

# An Introduction to Algorithmic Refinement

J.N. Oliveira

Formal Methods II, 2002-06

June 4, 2006

## Implicit/explicit refinement

Given VDM-SL implicit specification

```
S(a:A) r:B
pre ...
post ...
```

function  $B \xleftarrow{f} A$  is said to **satisfy**, to **refine**, or to **implement**  $S$ , written

$$S \vdash f$$

iff, for every  $a$ ,

$$\forall a \in A. \quad \text{pre-}S \ a \Rightarrow \text{post-}S(f \ a, a)$$

## In pointfree notation

$$\begin{aligned} & a \in \text{dom } S \Rightarrow (f \ a)Sa \\ \equiv & \quad \{ \text{rule } (f \ b)Ra \equiv b(f^\circ \cdot R)a \} \\ & \text{dom } S \subseteq f^\circ \cdot S \\ \equiv & \quad \{ \text{shunting} \} \\ & f \cdot \text{dom } S \subseteq S \end{aligned}$$

Summary: **explicit** specification (= **implementation**)  $f$  is thus more defined and more deterministic than **implicit** specification  $S$ :

$$S \vdash f \quad \equiv \quad f \cdot \text{dom } S \subseteq S \quad (1)$$

## Example

Recall

```
IsPermutation: seq of int * seq of int -> bool
IsPermutation(l1,l2) ==
  forall e in set (elems l1 union elems l2) &
    card {i | i in set inds l1 & l1(i) = e} =
      card {i | i in set inds l2 & l2(i) = e};
```

We want to find  $f$  such that

$$IsPermutation \vdash f$$

Recall that  $IsPermutation = \ker seq2bag$ , where...

## About seq2bag

VDM-SL definition:

```
seq2bag(s) ==
  cases s:
    [] -> {}
    others -> { hd s |-> 1 } bunion seq2bag(tl s)
  end;
```

Definition of gene  $g$  of the seq2bag catamorphism:

$$g = [\{\mapsto\}, \oplus \cdot (singb \times id)]$$

where  $singb\ a = \{a \mapsto 1\}$  and  $\oplus$  denotes bag union (bunion is not standard VDM-SL: define it).

## Implementing *IsPermutation*

$$\begin{aligned}
 & \text{IsPermutation} \vdash f \\
 \equiv & \quad \{ \text{definition} \} \\
 & f \cdot \text{dom IsPermutation} \subseteq \text{IsPermutation} \\
 \equiv & \quad \{ \text{definition} \} \\
 & f \cdot \text{dom} (\ker \text{seq2bag}) \subseteq \ker \text{seq2bag} \\
 \equiv & \quad \{ \text{kernel of a function} \} \\
 & f \cdot \text{id} \subseteq \text{seq2bag}^\circ \cdot \text{seq2bag} \\
 \equiv & \quad \{ \text{shunting rule} \} \\
 & \text{seq2bag} \cdot f \subseteq \text{seq2bag} \\
 \equiv & \quad \{ \text{equality of functions} \} \\
 & \text{seq2bag} \cdot f = \text{seq2bag}
 \end{aligned}$$

## Handling refinement equations

$f$  is the “unknown” of refinement equation

$$\text{seq2bag} \cdot f = \text{seq2bag}$$

Since  $\text{seq2bag}$  and  $f$  are list catamorphisms, one can resort to cata-fusion,

$$\begin{aligned}
 & \text{seq2bag} \cdot f = \text{seq2bag} \\
 \equiv & \quad \{ \text{let } f = \langle \alpha \rangle \text{ and } \text{seq2bag} = \langle g \rangle \} \\
 & \text{seq2bag} \cdot \langle \alpha \rangle = \langle g \rangle \\
 \Leftarrow & \quad \{ \text{cata-fusion} \} \\
 & \text{seq2bag} \cdot \alpha = g \cdot (\text{id} + \text{id} \times \text{seq2bag})
 \end{aligned}$$

## Solving refinement equations

By decomposing  $\alpha := [\beta, \gamma]$ , we obtain equations

$$\begin{aligned}
 \beta &= \underline{[]} \\
 \text{seq2bag} \cdot \gamma &= \oplus \cdot (\text{singb} \times \text{seq2bag})
 \end{aligned}$$

- Cata-cancellation yields solution  $\gamma = \text{cons}$ , leading to  $\alpha = \text{in}$  and  $f = \text{id}$ .
- $\oplus$  is commutative, thus solution  $\gamma(a, l) = l \hat{^} [a]$  leading to  $f = \text{invl}$ .

Guessing further solutions: any list **sorting** function will solve the equation!  
(More about this later...)

## Properties of $\vdash$

Basic:

$$\perp \vdash f \quad , \quad \top \vdash f \quad (2)$$

$$(S \cap R) \vdash f \iff S \vdash f \wedge R \vdash f \quad (3)$$

$$(S \cup R) \vdash f \iff S \vdash f \wedge R \vdash f \quad (4)$$

$$(\ker g) \vdash f \iff g \cdot f = g \quad (5)$$

$$g \vdash f \iff f = g \quad (6)$$

Monotonicity:

$$S \vdash f \implies F S \vdash F f \quad (7)$$

## Proof of monotonicity

$$\begin{aligned}
 & F S \vdash F f \\
 \equiv & \quad \{ \text{definition} \} \\
 & (F f) \cdot \text{dom} (F S) \subseteq F S \\
 \equiv & \quad \{ \text{property } \text{dom} (F S) = F(\text{dom } R) \} \\
 & (F f) \cdot F(\text{dom } S) \subseteq F S \\
 \equiv & \quad \{ \text{relators commute with composition} \} \\
 & F(f \cdot \text{dom } S) \subseteq F S \\
 \Leftarrow & \quad \{ \text{relators are monotone} \} \\
 & f \cdot \text{dom } S \subseteq S \\
 \equiv & \quad \{ \text{definition} \} \\
 & S \vdash f
 \end{aligned}$$

## Stepwise refinement

Extend  $f$  in  $S \vdash f$  to a relation

$$S \vdash R \equiv R \cdot \text{dom } S \subseteq S \wedge \text{dom } S \subseteq \text{dom } R \quad (8)$$

Obs.:

- clause  $\text{dom } S \subseteq \text{dom } R$  ensures that implementations can only be **more defined**
- clause  $R \cdot \text{dom } S \subseteq S$  ensures that implementations can only be **more deterministic**
- Note that  $\perp \vdash R$  still holds but, in general,  $\top \vdash R$  requires  $R$  to be **entire**, since  $\text{dom } \top = \text{id}$ .

## Example

Let spec  $S_{\nu, \epsilon}$  be

```
sqrt (x: real) r: real
pre  abs(x) <= nu
post abs(r*r-x) <= epsilon
```

Then, wherever  $\nu_1 \leq \nu_2$  and  $\epsilon_1 \geq \epsilon_2$ ,

$$S_{\nu_1, \epsilon_1} \vdash S_{\nu_2, \epsilon_2}$$

In the “limit”,  $\dots \vdash S_{\infty, 0} = sq^\circ \vdash f$  where  $f \ x = +\sqrt{x}$  or  $f \ x = -\sqrt{x}$ .

## Refinement is a partial order

**Reflexivity:**  $\text{id} \subseteq \vdash$ , that is

$$S \vdash S$$

**Transitivity:**  $\vdash \cdot \vdash \subseteq \vdash$ , that is

$$S \vdash R \wedge R \vdash T \Rightarrow S \vdash T$$

**Antisymmetry:**  $\vdash \cap \vdash^\circ \subseteq \text{id}$

$$S \vdash R \wedge R \vdash S \Rightarrow S = R$$

**F-monotonicity:**

$$S \vdash R \Rightarrow \mathbf{F} S \vdash \mathbf{F} R$$

## Stepwise refinement

---

The laws of  $\vdash$  make it possible to refine a starting spec  $S$  along several steps,

$$S \vdash S_1 \vdash S_2 \vdash \dots$$

each one introducing more and more definition and/or determinism, and very often leading into a function (totally defined deterministic algorithm):

$$S \vdash S_1 \vdash S_2 \vdash \dots \vdash S_n \vdash f$$

What do we do after  $f$ ?

## Back to $g \vdash f$

---

- Formally,  $g \vdash f \equiv g = f$ , that is, spec  $g$  is **extensionally** equivalent to implementation  $f$ .
- But there is more to it: in general, we think of  $f$  as being “more **efficient**” than  $g$ .
- Efficiency can only be formalized in the discipline of **algorithmic complexity** (out of scope here)
- We will study functional laws which add to efficiency and generalize well-known (**while**) loop generation and intercombination rules.

## Main refinement strategies

---

- Refinement by “sequential loop” inter-combination: **fusion** and **absorption** laws:

“**Deforestation**” — removal of intermediate data-structures

- Refinement by “parallel loop” inter-combination: **mutual recursion** elimination:

On this purpose we will see Fokkinga’s law and its well-known corollary, the “banana-split” law.

## Mutual recursion elimination

Consider the following pair of mutually dependent functions:

```
f(n) == if n = 0 then n else g(n - 1);
g(n) == if n = 0 then 1 else f(n - 1) + g(n - 1);
```

Can any of these functions — say  $g$  — be converted into a while loop?

In pointfree notation:

$$\begin{aligned} f \cdot [\underline{0}, suc] &= [id, g] \\ g \cdot [\underline{0}, suc] &= [\underline{1}, + \cdot \langle f, g \rangle] \end{aligned}$$

## Mutual dependence made explicit

$$\begin{aligned} f \cdot [\underline{0}, suc] &= [id, \pi_2 \cdot \langle f, g \rangle] \\ g \cdot [\underline{0}, suc] &= [\underline{1}, + \cdot \langle f, g \rangle] \end{aligned} \quad \text{cf.} \quad N_0 \begin{array}{c} \xrightarrow{\quad \cong \quad} \\ \xleftarrow{\quad \quad} \end{array} \underbrace{1 + N_0}_{F N_0}$$

which is such that  $F f = id + f$ . So (+-absorption)  $in = [id, suc]$  we can write

$$\begin{aligned} f \cdot in &= [id, \pi_2] \cdot F \langle f, g \rangle \\ g \cdot in &= [\underline{1}, +] \cdot F \langle f, g \rangle \end{aligned}$$

## The mutual-recursion law

This situation is handled by the so-called **mutual-recursion law**, also called “Fokkinga law”:

$$\begin{cases} f \cdot in = h \cdot F \langle f, g \rangle \\ g \cdot in = k \cdot F \langle f, g \rangle \end{cases} \quad \equiv \quad \langle f, g \rangle = \langle \langle h, k \rangle \rangle$$

that is, in general

$$\begin{cases} f_1 \cdot in = h_1 \cdot F \langle f_1, \dots, f_n \rangle \\ \vdots \\ f_n \cdot in = h_n \cdot F \langle f_1, \dots, f_n \rangle \end{cases} \quad \equiv \quad \langle f_1, \dots, f_n \rangle = \langle \langle h_1, \dots, h_n \rangle \rangle$$

## Proof

$$\begin{aligned}
 \langle f, g \rangle &= \langle \langle h, k \rangle \rangle \\
 &\equiv \{ \text{cata-universal} \} \\
 \langle f, g \rangle \cdot in &= \langle h, k \rangle \cdot F \langle f, g \rangle \\
 &\equiv \{ \times\text{-fusion twice (lhs and rhs)} \} \\
 \langle f \cdot in, g \cdot in \rangle &= \langle h \cdot F \langle f, g \rangle, k \cdot F \langle f, g \rangle \rangle \\
 &\equiv \{ \text{"split" structural equality} \} \\
 &\quad \left\{ \begin{array}{l} f \cdot in = h \cdot F \langle f, g \rangle \\ g \cdot in = k \cdot F \langle f, g \rangle \end{array} \right.
 \end{aligned}$$

## Example

Let  $h = [id, \pi_2]$  and  $k = [\underline{1}, +]$  in the example above:

$$\begin{aligned}
 \langle f, g \rangle &= \{ \text{Fokkinga law} \} \\
 &\quad \langle \langle [id, \pi_2], [\underline{1}, +] \rangle \rangle \\
 &= \{ \text{exchange law} \} \\
 &\quad \langle [\langle id, \underline{1} \rangle, \langle \pi_2, + \rangle] \rangle
 \end{aligned}$$

```

fg(n) == if n = 0 then mk_(0,1)
         else let p=fg(n-1)
              in mk_(p.#2,p.#1 + p.#2);

```



## Example

Since  $fg = \langle f, g \rangle$ , we obtain  $g = \pi_2 \cdot fg$ . On the other hand, it is easy to extract  $g$  from

```
f(n) == if n = 0 then n else g(n - 1);  
g(n) == if n = 0 then 1 else f(n - 1) + g(n - 1);
```

as the standard Fibonacci function:

```
g(n) == if n = 0 then 1  
        else if n = 1 then 1  
        else g(n - 2) + g(n - 1);
```

Summary: we have calculated  $\pi_2 \cdot fg$  as a **linear** version of Fibonacci ( $g = \pi_2 \cdot fg$ ).

## Corollary: “banana-split” (1)

Consider the function which computes the **average** of a non-empty list of natural numbers:

$$average \stackrel{\text{def}}{=} (/) \cdot \langle sum, length \rangle$$

Both  $sum$  and  $length$  are  $\mathbb{N}^+$  catamorphisms:

$$sum = ([id, +]) \\ length = ([1, succ \cdot \pi_2])$$

Function  $average$  performs two independent **traversals** of the argument list before division ( $/$ ) takes place. Can we avoid this? “Banana-split” will fuse such two traversals.

## Corollary: “banana-split” (2)

Let  $h = i \cdot F \pi_1$  and  $k = j \cdot F \pi_2$  in the mutual recursion law. Then

$$\begin{aligned} f \cdot in &= (i \cdot F \pi_1) \cdot F \langle f, g \rangle \\ &\equiv \quad \{ \text{composition is associative and } F \text{ is a functor} \} \\ f \cdot in &= i \cdot F (\pi_1 \cdot \langle f, g \rangle) \\ &\equiv \quad \{ \text{by } \times\text{-cancellation} \} \\ f \cdot in &= i \cdot F f \\ &\equiv \quad \{ \text{by cata-cancellation} \} \\ f &= ([i]) \end{aligned}$$

### Corollary: “banana-split” (3)

Similarly,  $g = \langle j \rangle$  will follow from  $k = j \cdot F \pi_2$ . Then, from the mutual recursion law we get

$$\langle \langle i \rangle, \langle j \rangle \rangle = \langle \langle i \cdot F \pi_1, j \cdot F \pi_2 \rangle \rangle$$

that is

$$\langle \langle i \rangle, \langle j \rangle \rangle = \langle (i \times j) \cdot \langle F \pi_1, F \pi_2 \rangle \rangle \quad (9)$$

This law provides us with a very useful tool for “parallel” loop inter-combination: “loops”  $\langle i \rangle$  and  $\langle j \rangle$  are fused together into a single “loop”  $\langle (i \times j) \cdot \langle F \pi_1, F \pi_2 \rangle \rangle$ .

### Genericity of “banana-split”

Banana-split fuses two data-structure traversals (“loops”) in the generic sense. For instance,

$$average \stackrel{\text{def}}{=} (/) \cdot \langle sum, length \rangle$$

still makes sense in the case of binary leaf trees, for

$$sum = \langle [id, +] \rangle$$

$$length = \langle [1, +] \rangle$$

Again  $sum$  and  $length$  can be fused together (bi-recursively).

### Recursion elimination

Very common pattern in functional hylomorphisms:

$$\begin{aligned} f &: A \longrightarrow C \\ f &= p \rightarrow b, \theta \cdot \langle d, f \cdot e \rangle \end{aligned}$$

Diagram:

$$\begin{array}{ccccc} A & \xrightarrow{p?} & A + A & \xrightarrow{id + \langle d, e \rangle} & A + B \times A \\ f \downarrow & & & & \downarrow id + id \times f \\ C & \xleftarrow{[b, \theta]} & A + B \times C & & \end{array}$$

That is,

$$\langle \mu f : [b, \theta] \cdot (id + id \times f) \cdot (id + \langle d, e \rangle) \cdot p? \rangle$$

## More general intermediate type

By splitting  $b = b_2 \cdot b_1$  we obtain a more general intermediate type

$$\begin{aligned} f & : A \longrightarrow C \\ f & = p \rightarrow b_2 \cdot b_1, \theta \cdot \langle d, f \cdot e \rangle \end{aligned}$$

Diagram:

$$\begin{array}{ccccc} A & \xrightarrow{p?} & A + A & \xrightarrow{b_1 + \langle d, e \rangle} & D + B \times A \\ f \downarrow & & & & \downarrow id + id \times f \\ C & \xleftarrow{[b_2, \theta]} & & & D + B \times C \end{array}$$

## Associative $\theta$

The extra requirement (required by calculations to follow) that  $\theta$  to be associative, ie for all  $i, j, k$ ,

$$\theta \cdot \langle \theta \cdot \langle i, j \rangle, k \rangle = \theta \cdot \langle i, \theta \cdot \langle j, k \rangle \rangle$$

leads us finally to the type scheme

$$\begin{array}{ccccc} A & \xrightarrow{p?} & A + A & \xrightarrow{b_1 + \langle d, e \rangle} & D + C \times A \\ f \downarrow & & & & \downarrow id + id \times f \\ C & \xleftarrow{[b_2, \theta]} & & & D + C \times C \end{array}$$

Can we eliminate recursion from this hylomorphism, eg. convert it to a while loop?

## What is a while loop

A while loop

$$\underline{\text{while } p \text{ do } l} = (\neg \cdot p) \rightarrow id, (\underline{\text{while } p \text{ do } l}) \cdot l$$

is a very special case of hylomorphism  $\llbracket R, S \rrbracket$  where  $S$  does all the work and  $R$  does nothing:

$$\begin{array}{ccccc} & (id + l) \cdot (\neg \cdot p)? & & & \\ \underline{\text{while } p \text{ do } l} & \downarrow & A & \xrightarrow{\quad} & A + A \\ & & \downarrow & & \downarrow id + (\underline{\text{while } p \text{ do } l}) \\ & & A & \xleftarrow{\quad} & A + A \\ & & [id, id] & & \end{array}$$

## Calculation plan

We want to convert the given hylo

$$f = p \rightarrow b, \theta \cdot \langle d, f \cdot e \rangle$$

into

$$\begin{aligned} f &= p \rightarrow b, w \cdot \langle d, e \rangle \\ w &= x \rightarrow y, w \cdot z \end{aligned}$$

the latter being the obvious `while`-loop

$$w = y \cdot (\underline{while} (\neg \cdot x) \underline{do} z)$$

The unknowns of the problem are  $x, y$  and  $z$ .

## Calculation

Clearly, finding  $w$  such that

$$w = \theta \cdot (id \times f)$$

is enough. We recall McCarthy-fusion laws

$$\begin{aligned} f \cdot (p \rightarrow g, h) &= p \rightarrow f \cdot g, f \cdot h \\ (p \rightarrow f, g) \cdot h &= (p \cdot h) \rightarrow (f \cdot h), (g \cdot h) \end{aligned}$$

and calculate:

$$\begin{aligned} &\langle \mu f : : \underbrace{p \rightarrow b, \theta \cdot \langle d, f \cdot e \rangle}_{Ff} \rangle \\ = &\{ \text{square rule, } \langle \mu f : : Ff \rangle = \langle \mu f : : F(Ff) \rangle \} \end{aligned}$$

### Calculation (contd.)

$$\begin{aligned}
& \langle \mu f :: p \rightarrow b, \theta \cdot \langle d, (p \rightarrow b, \theta \cdot \langle d, f \cdot e \rangle) \cdot e \rangle \rangle \\
= & \quad \{ \text{McCarthy-fusion laws} \} \\
& \langle \mu f :: p \rightarrow b, p \cdot e \rightarrow \theta \cdot \langle d, b \cdot e \rangle, \theta \cdot \langle d, \theta \cdot \langle d, f \cdot e \rangle \cdot e \rangle \rangle \\
= & \quad \{ \times\text{-fusion and McCarthy-fusion again} \} \\
& \langle \mu f :: p \rightarrow b, \\
& \quad (p \cdot \pi_2 \rightarrow \theta \cdot (id \times b), \theta \cdot (id \times (\theta \cdot \langle d, f \cdot e \rangle))) \cdot \langle d, e \rangle \rangle \\
= & \quad \{ \theta \text{ is associative} \} \\
& \langle \mu f :: p \rightarrow b, \\
& \quad p \cdot \pi_2 \rightarrow \theta \cdot (id \times b), \theta \cdot \langle \theta \cdot (id \times d), f \cdot e \cdot \pi_2 \rangle \cdot \langle d, e \rangle \rangle
\end{aligned}$$

### Calculation (contd.)

We now focus on the inner conditional:

$$\begin{aligned}
& p \cdot \pi_2 \rightarrow \theta \cdot (id \times b), \theta \cdot \langle \theta \cdot (id \times d), f \cdot e \cdot \pi_2 \rangle \\
= & \quad \{ \times\text{-absorption} \} \\
& p \cdot \pi_2 \rightarrow \theta \cdot (id \times b), \theta \cdot (id \times f) \cdot \langle \theta \cdot (id \times d), e \cdot \pi_2 \rangle \\
= & \quad \{ \text{pattern matching (recall assumption } w = \theta \cdot (id \times f)) \} \\
& \underbrace{\underbrace{p \cdot \pi_2}_x \rightarrow \underbrace{\theta \cdot (id \times b)}_y, \underbrace{\theta \cdot (id \times f)}_w \cdot \underbrace{\langle \theta \cdot (id \times d), e \cdot \pi_2 \rangle}_z}_{w}
\end{aligned}$$

So we found tail-recursive solution

$$w = p \cdot \pi_2 \rightarrow \theta \cdot (id \times b), w \cdot \langle \theta \cdot (id \times d), e \cdot \pi_2 \rangle$$

### Calculation (contd.)

Altogether:

$$\begin{aligned}
f &= p \rightarrow b, w \cdot \langle d, e \rangle \\
w &= p \cdot \pi_2 \rightarrow \theta \cdot (id \times b), w \cdot \langle \theta \cdot (id \times d), e \cdot \pi_2 \rangle
\end{aligned}$$

Quite often  $b = \underline{u}$ , where  $u$  is the unit of  $\theta$  ( $a \theta u = a$ ). We then get a simpler  $w$ :

$$\begin{aligned}
w &= p \cdot \pi_2 \rightarrow \pi_1, \langle \theta \cdot (id \times d), e \cdot \pi_2 \rangle \\
&= \pi_1 \cdot \underline{while} \neg \cdot p \cdot \pi_2 \underline{do} \langle \theta \cdot (id \times d), e \cdot \pi_2 \rangle
\end{aligned}$$

## Example - factorial

From

$$fac = (=0) \rightarrow \underline{1}, mul \cdot \langle id, fac \cdot pred \rangle$$

we get VDM-SL functions

```
fac(n) == if n=0 then 1 else w(n,n-1)
w(m,i) == if i=0 then m else w(m * i, i-1 )
```

The next slide shows how the while loop can be brought explicit by moving from functions to operations.

## Example - factorial

```
fac : nat ==> nat
fac(n) ==
  if n=0 then return 1
  else (dcl m: nat := n,
        i: nat := n -1;
        while (i <> 0) do
          (m := m * i; i := i-1);
        return m;
  );
```

## Other results

For  $b = \underline{u}$  there is another recursion removal law which we could prove similarly:

$$\begin{aligned} f & : A \longrightarrow C \\ f & = p \rightarrow \underline{u}, \theta \cdot \langle d, f \cdot e \rangle \end{aligned}$$

is equivalent to

$$\begin{aligned} f & = w \cdot \langle \underline{u}, id \rangle \\ w & = p \cdot \pi_2 \rightarrow \pi_1, w \cdot \langle \theta \cdot (id \times d), e \cdot \pi_2 \rangle \end{aligned}$$

The corresponding VDM-SL version of factorial is presented in the slide which follows:

### Example - “another” factorial

---

```
fac : nat ==> nat
fac(n) ==
  (dcl m: nat := 1,
   i: nat := n;
   while (i <> 0) do
     (m := m * i; i := i - 1);
   return m
  );
```

### For versus while loops

---

VDM-SL/VDM++ syntax includes for loops. Therefore, the two while loops above can be converted into, respectively,

```
fac(n) ==
  if n=0 then return 1
  else (dcl m: nat := n;
        for i=n-1 to 1 by -1 do m := m * i;
        return m;
  );

and

fac(n) ==
  (dcl m: nat := 1;
   for i = n to 1 by -1 do m := m * i;
   return m
  );
```

## Data refinement in full

Simultaneous **algorithm/data** refinement: given

- a spec  $A \xleftarrow{S} B$
  - abstraction function  $A \xleftarrow{F_1} C$
  - representation relation  $D \xleftarrow{R_2} B$
- then  $C \xleftarrow{I} D$  will be said to **implement**  $S$  iff

$$S \vdash F_1 \cdot I \cdot R_2 \quad \begin{array}{ccc} & A & \xleftarrow{S} B \\ F_1 \uparrow & & \downarrow R_2 \\ & C & \xleftarrow{I} D \end{array} \quad (10)$$

## Analysis of refinement equation

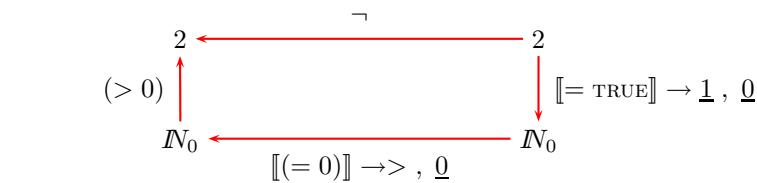
- The above refinement equation is to be solved for  $I$  (the unknown), and will in general exhibit more than one solution.
- $S \vdash F_1 \cdot I \cdot R_2$  means

$$F_1 \cdot I \cdot R_2 \cdot \text{dom } S \subseteq S \quad \wedge \quad \text{dom } S \subseteq \text{dom } (F_1 \cdot I \cdot R_2)$$

- In case  $F_i = R_i = \text{id}$  ( $i = 1, 2$ ) — no data refinement involved — it boils down to algorithmic refinement:

$$S \vdash \text{id} \cdot I \cdot \text{id}$$

## Example



recall

$$R \rightarrow S, T \stackrel{\text{def}}{=} (S \cdot \text{dom } R) \cup T \cdot (\text{id} - \text{dom } R)$$

Note how non-determinism of implementation is coped with by the target abstraction function.



## Solving refinement equations

Since  $\text{dom}(S \cdot R) = \text{dom}(\text{dom } S \cdot R)$ , the second clause above rewrites to

$$\text{dom } S \subseteq \text{dom}(\text{dom } F_1 \cdot (I \cdot R_2))$$

In case  $F_1(f_1)$  is entire:

$$\text{dom } S \subseteq \text{dom}(I \cdot R_2)$$

In case  $\text{spec } S$  and  $F_1(f_1)$  are entire and  $R_2 = f_2^\circ$ ,  $I$  will be entire and such that

$$I \subseteq f_1^\circ \cdot S \cdot f_2$$

## Functional solutions

Case in which all entities in a refinement equation are total functions (note the lowercase letters):

$$f_1 \cdot i = s \cdot f_2 \quad (11)$$

- Example:  $i = f^*$  will implement  $s = \mathcal{P}f$  under data-refinement  $f_1 = f_2 = \text{elems}$ .
- $i = f^*$  is not a unique solution. These arise wherever  $f_1$  is iso ( $f_1^\circ$  is a function):

$$i = f_1^\circ \cdot s \cdot f_2$$

This appeals to calculating  $i$  by cata-fusion over inductive implementation type  $D$ .

## Example

Set by list refinement:

$$(a \text{ belongs}) = (a \in) \cdot \text{elems}$$

( $f_1 = \text{id}$ ):

$$\begin{array}{ccc} & (a \in) & \\ \text{id} \uparrow & \xleftarrow{\quad} & \mathcal{P}A \\ 2 & & \uparrow \text{elems} \\ & \xleftarrow{\quad} & A^* \\ & a \text{ belongs} & \end{array}$$

We know that  $\text{elems} = \langle \text{ins} \rangle$ . Since target function is a list cata  $(a \text{ belongs}) = \langle \beta \rangle$ , by cata-fusion refinement equation will hold provided  $(a \in) \cdot \text{ins} = \beta \cdot (\text{id} + \text{id} \times (a \in))$  holds.

### Example (cont.)

---

Let  $\beta = [\beta_1, \beta_2]$ .

- Since  $a \in \emptyset = \text{FALSE}$ , we calculate  $\beta_1 = \underline{\text{FALSE}}$ .
- We are left with

$$a \in (\{x\} \cup s) = \beta_2(x, a \in s)$$

From  $a \in \{x\} \cup s = (a \in \{x\}) \vee (a \in s)$ , we infer  $\beta_2(x, b) \equiv a = x \vee b$ .

- All in all:

```
belongs(a)(l) ==  
  if l = [] then false  
  else (a = hd l) or belongs(a)(tl l)
```

## Bibliografia