

# An Introduction to Data Refinement

J.N. Oliveira  
DIUM May 22, 2006

Formal Methods II, 2002-06,  
May 22, 2006

## FM software design process

---

- **Formal specification** — “what” the intended software system should do
- **Implementation** — machine code produced instructing the hardware about “how” to do it

In general, there is more than one way in which a particular machine can accomplish “what” the specifier bore in mind:

- Relationship between specifications and implementations is **one-to-many**
- Specifications are more **abstract** than implementations.

## Overall idea

---

- **Calculate** implementations from specifications

$$\begin{aligned} Spec &= X \\ &\leq X' \\ &\leq X'' \\ &\leq \dots \\ &\leq Imp \end{aligned}$$

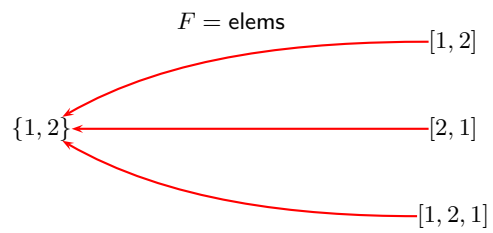
by adding **details** in a controlled manner.

- Define a suitable ordering  $\leq$  on datatypes and develop corresponding **data refinement** theory

## Example of data refinement

---

Finite **sets** represented by finite **lists**:



## Refinement inequation

---

$$\mathcal{P}_f A \begin{array}{c} \xrightarrow{\leq} \\ \xleftarrow{\text{elems}} \end{array} A^*$$

meaning that

- sets are “implemented” by lists
- $A^*$  is able to “represent”  $\mathcal{P}_f A$
- $A^*$  is “abstracted” by  $\mathcal{P}_f A$
- $A^*$  is a refinement (“refines”)  $\mathcal{P}_f A$

## Refinement inequations

$A$  is implemented by  $B$ , as witnessed by pair  $f, r$ , iff

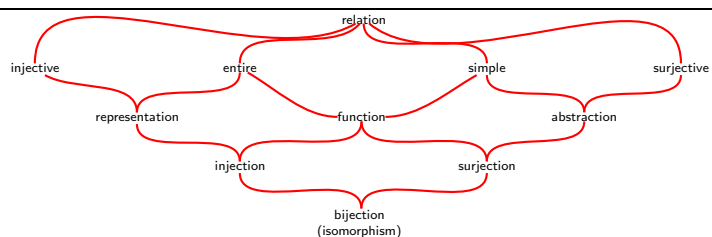
$$A \begin{array}{c} \xrightarrow{r} \\ \leq \\ \xleftarrow{f} \end{array} B$$

such that

- **representation**  $r$  is injective
- **abstraction**  $f$  is surjective
- that is,

$$f \cdot r = id$$

## Recall...



## Recall...

Taxonomy

	Reflexive	Coreflexive
$\ker R$	entire $R$	injective $R$
$\text{img } R$	surjective $R$	simple $R$

Kernel

$$\ker R \stackrel{\text{def}}{=} R^\circ \cdot R$$

Image (its dual)

$$\text{img } R \stackrel{\text{def}}{=} \ker (R^\circ)$$

## Not general enough (I)

In the following inequation

$$A \begin{array}{c} \xrightarrow{i_1} \\ \leq \\ \xleftarrow{i_1^\circ} \end{array} A + 1$$

expressing the fact that every element of datatype  $A$  can be represented by a “pointer”,

- $r = i_1$  is injective, but
- its converse  $i_1^\circ$  is **partial** (=not entirely defined)

## Not general enough (II)

Representations  $r$  need not be functions. Back to

$$\mathcal{P}_f A \begin{array}{c} \xrightarrow{R} \\ \leq \\ \xleftarrow{elems} \end{array} A^*$$

relation  $R = elems^\circ$  will be perfectly acceptable as a representation since

$$elems \cdot elems^\circ = id$$

because  $elems$  is a surjection.

## Data refinement

Principle of **data abstraction**:  $A$  abstracts  $B$  wherever

- A surjective **abstraction**  $A \xleftarrow{F} B$  can be found:

$$\text{img } F = id \quad (1)$$

$F$  is thus **simple** but possibly partial.

- Any **entire** subrelation  $R$  of  $F^\circ$  is said to be a **representation** for  $F$ . So  $R \subseteq F^\circ$ .

## Representation relations

- It follows that  $R$  is **injective**, since  $\ker R \subseteq \ker F^\circ$  and  $\ker F^\circ = \text{img } F = id$ .
- So, no two different abstract values  $a, a' \in A$  get mixed up along the representation process.
- Altogether,  $\ker R = id$  because  $id \subseteq \ker R \subseteq id$  ( $R$  is entire).
- It follows that  $R$  is a **right-inverse** of  $F$ , that is

$$F \cdot R = id \quad (2)$$

This is proved by circular inclusion

$$F \cdot R \subseteq id \subseteq F \cdot R$$

in the next slide.

## Right invertibility

$$\begin{aligned}
 & F \cdot R \subseteq id \wedge id \subseteq F \cdot R \\
 \equiv & \quad \{ \text{img } F = id \text{ and } \ker R = id \} \\
 & F \cdot R \subseteq F \cdot F^\circ \wedge R^\circ \cdot R \subseteq F \cdot R \\
 \equiv & \quad \{ \text{converses} \} \\
 & F \cdot R \subseteq F \cdot F^\circ \wedge R^\circ \cdot R \subseteq R^\circ \cdot F^\circ \\
 \Leftarrow & \quad \{ (F \cdot) \text{ and } (R^\circ \cdot) \text{ are monotone} \} \\
 & R \subseteq F^\circ \wedge R \subseteq F^\circ \\
 \equiv & \quad \{ R \subseteq F^\circ \text{ is assumed} \} \\
 & \text{TRUE}
 \end{aligned}$$

## Refinement inequations

$$A \begin{array}{c} \xrightarrow{R} \\ \leq \\ \xleftarrow{F} \end{array} B \quad \text{such that} \quad F \cdot R = id_A$$

This inequation has several informal interpretations:

- $A$  is “smaller” than  $B$
- $B$  is able to “represent”  $A$
- $B$  is “abstracted” by  $A$
- $A$  is “implemented” by  $B$
- $B$  is a refinement (“refines”)  $A$

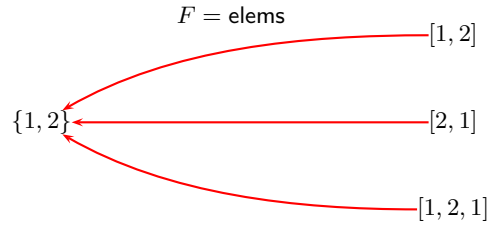
## Refinement equations

Isomorphisms:  $A \begin{array}{c} \xrightarrow{r} \\ \cong \\ \xleftarrow{f} \end{array} B \quad \text{such that} \quad r = f^\circ$

$$\begin{aligned} & r = f^\circ \\ \equiv & \quad \{ \text{add variables} \} \\ & b \, r \, a \equiv b \, f^\circ \, a \\ \equiv & \quad \{ \text{functions and converses} \} \\ & b = r \, a \equiv f \, b = a \end{aligned}$$

## Example

Back to representing finite **sets** by finite **lists**:



Among the many  $R \subseteq F^\circ$ , we may choose the following:

## Relational representation

```
Listify : set of nat -> seq of nat
Listify(s) ==
  if s = {} then []
  else let e in set s
    in [e] ^ Listify(s \ {e});
```

Intuitively,

$$\text{rng Listify} = \llbracket \text{noRepeats} \rrbracket$$

where

```
noRepeats(s) == card elems s = len s
```

## Functional representation

```
listify : set of nat -> seq of nat
listify(s) ==
  if s = {} then []
  else let e = minset(s)
    in [e] ^ listify(s \ {e});
```

Intuitively,

$$\text{rng listify} = \llbracket \text{IsOrdered} \rrbracket \cdot \llbracket \text{noRepeats} \rrbracket$$

## Concrete invariants

- Wherever

$$A \begin{array}{c} \xrightarrow{R} \\ \leq \\ \xleftarrow{F} \end{array} B \quad \text{such that } R \subseteq F^\circ \text{ and } \text{rng } R = [\phi]$$

we say that  $\phi$  is the **concrete invariant** induced by  $R$ .

- In case  $R$  is a function, and because it always is injective, one has

$$A \cong B_\phi$$

where  $B_\phi$  denotes the subset of  $B$  which satisfies concrete-invariant  $\phi$ .

## Example of a partial abstraction

Every element of datatype  $A$  can be represented by a “pointer”:

$$A \begin{array}{c} \xrightarrow{i_1} \\ \leq \\ \xleftarrow{i_1^\circ} \end{array} A + 1$$

- **Simplicity** of the abstraction is ensured by a known fact: the converse of an injective relation is simple.
- Concrete **invariant**:  $\phi = [\text{TRUE}, \text{FALSE}]$

## Another partial abstraction

Finite mappings “are” (simple) finite relations:

$$\text{map } A \text{ to } B \begin{array}{c} \xrightarrow{mkr} \\ \leq \\ \xleftarrow{mkf} \end{array} \text{set of } (A * B)$$

$mkf = mkr^\circ$

VDM-SL:

```
mkr : map A to B -> set of (A * B)
mkr(f) == { mk_(a,f(a)) | a in set dom f };

mkf : set of (A * B) -> map A to B
mkf(r) == { p.#1 |-> p.#2 | p in set r }
pre isSimple(r);
```

(Guess the concrete invariant.)



## Properties of $\leq$ :

### Reflexivity

$$A \begin{array}{c} \xrightarrow{id} \\ \leq \\ \xleftarrow{id} \end{array} A \quad \text{cf.} \quad id \cdot id = id$$

### Transitivity

$$A \begin{array}{c} \xrightarrow{R} \\ \leq \\ \xleftarrow{F} \end{array} B \quad \wedge \quad B \begin{array}{c} \xrightarrow{S} \\ \leq \\ \xleftarrow{G} \end{array} C \quad \Rightarrow \quad A \begin{array}{c} \xrightarrow{S \cdot R} \\ \leq \\ \xleftarrow{F \cdot G} \end{array} C$$

## Proof of transitivity

- First show that composition preserves simplicity and surjectiveness:

$$\begin{aligned} & \text{img}(F \cdot G) = id \\ \equiv & \quad \{ \text{expanding and converses} \} \\ & F \cdot (\text{img } G) \cdot F^\circ = id \\ \equiv & \quad \{ G \text{ is simple and surjective} \} \\ & \text{img } F = id \\ \equiv & \quad \{ F \text{ is simple and surjective} \} \\ & id = id \end{aligned}$$

- Then note that  $S \cdot R \subseteq (F \cdot G)^\circ$  by monotonicity.

## Structural data refinement

$$A \begin{array}{c} \xrightarrow{R} \\ \leq \\ \xleftarrow{F} \end{array} B \Rightarrow F A \begin{array}{c} \xrightarrow{F R} \\ \leq \\ \xleftarrow{F F} \end{array} F B$$

where  $F$  is an arbitrary relator (functor):

$$\begin{aligned} & (F F) \cdot (F R) \\ = & \quad \{ \text{relators commute with composition} \} \\ & F (F \cdot R) \\ = & \quad \{ R \text{ is right-inverse of } F \} \\ & F id \\ = & \quad \{ \text{relators commute with } id \} \\ & id \end{aligned}$$

therefore  $F R$  is right-inverse of  $F f$ . Of course, this result extends to bifunctors.

## Relators

A **relator** is a functor on relations

$$\begin{array}{ccc} A & & F A \\ X \downarrow & & \downarrow F X \\ B & & F B \end{array}$$

which is monotonic and commutes with converse:

$$\begin{aligned} R \subseteq S & \Rightarrow (F R) \subseteq (F S) \\ F (R^\circ) & = (F R)^\circ \end{aligned}$$

## Relators

Recall that  $F$  will commute with **composition** and **identity** too:

$$F (R \cdot S) = (F R) \cdot (F S) \quad (3)$$

$$F id = id \quad (4)$$

Example:  $X^*$  will be such that

$$l(X^*)l' \equiv \text{len } l = \text{len } l' \wedge \forall i \in \text{inds } l. (l \ i) X(l' \ i)$$

## Polynomial relators

$$\begin{array}{ll}
 \text{Identity:} & \text{Id } R = R \\
 \text{Constant:} & \text{K } R = id_K \\
 \text{Product:} & R \times S = \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \\
 \text{Sum:} & R + S = [i_1 \cdot R, i_2 \cdot S]
 \end{array}$$

where

$$\begin{aligned}
 \langle R, S \rangle &= \pi_1^\circ \cdot R \cap \pi_2^\circ \cdot S \\
 [R, S] &= (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ)
 \end{aligned}$$

For instance,

$$Maybe\ A \equiv (\text{Id} + 1)A = A + 1$$

## “Maybe” transpose

Useful isomorphism for conversion of simple relations into a *Maybe*-valued functions

$$\begin{array}{ccc}
 & \text{untot} = (i_1^\circ \cdot) & \\
 (B + 1)^A & \xrightleftharpoons[\text{tot}]{\cong} & A \multimap B
 \end{array}$$

where  $A \multimap B$  denotes the set of all simple relations from  $A$  to  $B$ :

$$f = \text{tot } R \equiv (b\ R\ a \equiv (f\ a = i_1\ b))$$

## “Maybe” transpose — VDM-SL

$$\begin{array}{ccc}
 & \text{tot} & \\
 A \multimap B & \xrightleftharpoons[\text{untot}]{\cong} & (B + 1)^A
 \end{array} \tag{5}$$

where, for types  $A$ ,  $B$  and  $\text{JustB} :: \text{value} : B$ ,

```

tot: map A to B -> A -> [JustB]
tot(sigma)(a) ==
  if a in set dom(sigma) then mk_JustB(sigma(a)) else nil;

untot: (A -> [JustB]) -> map A to B
untot(f) == { a |-> b | a: A, b: B & f(a) = mk_JustB(b) };

```

## Pointwise $untot = (i_1^\circ \cdot)$

As checked next:

$$\begin{aligned}
 untot\ f &= i_1^\circ \cdot f \\
 &\equiv \{ \text{relations as set comprehensions} \} \\
 untot\ f &= \{ (b, a) \mid a \in A, b \in B : b(i_1^\circ \cdot f)a \} \\
 &\equiv \{ \text{using rule } b(f^\circ \cdot R \cdot g)a \equiv (f\ b)R(g\ a) \} \\
 untot\ f &= \{ (b, a) \mid a \in A, b \in B : i_1\ b = f\ a \} \\
 &\equiv \{ \text{VDM-SL notation} \} \\
 untot\ f &= \{ a \mid \rightarrow b \mid a:A, b:B \ \& \ f(a) = \text{mk\_Just}(b) \}
 \end{aligned}$$

## Corol. of “Maybe” transpose (I)

Isomorphism

$$A^1 \xrightleftharpoons{\cong} A$$

extends to partial functions as follows:

$$1 \multimap A \xrightleftharpoons[f]{f^\circ} A + 1 \quad (\text{guess } f \text{ and } f^\circ).$$

That is, the “singleton” finite map is a disguise of a “pointer”.

## Corol. of “Maybe” transpose (II)

Sets are **degenerated** maps:

$$\mathcal{P}A \xrightleftharpoons{\cong} A \multimap 1$$

Calculation:

$$\begin{aligned}
 &A \multimap 1 \\
 \equiv &\{ \text{tot representation} \} \\
 &(1 + 1)^A \\
 \equiv &\{ \text{basic} \} \\
 &2^A \\
 \equiv &\{ 2^A \text{ is isomorphic to } \mathcal{P}A \} \\
 &\mathcal{P}A
 \end{aligned}$$

## Corol. of “Maybe” transpose (IIa)

$$\begin{array}{ccc} & \xrightarrow{\text{set2fm}} & \\ \mathcal{P}A & \overset{\cong}{\rightleftarrows} & A \rightarrow 1 \\ & \xleftarrow{\text{dom}} & \end{array}$$

VDM-SL

```
set2fm : set of A -> map A to Nil
set2fm(s) == { a |-> nil | a in set s };
```

Pointfree

$$\text{set2fm} \stackrel{\text{def}}{=} (!\cdot)$$

## Right-invertibility

Calculation:

$$\begin{aligned} & \text{dom} \cdot \text{set2fm} = id \\ \equiv & \quad \{ \} \\ & \text{dom} (\text{set2fm } s) = s \\ \equiv & \quad \{ \} \\ & \text{dom} (! \cdot s) = s \\ \equiv & \quad \{ ! \text{ is a function, } \text{dom} (f \cdot R) = \text{dom } R \} \\ & \text{dom } s = s \\ \equiv & \quad \{ s \text{ is coreflexive} \} \\ & s = s \end{aligned}$$

### Corol. of “Maybe” transpose (III)

Isomorphism

$$B \times C \multimap A \stackrel{\text{scurry}}{\cong} (C \multimap A)^B$$

extends currying

$$B^{C \times A} \stackrel{\text{curry}}{\cong} (B^A)^C$$

to simple relations, as calculated in the next slide.

### Corol. of “Maybe” transpose (III)

$$\begin{aligned} B \times C \multimap A &\cong \{ \text{tot/untot} \} \\ &= (A + 1)^{B \times C} \\ &\cong \{ \text{curry/uncurry} \} \\ &= ((A + 1)^C)^B \\ &\cong \{ (i_1^\circ \cdot)^B \} \\ &= (C \multimap A)^B \end{aligned}$$

This is referred to as the **multiple-key** decomposition / synthesis isomorphism.

### Corol. of “Maybe” transpose (III)

The *scurry* isomorphism is as follows, where we abbreviate *scurry*  $R$  to  $\overline{R}$ :

$$f = \overline{R} \equiv \langle \forall a, b, c : : c (f a) b \equiv c R (a, b) \rangle$$

Its VDM-SL equivalent for finite mappings is

```
scurry : map A*B to C -> (A -> map B to C)
scurry(M)(a) == { b |-> M(mk_(a',b))
                  | mk_(a',b) in set dom M
                  & a'=a };
```

### Corol. of “Maybe” transpose (IV)

Refinement of **nested** simplicity by decomposition:

$$\begin{array}{ccc}
 & \text{unnjoin} & \\
 A \multimap (D \times (B \multimap C)) & \leq & (A \multimap D) \times ((A \times B) \multimap C) \\
 & \text{\(\bowtie_n\)} & 
 \end{array}$$

where  $R \bowtie_n S = \langle R, \overline{S} \rangle$

and  $unnjoin\ R = (\pi_1 \cdot R, unpcurry(\pi_2 \cdot R))$  (see definition of  $unpcurry$  in the sequel.)

## Calculation

$$\begin{aligned}
& A \multimap (D \times (B \multimap C)) \\
\dot{=} & \quad \{ \text{Maybe transpose} \} \\
& ((D \times (B \multimap C)) + 1)^A \\
\leq & \quad \{ \text{Maybe-(right)strength is involved in the abstraction} \} \\
& ((D + 1) \times (B \multimap C))^A \\
\dot{=} & \quad \{ \text{splitting} \} \\
& (D + 1)^A \times (B \multimap C)^A \\
\dot{=} & \quad \{ \text{Maybe transpose and multiple-key synthesis} \} \\
& (A \multimap D) \times (A \times B \multimap C)
\end{aligned}$$

## Details on the $\bowtie_n$ abstraction

Pointwise:

$$\begin{aligned}
 (d, M)(R \bowtie_n S)a &\equiv d R a \wedge M = (\overline{S})a \\
 &\equiv \{ \text{scurry} \} \\
 &\quad d R a \wedge (c M b \equiv c S(a, b))
 \end{aligned}$$

VDM-SL equivalent for finite mappings:

```

njoin : (map A to D)*(map A*B to C)
        -> map A to (D* (map B to C))
njoin(M,N) ==
  { a |-> mk_(M(a), { b |-> N(mk_(a,b))
                    | mk_(a,b) in set dom N })
    | a in set dom M };

```

## Its representation is

```

unnjoin : map A to (D* map B to C) ->
          (map A to D)*(map A*B to C)
unnjoin(M) ==
  mk_({ a |-> M(a).#1 | a in set dom M },
    merge [{ mk_(a,b) |-> M(a).#2(b)
            | b in set dom M(a).#2 }
          | a in set dom M }
  );

```

Concrete invariant induced by *unnjoin*:

$$\phi_{unnjoin}(M, N) = N \preceq M \cdot \pi_1$$

where  $R \preceq S \equiv \text{dom } R \subseteq \text{dom } S$



### Corol. of “Maybe” transpose (V)

$$(B + C) \multimap A \quad \overset{\text{unpeither}}{\cong} \quad (B \multimap A) \times (C \multimap A)$$

*peither*

where

$$\text{peither}(\sigma, \tau) = [\sigma, \tau]$$

for  $[R, S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ)$ , that is

$$\text{peither} = \cup \cdot ((\cdot i_1^\circ) \times (\cdot i_2^\circ))$$

### Corol. of “Maybe” transpose (Va)

```
JustB :: value:B;
JustC :: value:C;
BorC  = JustB | JustC ;
```

$$\text{map } (\text{BorC}) \text{ to } A \quad \overset{\text{unpeither}}{\cong} \quad (\text{map } B \text{ to } A) \times (\text{map } C \text{ to } A)$$

*peither*

```
peither: (map B to A) * (map C to A) -> map BorC to A
peither(m,n) == { mk_JustB(b) |-> m(b) | b in set dom m } munion
                { mk_JustC(c) |-> n(c) | c in set dom n};
```

NB: a “1st NF” representation rule

## Relational projection

Given a binary relation  $R$  and suitably typed functions  $f$  and  $g$ ,

- the  $g, f$ -projection of  $R$  is defined as binary relation

$$\pi_{g,f}R \stackrel{\text{def}}{=} g \cdot R \cdot f^\circ \quad (6)$$

- wherever  $R$  is simple and  $g \cdot R \cdot f^\circ$  is also simple, we write  $f \rightarrow g$  instead of  $\pi_{g,f}R$ . So,

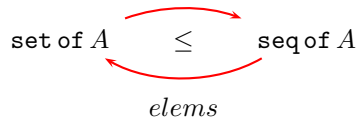
$$f \rightarrow g \stackrel{\text{def}}{=} (g \cdot) \cdot (\cdot f^\circ)$$

- $(f \rightarrow g)R$  is always simple when  $f$  is injective.
- So, we could have written e.g.

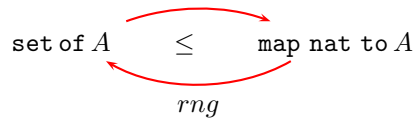
$$peither = \cup \cdot ((i_1 \rightarrow id) \times (i_2 \rightarrow id))$$

## Refining finite sets (II)

List (cf. example before):

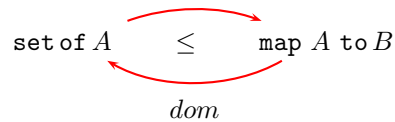


Index  $A$ :

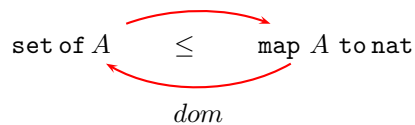


## Refining finite sets (III)

Classify  $A$  by  $B$  ( $B \supset \{\}$ ):



Quantify  $A$  ("multisets"):



## Refining finite maps (II)

$$\begin{array}{ccc}
 & \xrightarrow{\text{uncojoin}} & \\
 A \multimap (B + C) & \leq & (A \multimap B) \times (A \multimap C) \\
 & \xleftarrow{\text{cojoin}} &
 \end{array}$$

where

$$\text{cojoin} = \cup \cdot ((i_1 \cdot) \times (i_2 \cdot))$$

NB: *cojoin* is partial since the union of two partial functions not always is a partial function.

## Refining finite maps (IIa)

Note the representation function:

```

uncojoin : map A to BorC -> (map A to B) * (map A to C)
uncojoin(f) ==
  mk_( { a |-> f(a).value
        | a in set dom f & is_JustB(f(a)) },
        { a |-> f(a).value
        | a in set dom f & is_JustC(f(a)) }
  );

```

## Refining finite maps (III)

$$\begin{array}{ccc}
 & \xrightarrow{\text{unjoin}} & \\
 A \multimap B \times C & \leq & (A \multimap B) \times (A \multimap C) \\
 & \xleftarrow{\bowtie} &
 \end{array}$$

where

$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \langle \sigma, \tau \rangle$$

where  $\langle R, S \rangle \stackrel{\text{def}}{=} (\pi_1^\circ \cdot R) \cap (\pi_2^\circ \cdot S)$ . A right-inverse of *join* is

$$\text{unjoin} \stackrel{\text{def}}{=} \langle id \multimap \pi_1, id \multimap \pi_2 \rangle$$

## Refining finite maps (IIIa)

$$\text{map } A \text{ to } B * C \leq (\text{map } A \text{ to } B) \times (\text{map } A \text{ to } C)$$

where (writing join for  $\bowtie$ )

```
join : (map A to B) * (map A to C) -> map A to (B * C)
join(m,n) == { a |-> mk_(m(a),n(a))
              | a in set dom m inter dom n };
```

## Refining finite maps (IVa)

In general:

$$(C \times A) \multimap B \leq C \multimap (A \multimap B)$$

*pcurry* (top arrow), *unpcurry* (bottom arrow)

```
unpcurry : map C to (map A to B) -> map (C * A) to B
unpcurry(f) ==
  merge { let g=f(a)
          in { mk_(a,b) |-> g(b) | b in set dom g }
        | a in set dom f };
```

## Refining finite maps (IVb)

Pointwise

```
pcurry : map (C * A) to B -> map C to (map A to B)
pcurry(f) ==
  let y = { x.#1 | x in set dom f }
  in { a |-> { p.#2 |-> f(p)
             | p in set dom f & p.#1=a }
    | a in set y };
```

Pointfree

$$\text{pcurry } M = \overline{M} - \underline{\perp}$$

(recall *scurry*)

## Transposing relations

Let  $B := 2$  in the *curry/uncurry* isomorphism and obtain

$$\begin{array}{ccc} & \Lambda & \\ \mathcal{P}(A \times C) & \xrightarrow{\quad \cong \quad} & (\mathcal{P}A)^C \\ & \Lambda^\circ & \end{array}$$

where

$$f = \Lambda R \quad \equiv \quad R = \in \cdot f \quad (7)$$

and  $A \xleftarrow{\in} \mathcal{P}A$  is the membership relation.

## Transposing finite relations

$$\begin{array}{ccc} & \text{collect} & \\ \text{set of } (C * A) & \xrightarrow{\quad \leq \quad} & \text{map } C \text{ to set of } A \\ & \text{discollect} & \end{array}$$

```
collect : set of (C * A) -> map C to set of A
collect(r) == { c |-> { q.#2 | q in set r & c=q.#1 }
                  | c in set { p.#1 | p in set r } };

discollect : map C to set of A -> set of (C * A)
discollect(f) == dunion { { mk_(c,a) | a in set f(c) }
                          | c in set dom f };
```

## What about recursive data?

How does one refine *recursive* VDM-SL models such as e.g.

```
FS :: D: map Id to Node; -- FS means file system
Node = File | FS;       -- a Node is either a file
                        -- or a directory
Id = seq of char;       -- node identifiers
File :: F: seq of token -- sequential files
```

that is,  $FS = \mu F$  for  $F X = Id \rightarrow (File + X)$ :

$$\begin{array}{ccc} & \text{out} & \\ \mu F & \xrightarrow{\quad \cong \quad} & Id \rightarrow (File + \mu F) \\ & \text{in} & \end{array}$$

## The DecTree example

or...

```
DecTree :: question: What
         answers: map Answer to DecTree
What = seq of char;
Answer = seq of char;
```

that is,  $DecTree = \mu F$  in

$$DecTree \cong What \times (Answer \rightarrow DecTree)$$

for  $F X = What \times (Answer \rightarrow X)$

## The Exp example

or even...

```
Exp      = Var | Term ;
Var      :: variable: Symbol ;
Term     :: operator: Symbol
          arguments: seq of Exp
          inv t == len t.arguments <= 20 ;
Symbol   = seq of char
          inv s == len s <= 10 ;
```

that is,  $Exp = \mu F$  in

$$Exp \cong Symbol + Symbol \times Exp^*$$

for  $F X = Symbol + Symbol \times X^*$

## Getting away with recursion

Given

$$\mu F \begin{array}{c} \xrightarrow{\text{out}} \\ \cong \\ \xleftarrow{\text{in}} \end{array} F \mu F$$

one has

$$\mu F \begin{array}{c} \xrightarrow{\leq} \\ \xleftarrow{F} \end{array} \underbrace{(K \rightarrow F K) \times K}_{\text{"heap''}}$$

for  $K$  a data type of "heap addresses", or "pointers", such that  $K \cong \mathbb{N}$ .

## An example to start with

Since

$$Exp = \mu X. (Symbol + Symbol \times X^*)$$

we have:

$$\begin{aligned} & Exp \\ \leq & \quad \{ \text{remove recursion} \} \\ & (K \rightarrow (Symbol + Symbol \times K^*)) \times K \\ \leq & \quad \{ \text{remove finite lists} \} \\ & (K \rightarrow (Symbol + Symbol \times (IN \rightarrow K))) \times K \end{aligned}$$

## Example continued

$$\begin{aligned} & \leq \quad \{ \text{recall } A \rightarrow (B + C) \leq (A \rightarrow B) \times (A \rightarrow C) \} \\ & (K \rightarrow Symbol) \times (K \rightarrow (Symbol \times (IN \rightarrow K))) \times K \\ & \leq \quad \{ \text{remove nested } \rightarrow \} \\ & (K \rightarrow Symbol) \times (K \rightarrow Symbol) \times ((IN \times K) \rightarrow K) \times K \\ & \doteq \quad \{ A \times A \doteq A^2 \} \\ & (K \rightarrow Symbol)^2 \times ((IN \times K) \rightarrow K) \times K \\ & \doteq \quad \{ \text{recall } (C \rightarrow A)^B \doteq B \times C \rightarrow A \} \\ & \underbrace{((2 \times K) \rightarrow Symbol)}_{SYMBOLS} \times \underbrace{((IN \times K) \rightarrow K)}_{EXPRESSIONS} \times K \end{aligned}$$

## SQL encoding

Symbols table:

```
CREATE TABLE SYMBOLS (
  Symbol CHAR (20) NOT NULL,
  NodeId NUMERIC (10) NOT NULL,
  IfVar BOOLEAN NOT NULL
  CONSTRAINT SYMBOLS_pk
    PRIMARY KEY(NodeId, IfVar)
);
```

## SQL encoding

Expressions table:

```
CREATE TABLE EXPRESSIONS (
  FatherId NUMERIC (10) NOT NULL,
  ArgNr    NUMERIC (10) NOT NULL,
  ChildId  NUMERIC (10) NOT NULL
  CONSTRAINT EXPRESSIONS_pk
    PRIMARY KEY (FatherId, ArgNr)
);
```

Can you **rely** on this implementation? Need for an **abstraction** invariant!

## Abstraction function

- Main rôle in representation is played by simple **F-coalgebra**  $K \rightarrow F K$ , understood as a (finite) piece of “linear storage”, a “heap” or a “database” file.
- $\overline{F}$  (recall  $\overline{F}$  notation from above), of type  $(K \rightarrow \mu F)^{(K \rightarrow F K)}$ , is nothing but the **F-anamorphism** combinator:

$$\begin{array}{ccc}
 \mu F & \xleftarrow{\text{in}} & F(\mu F) \\
 \overline{F}H \uparrow & & \uparrow F(\overline{F}H) \\
 K & \xrightarrow{H} & F K
 \end{array}
 \quad
 \begin{array}{l}
 \overline{F} = \llbracket - \rrbracket_F \\
 \overline{F} H = \mu X. \text{in} \cdot (F X) \cdot H
 \end{array}$$

## Partiality of implementation

$F(\sigma, k) = (\overline{F}\sigma)k$  will be undefined wherever

- $k \notin \text{dom } \sigma$
- $\sigma$  is not “closed” over itself (see below)
- $\sigma$  is non-well-founded (see below)

Thus concrete invariant

$$\phi(\sigma, k) \stackrel{\text{def}}{=} k \in \text{dom } \sigma \wedge (\text{closed } \sigma) \wedge (\text{wellf } \sigma)$$

In order to define **closed**  $\sigma$  and **wellf**  $\sigma$  we need  $\sigma$ ’s **accessibility** relation  $\prec_\sigma$  (next slide).



## Accessibility and membership

Accessibility relation for  $\sigma$ :

$$K \xleftarrow{\sigma} K$$

$$\xleftarrow{\sigma} \stackrel{\text{def}}{=} \in_F \cdot \sigma$$

where  $K \xleftarrow{\in_F} F K$  extends  $K \xleftarrow{\in} \mathcal{P}K$  inductively over polynomial functors, as follows:

- Constant and identity functors:

$$\in_C \stackrel{\text{def}}{=} \perp$$

$$\in_{\lambda X.X} \stackrel{\text{def}}{=} id$$

## Membership (continued)

- Product and coproduct

$$\in_{F \times G} \stackrel{\text{def}}{=} (\in_F \cdot \pi_1) \cup (\in_G \cdot \pi_2)$$

$$\in_{F+G} \stackrel{\text{def}}{=} [\in_F, \in_G]$$

- Functor composition

$$\in_{F \cdot G} \stackrel{\text{def}}{=} \in_G \cdot \in_F$$

- Type functors: just an example,

$$\in_{X^*} \stackrel{\text{def}}{=} \in \cdot elems$$

## Example

Recall  $\mathbf{F} X = \mathbf{Symbol} + \mathbf{Symbol} \times X^\star$

$$\begin{aligned}
 & \in_{\mathbf{Symbol} + \mathbf{Symbol} \times X^\star} \\
 = & \quad \{ \in \text{ for coproduct bifunctor } \} \\
 & [\in_{\mathbf{Symbol}} , \in_{\mathbf{Symbol} \times X^\star}] \\
 = & \quad \{ \in \text{ for constant and product (bi)functors } \} \\
 & [\perp , (\in_{\mathbf{Symbol}} \cdot \pi_1) \cup (\in_{X^\star} \cdot \pi_2)] \\
 = & \quad \{ \in \text{ for constant and identity functor } \} \\
 & [\perp , (\perp \cdot \pi_1) \cup (\in \cdot \mathit{elems} \cdot \pi_2)] \\
 = & \quad \{ \perp \text{ and } [R, S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \} \\
 & \in \cdot \mathit{elems} \cdot \pi_2 \cdot i_2^\circ
 \end{aligned}$$

## Example (pointwise)

$$\begin{aligned}
 & k \in_{\mathbf{Symbol} + \mathbf{Symbol} \times X^\star} x \\
 \equiv & \quad \{ \text{ calculation above } \} \\
 & k(\in \cdot \mathit{elems} \cdot \pi_2 \cdot i_2^\circ) x \\
 \equiv & \quad \{ \text{ relational composition } \} \\
 & k(\in \cdot \mathit{elems} \cdot \pi_2)(a, l) \wedge x = i_2(a, l) \\
 \equiv & \quad \{ \text{ trivia } \} \\
 & k \in (\mathit{elems} \, l) \wedge x = i_2(a, l)
 \end{aligned}$$

## Another example

Let  $F X = 1 + A \times X$ . Then,

$$\begin{aligned}
 & \in_{1+A \times X} \\
 = & \quad \{ \in \text{ for coproduct bifunctor } \} \\
 & [\in_1, \in_{A \times X}] \\
 = & \quad \{ \in \text{ for constant and product (bi)functors } \} \\
 & [\perp, (\in_A \cdot \pi_1) \cup (\in_{\lambda X.X} \cdot \pi_2)] \\
 = & \quad \{ \in \text{ for constant and identity functor } \} \\
 & [\perp, (\perp \cdot \pi_1) \cup (id \cdot \pi_2)] \\
 = & \quad \{ \perp \text{ and } [R, S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \} \\
 & \pi_2 \cdot i_2^\circ
 \end{aligned}$$

## Example (pointwise)

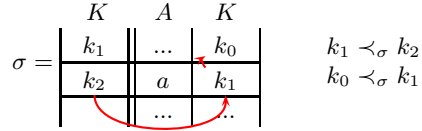
$$\begin{aligned}
 & k \in_{1+A \times X} x \\
 \equiv & \quad \{ \text{ calculation above } \} \\
 & k(\pi_2 \cdot i_2^\circ)x \\
 \equiv & \quad \{ \text{ relational composition } \} \\
 & k(\pi_2)(a, k') \wedge x = i_2(a, k') \\
 \equiv & \quad \{ \text{ trivia } \} \\
 & x = i_2(a, k') \wedge k = k' \\
 \equiv & \quad \{ \text{ trivia } \} \\
 & x = i_2(a, k)
 \end{aligned}$$

## Accessibility (linear example)

Pointer reachability in case of a "linear" heap  $(1 + A \times K) \xleftarrow{\sigma} K$ :

$$k_1 \prec_{\sigma} k_2 \quad \equiv \quad k_2 \in \text{dom } \sigma \wedge (\sigma k_2) = i_2(a, k_1)$$

In a drawing:



## Closure and wellfoundedness

Let  $\prec_{\sigma}^+$  denote the transitive closure of  $\prec_{\sigma}$ . Then we define

$$\text{closed } \sigma \quad = \quad \text{rng } \prec_{\sigma}^+ \subseteq \text{dom } \sigma$$

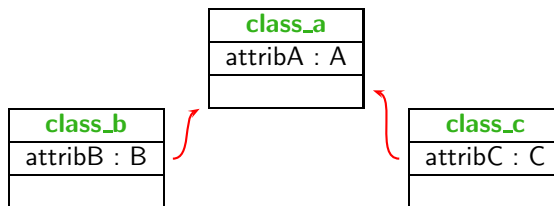
that is, all reachable  $k$  are defined, and

$$\text{wellf } \sigma \quad = \quad (\prec_{\sigma}^+) \cap \text{id} = \perp$$

that is,  $\prec_{\sigma}^+$  is irreflexive (no cycles, no looping)

## O.O. Data Implementation

UML:

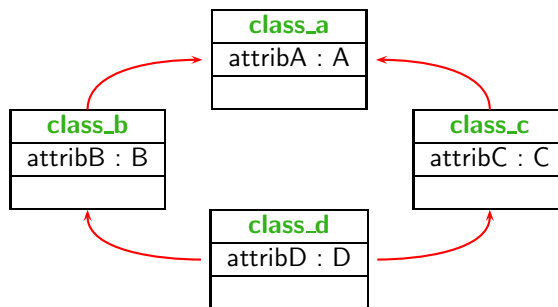


Formal model:  $K \rightarrow \text{Structure}$  where

$$\begin{aligned} \text{Structure} &= A + A \times B + A \times C \\ &\cong A \times (1 + B + C) \end{aligned}$$

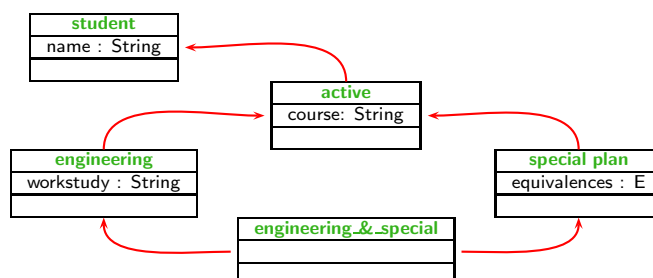
$$K \rightarrow (A + A \times B)$$

## Multiple inheritance



$$K \rightarrow A \times (1 + B + C + B \times C \times D)$$

## Example



$$K \rightarrow A \times (1 + B + C + B \times C)$$