An Introduction to Algorithmic Refinement

Formal Methods II, 2002/03

J.N. Oliveira

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

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

Summary: **explicit** specifi cation (= **implementation**) f is thus more defi ned and more deterministic than **implicit** specifi cation S:

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

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:

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

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

Implementing $IsPermutation$

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

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,

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

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:

$$\bot \vdash f \quad , \quad \top \vdash f \tag{2}$$

$$(S \cap R) \vdash f \;\Leftarrow\; S \vdash f \wedge R \vdash f \tag{3}$$

$$(S \cup R) \vdash f \;\Leftarrow\; S \vdash f \wedge R \vdash f \tag{4}$$

$$(\textit{ker}\ g) \vdash f \;\equiv\; g \cdot f = g \tag{5}$$

$$g \vdash f \;\equiv\; f = g \tag{6}$$

Monotonicity:

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

Proof of monotonicity

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

Stepwise refinement

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

$$S \vdash R \quad \equiv \quad R \cdot \mathit{dom}\, S \subseteq S \wedge \mathit{dom}\, S \subseteq \mathit{dom}\, R \tag{8}$$

Obs.:

- clause $\mathit{dom}\, S \subseteq \mathit{dom}\, R$ ensures that implementations can only be **more defined**
- clause $R \cdot \mathit{dom}\, S \subseteq S$ 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 $\mathit{dom}\, \top = id$.

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: $\vdash \subseteq id$, 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: $\vdash \cap \vdash^{\circ} \subseteq id$

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

F-monotonicity:

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

Stepwise refinement

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

$$S \vdash S_1 \vdash S_2 \vdash \ldots$$

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

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

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

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

cf.

$$
\begin{tikzcd}
I\!N_0 \arrow[rr, bend left, ""] \arrow[rr, phantom, "\cong"] & & \underbrace{1 + I\!N_0}_{\mathsf{F}\, I\!N_0} \arrow[ll, bend left, ""]
\end{tikzcd}
$$

$$in = [id, suc]$$

which is such that F $f = id + f$. So ($+$-absorption) we can write

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

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

$$\langle f, g \rangle = (\!|\langle h, k \rangle|\!)$$

$$\equiv \quad \{ \text{ cata-universal } \}$$

$$\langle f, g \rangle \cdot in = \langle h, k \rangle \cdot \mathsf{F}\, \langle f, g \rangle$$

$$\equiv \quad \{ \; \times\text{-fusion twice (lhs and rhs)} \}$$

$$\langle f \cdot in, g \cdot in \rangle = \langle h \cdot \mathsf{F}\, \langle f, g \rangle, k \cdot \mathsf{F}\, \langle f, g \rangle \rangle$$

$$\equiv \quad \{ \text{ “split” structural equality } \}$$

$$\left\{ \begin{array}{l} f \cdot in = h \cdot \mathsf{F}\, \langle f, g \rangle \\ g \cdot in = k \cdot \mathsf{F}\, \langle f, g \rangle \end{array} \right.$$

Example

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

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

```
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:

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

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

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

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

$$f \cdot in = (i \cdot \mathsf{F}\, \pi_1) \cdot \mathsf{F}\, \langle f, g \rangle$$

$\equiv$ { composition is associative and F is a functor}

$$f \cdot in = i \cdot \mathsf{F}\, (\pi_1 \cdot \langle f, g \rangle)$$

$\equiv$ { by $\times$-cancellation }

$$f \cdot in = i \cdot \mathsf{F}\, f$$

$\equiv$ { by cata-cancellation}

$$f = (\!|i|\!)$$

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

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

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

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

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

Data refinement in full

Simultaneous algorithm/data refinement: given

- a spec $B \xleftarrow{S} A$
- 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

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

- In case $F = R = id$ (no data refinement involved), it boils down to algorithmic refinement:

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

Example

$$
\begin{tikzcd}
2 & & 2 \arrow[ll, "\neg"'] \arrow[d, "id \to \underline{1}\,,\ \underline{0}"] \\
\mathbb{N}_0 \arrow[u, "(0 \neq)"] & & \mathbb{N}_0 \arrow[ll, "(0 =) \to > \,,\ \underline{0}"]
\end{tikzcd}
$$

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

Solving refinement equations

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

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

In case F_1 (f_1) is entire:

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

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 refi nement equation are total functions (note the lowercase letters):

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

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

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

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

Example

Set by list refinement:

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

$(f_1 = id)$:

$$\begin{tikzcd} 2 & \mathcal{P}A \arrow[l, "(a\ \in)"'] \\ 2 \arrow[u, "id"] & A^\star \arrow[l, "a\ belongs"] \arrow[u, "elems"'] \end{tikzcd}$$

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)
```

Calculation of `while/for` loops

Left-linear recursion: refinement towards `while/for` loops — see pp. 125–131 of

J.N. Oliveira. Operation refinement, June 2000. Departamento de Informática, Universidade do Minho. Chapter of book in preparation.