, r))

- Prove: ret(inter(σ, r)) = ret $\sigma \cap$ ret r

Modelling Set Difference

[X]

substract:seq X x seq X $\rightarrow$ seq X

$\forall$x: X;σ, r:seq X •(substract (<>, r) = <>) $\wedge$
(x $\in$ ran r
$\Rightarrow$ substract(<x> ^ σ, r) = <x> ^ substract (σ, r)) $\wedge$
(x $\notin$ ran r
$\Rightarrow$ substract(<x> ^ σ, r) = <x> ^ substract(σ, r))

- Prove: ret(substract(σ, r)) = ret σ \ ret r

Reification & Decomposition with Schemas

- **Schemas:** used to define the state space of some problem domain and to specify various operations on that state space.
- **Retrieve Functions:** relates objects in the concrete world to those object in the abstract world they are modelling.
- We may use a schema to perform the work done by a retrieve function

- **Specification:** the abstract specification
- **Design:** the concrete specification
- See Fig 13.3 Diller

Example Design and Specification

- Schemas InitClass1' and InitClass2' are the initial state schemas of the specification and the design
- The functions append and subtract have the property that if σ and r do not contain any duplicate members then neither does append(σ, r) nor subtract (σ, r).
- Hence we can be sure that ant state obtained from the initial state by the application of the operations Enter2 and Leave2 has the property that the sequence l that it is part of, doesn't contain any duplicate elements.

Relating Specification and Design

Class1Class 2

d:P Person
l:seq Person

# d <= Max
# l <= Max
d = ran l

Recall that when modelling a set U as a sequence σ, we said that U = ret σ. Here ret σ = ran σ.
Hence d = ran l in included in Class1Class2.

Correctness of Design

- Prove that the initial states correspond to one another
- Prove that for each operation in the design, that it is correct and applicable (this corresponds to proving the correctness criterion for the retrieve function).
- To prove that the initial states correspond in the abstract and concrete worlds we prove that InitClass2' $\wedge$ Class1Class2' $\Rightarrow$ InitClass1' is a theorem of FOL.
- Class1Class2 is like the retrieve function we talked about earlier.

Proof:

InitClass2' $\wedge$ Class1Class2' $\Rightarrow$ InitClass1'

$\equiv$ <by definition>

l'<= Max $\wedge$ l' = <> $\wedge$

d'<= Max $\wedge$ # l' <= Max $\wedge$ d' = ran l'

$\Rightarrow$ # d'<= Max $\wedge$ d' = $\emptyset$

This is true iff # l'<= Max, l' = <>, # d'<= Max, d' = ran l'

$\mapsto$ # d'<= Max $\wedge$ d' = $\emptyset$

is a valid sequent, which it is.

Applicability of Enter2

Prove: pre Enter1 $\wedge$ Class1Class2 $\Rightarrow$ pre Enter2 is a theorem of FOL

#d < Max $\wedge$ p? $\notin$ d $\wedge$

d<= Max $\wedge$ # l <= Max $\wedge$ d = ran l

$\Rightarrow$ # l < Max $\wedge$ p? $\notin$ ran l

This is a theorem of FOL iff the following is a valid sequent:

#d < Max, p? $\notin$ d, # d<= Max, # l <= Max, d = ran l

$\mapsto$ # l < Max $\wedge$ **p? $\notin$ ran l**

– # l < Max follows from #d < Max, d = ran l and # l = # ran l where # l = # ran l follows as l doesn't contain duplicates.

– **p? $\notin$ ran l** follows from p? $\notin$ d and d = ran l

Correctness of Enter2

Prove:

pre Enter1 $\wedge$ ΔClass1Class2 $\wedge$ Enter2 $\Rightarrow$ Enter2 is a theorem of FOL

#d < Max $\wedge$ p? $\notin$ d $\wedge$

d<= Max $\wedge$ # l <= Max $\wedge$ d = ran l $\wedge$

d'<= Max $\wedge$ # l' <= Max $\wedge$ d' = ran l' $\wedge$

l < Max $\wedge$ # l'<= Max $\wedge$ p? $\notin$ ran l $\wedge$ l' = append(I,<p?>)

$\Rightarrow$ # d < Max $\wedge$ # d'<= Max $\wedge$ p? $\notin$ d $\wedge$ d' = d $\cup$ {p?}

$\equiv$

#d < Max , p? $\notin$ d, d = ran l, # d'<= Max, # l' <= Max, d' = ran l',

l < Max, p? $\notin$ ran l, l' = append(l,<p?>)

$\mapsto$ # d < Max $\wedge$ # d'<= Max $\wedge$ p? $\notin$ d $\wedge$ d' = d $\cup$ {p?}