An Introduction to Algorithmic Refinement

J.N. Oliveira

Formal Methods II, 2002-06

June 17, 2007

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,

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

In pointfree notation

$$\begin{array}{ll} & a \in \delta\, S \Rightarrow (f\ a) S a \\ \equiv & \quad \{\ \text{rule } (f\ b) R a \equiv b (f^\circ \cdot R) a\ \} \\ & \delta\, S \subseteq f^\circ \cdot S \\ \equiv & \quad \{\ \text{shunting}\ \} \\ & f \cdot \delta\, S \subseteq S \end{array}$$

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

$$S \vdash f \quad \equiv \quad f \cdot \delta\, S \subseteq S \tag{1}$$

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:

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

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

Implementing $IsPermutation$

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

Handling refinement equations

f is the "unknown" of refinement equation

$$seq2bag \cdot f = seq2bag$$

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

$$
\begin{array}{ll}
 & seq2bag \cdot f = seq2bag \\
\equiv & \quad \{ \text{ let } f = (\!|\alpha|\!) \text{ and } seq2bag = (\!|g|\!) \} \\
 & seq2bag \cdot (\!|\alpha|\!) = (\!|g|\!) \\
\Leftarrow & \quad \{ \text{ cata-fusion } \} \\
 & seq2bag \cdot \alpha = g \cdot (id + id \times seq2bag)
\end{array}
$$

Solving refinement equations

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

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

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

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

Properties of $\vdash$

Basic:

$$\bot \vdash f \quad , \quad \top \vdash f \tag{2}$$
$$(S \cap R) \vdash f \quad \Leftarrow \quad S \vdash f \ \wedge \ R \vdash f \tag{3}$$
$$(S \cup R) \vdash f \quad \Leftarrow \quad S \vdash f \ \wedge \ R \vdash f \tag{4}$$
$$(\ker g) \vdash f \quad \equiv \quad g \cdot f = g \tag{5}$$
$$g \vdash f \quad \equiv \quad f = g \tag{6}$$

Monotonicity:

$$S \vdash f \quad \Rightarrow \quad \mathsf{F}\, S \vdash \mathsf{F}\, f \tag{7}$$

Proof of monotonicity

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

Stepwise refinement

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

$$S \vdash R \quad \equiv \quad R \cdot \delta\, S \subseteq S \;\wedge\; \delta\, S \subseteq \delta\, R \tag{8}$$

Obs.:

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

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", $\cdots \vdash S_{\infty,0} = sq^\circ \vdash f$ where $f\ x = +\sqrt{x}$ or $f\ x = -\sqrt{x}$.

Refinement is a partial order

Reflexivity: $id \subseteq \vdash$, that is

$$S \vdash S$$

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

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

Antisymmetry: $\vdash \cap \vdash^\circ \subseteq id$

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

F-monotonicity:

$$S \vdash R \quad \Rightarrow \quad \mathsf{F}\, S \vdash \mathsf{F}\, R$$

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);
```

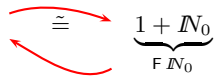
Can any of these functions — say g — be converted into a while loop?
In pointfree notation:

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

Mutual dependence made explicit

$$\begin{aligned} f \cdot [\underline{0}\ , suc] &= [id\ , \pi_2 \cdot \langle f, g \rangle] \\ g \cdot [\underline{0}\ , suc] &= [\underline{1}\ , + \cdot \langle f, g \rangle] \end{aligned}$$

cf.



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

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

The mutual-recursion law

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

$$\left\{ \begin{array}{l} f \cdot in = h \cdot \mathsf{F}\ \langle f, g \rangle \\ g \cdot in = k \cdot \mathsf{F}\ \langle f, g \rangle \end{array} \right. \quad \equiv \quad \langle f, g \rangle = (\!|\langle h, k \rangle|\!)$$

that is, in general

$$\left\{ \begin{array}{l} f_1 \cdot in = h_1 \cdot \mathsf{F}\ \langle f_1, \ldots, f_n \rangle \\ \vdots \\ f_n \cdot in = h_n \cdot \mathsf{F}\ \langle f_1, \ldots, f_n \rangle \end{array} \right. \quad \equiv \quad \langle f_1, \ldots, f_n \rangle = (\!|\langle h_1, \ldots, h_n \rangle|\!)$$

Proof

$$
\begin{array}{ll}
& \langle f, g\rangle = (\!|\langle h, k\rangle|\!) \\
\equiv & \quad \{ \text{ cata-universal } \} \\
& \langle f, g\rangle \cdot in = \langle h, k\rangle \cdot \mathsf{F}\,\langle f, g\rangle \\
\equiv & \quad \{ \text{ } \times\text{-fusion twice (lhs and rhs)} \} \\
& \langle f \cdot in, g \cdot in\rangle = \langle h \cdot \mathsf{F}\,\langle f, g\rangle, k \cdot \mathsf{F}\,\langle f, g\rangle\rangle \\
\equiv & \quad \{ \text{ "split" structural equality } \} \\
& \begin{cases} f \cdot in = h \cdot \mathsf{F}\,\langle f, g\rangle \\ g \cdot in = k \cdot \mathsf{F}\,\langle f, g\rangle \end{cases}
\end{array}
$$

Example

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

$$
\begin{array}{ll}
\langle f, g\rangle & \\
= & \quad \{ \text{ Fokkinga law} \} \\
& (\!|\langle [id\ , \pi_2], [\underline{1}\ , +]\rangle|\!) \\
= & \quad \{ \text{ exchange law} \} \\
& (\!|[\langle id, \underline{1}\rangle\ , \langle \pi_2, +\rangle]|\!)
\end{array}
$$

```
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:

$$sum = (\!|[id\ , +]|\!)$$
$$length = (\!|[\underline{1}\ , succ \cdot \pi_2]|\!)$$

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 $h = i \cdot \mathsf{F}\,\pi_1$ and $k = j \cdot \mathsf{F}\,\pi_2$ in the mutual recursion law. Then

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

Corollary: "banana-split" (3)

Similarly, $g = (\!|j|\!)$ will follow from $k = j \cdot \mathsf{F}\,\pi_2$ Then, from the mutual recursion law we get

$$\langle (\!|i|\!), (\!|j|\!) \rangle = (\!|\langle i \cdot \mathsf{F}\,\pi_1, j \cdot \mathsf{F}\,\pi_2 \rangle|\!)$$

that is

$$\langle (\!|i|\!), (\!|j|\!) \rangle = (\!|(i \times j) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2 \rangle|\!) \quad (9)$$

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

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

$$sum = (\!|[id\ , +]|\!)$$
$$length = (\!|[\underline{1}\ , +]|\!)$$

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

Recursion elimination

Very common pattern in functional hylomorphisms:

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

Diagram:

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

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

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

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 θ

The extra requirement (required by calculations to follow) that θ to be assciative, 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

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

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

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

Calculation plan

We want to convert the given hylo

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

into

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

the latter being the obvious `while`-loop

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

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

Calculation

Clearly, finding w such that

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

is enough. We recall McCarthy-fusion laws

$$\begin{array}{rcl} f \cdot (p \to g \ , \ h) & = & p \to f \cdot g, f \cdot h \\ (p \to f, g) \cdot h & = & (p \cdot h) \to (f \cdot h), (g \cdot h) \end{array}$$

and calculate:

$$\langle \mu \ f \ :: \ \underbrace{p \to b \ , \ \theta \cdot \langle d, f \cdot e \rangle}_{\mathsf{F}\, f} \rangle$$

$=$ { square rule, $\langle \mu \ f \ :: \ \mathsf{F} \ f \rangle = \langle \mu \ f \ :: \ \mathsf{F}(\mathsf{F} \ f) \rangle$ }

Calculation (contd.)

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

Calculation (contd.)

We now focus on the inner conditional:

$$
\begin{array}{ll}
 & p \cdot \pi_2 \to \theta \cdot (id \times b)\ ,\ \theta \cdot \langle \theta \cdot (id \times d), f \cdot e \cdot \pi_2\rangle \\
= & \quad \{\ \times\text{-absorption}\ \} \\
 & p \cdot \pi_2 \to \theta \cdot (id \times b)\ ,\ \theta \cdot (id \times f) \cdot \langle \theta \cdot (id \times d), e \cdot \pi_2\rangle \\
= & \quad \{\ \text{pattern matching (recall assumption } w = \theta \cdot (id \times f))\ \} \\
 & \underbrace{\underbrace{p \cdot \pi_2}_{x} \to \underbrace{\theta \cdot (id \times b)}_{y}\ ,\ \underbrace{\theta \cdot (id \times f)}_{w} \cdot \underbrace{\langle \theta \cdot (id \times d), e \cdot \pi_2\rangle}_{z}}_{w}
\end{array}
$$

So we've found tail-recursive solution

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

Calculation (contd.)

Altogether:

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

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

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

Example - factorial

From

$$fac \quad = \quad (=_0) \rightarrow \underline{1} \ , \ mul \cdot \langle id, fac \cdot pred \rangle$$

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 made explicit by moving from funcions 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 recusion removal law which we could prove similarly:

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

is equivalent to

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

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

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

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

$$F_1 \cdot I \cdot R_2 \cdot \delta\, S \subseteq S \quad \wedge \quad \delta\, S \subseteq \delta\,(F_1 \cdot I \cdot R_2)$$

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

$$S \vdash id \cdot I \cdot id$$

Example

$$\begin{array}{ccc} 2 & \xleftarrow{\neg} & 2 \\ {\scriptstyle (>0)}\uparrow & & \downarrow{\scriptstyle [\![= \text{TRUE}]\!] \rightarrow \underline{1}\ ,\ \underline{0}} \\ I\!N_0 & \xleftarrow[[\![(=0)]\!] \rightarrow >\ ,\ \underline{0}]{} & I\!N_0 \end{array}$$

recall

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

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

Solving refinement equations

Since $\delta\,(S \cdot R) = \delta\,(\delta\,S \cdot R)$, the second clause above rewrites to

$$\delta\,S \subseteq \delta\,(\delta\,F_1 \cdot (I \cdot R_2))$$

In case F_1 (f_1) is entire:

$$\delta\,S \subseteq \delta\,(I \cdot R_2)$$

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

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

Functional solutions

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

$$f_1 \cdot i \quad = \quad s \cdot f_2 \tag{11}$$

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

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

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

Example

Set by list refinement:

$$(a\ belongs) \quad = \quad (a \in) \cdot elems$$

($f_1 = id$):

$$\begin{array}{ccc} & (a \in) & \\ 2 & \longleftarrow & \mathcal{P}A \\ id \uparrow & & \uparrow elems \\ 2 & \longleftarrow & A^\star \\ & a\ belongs & \end{array}$$

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