Uma Introdução ao Refinamento de Algoritmos

J.N. Oliveira

5 de Maio de 2004

Refinamento implícito/explícito

Dada uma especificação implícita em VDM-SL

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

diz-se que a função $B \xleftarrow{f} A$ satisfaz, refina, ou implementa S, escrevendo-se

$$S \vdash f$$

se e só se, para todo a,

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

Em notação sem variáveis

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

Resumo: a especificação **explícita** (= **implementação**) f é portanto mais definida e mais determinística que a especificação **implícita** S:

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

Exemplo

Recordar

```
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};
```

Queremos encontrar f tal que

$$IsPermutation \vdash f$$

Recorde que $IsPermutation = ker\ seq2bag$, onde, em VDM-SL...

Acerca de seq2bag

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

Assim, $seq2bag = (\!|g|\!)$ para

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

onde $singb\ a = \{a \mapsto 1\}$ e $\oplus$ denota a união de "bags" (**NB:** `bunion` não é VDM-SLstandard: experimente defini-la).

Implementando $IsPermutation$

$$
\begin{array}{ll}
 & IsPermutation \vdash f \\
\equiv & \quad \{ \text{ definição } \} \\
 & f \cdot dom\, IsPermutation \subseteq IsPermutation \\
\equiv & \quad \{ \text{ definição } \} \\
 & f \cdot dom\,(ker\ seq2bag) \subseteq ker\ seq2bag \\
\equiv & \quad \{ \text{ kernel duma função } \} \\
 & f \cdot id \subseteq seq2bag^{\circ} \cdot seq2bag \\
\equiv & \quad \{ \text{ "shunting" } \} \\
 & seq2bag \cdot f \subseteq seq2bag \\
\equiv & \quad \{ \text{ igualdade de funções } \} \\
 & seq2bag \cdot f = seq2bag
\end{array}
$$

Equações de refinamento

f é a "incógnita" da equação de refinamento

$$seq2bag \cdot f = seq2bag$$

Uma vez que $seq2bag$ e f são catamorfismos sobre listas, podemos recorrer à fusão-cata

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

Sua resolução

Decompondo $\alpha := [\beta, \gamma]$, obtemos as equações

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

- Por cancelamento-cata, temos a solução $\gamma = cons$, obtendo $\alpha = in$ e $f = id$.
- $\oplus$ é comutativa, logo é solução $\gamma(a, l) = l \mathbin{\hat{}} [a]$ obtendo-se $f = invl$.

Encontrar mais soluções: qualquer função de ordenação de listas será solução da equação! (Veremos mais acerca disto depois...)

Propriedades de $\vdash$

Básicas:

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

Monotonia:

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

Prova da monotonia

$$
\begin{array}{ll}
 & \mathsf{F}\ S \vdash \mathsf{F}\ f \\
\equiv & \quad \{\ \text{definição}\} \\
 & (\mathsf{F}\ f) \cdot dom\,(\mathsf{F}\ S) \subseteq \mathsf{F}\ S \\
\equiv & \quad \{\ \text{propriedade } dom\,(\mathsf{F}\ S) = \mathsf{F}(dom\ R)\ \} \\
 & (\mathsf{F}\ f) \cdot \mathsf{F}(dom\ S) \subseteq \mathsf{F}\ S \\
\equiv & \quad \{\ \text{relacionadores comutam com a composição}\ \} \\
 & \mathsf{F}(f \cdot dom\ S) \subseteq \mathsf{F}\ S \\
\Leftarrow & \quad \{\ \text{relacionadores são monótonos}\ \} \\
 & f \cdot dom\ S \subseteq S \\
\equiv & \quad \{\ \text{definição}\ \} \\
 & S \vdash f
\end{array}
$$

Refinamento por passos

Estenda-se f em $S \vdash f$ a uma relação

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

Obs.:

- a condição $dom\ S \subseteq dom\ R$ assegura que as implementações podem apenas ser **mais definidas**
- a condição $R \cdot dom\ S \subseteq S$ assegura que as implementações podem apenas ser **mais determinísticas**
- Note que $\bot \vdash R$ continua válido, mas, em geral, para que $\top \vdash R$, é necessário que R seja **inteira**, já que $dom\ \top = id$.

Exemplo

Seja $S_{\nu,\epsilon}$ a especificação

$$sqrt\ (x : real)\ r : real$$
$$pre\ abs(x) \leq \nu$$
$$post\ abs(r^2 - x) \leq \epsilon$$

Então, sempre que $\nu_1 \leq \nu_2$ e $\epsilon_1 \geq \epsilon_2$,

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

No "limite", $\cdots \vdash S_{\infty,0} = sq^\circ$
Ora $sq^\circ \vdash f$, onde $f\ x = +\sqrt{x}$ ou $f\ x = -\sqrt{x}$.

Refinamento é uma ordem parcial

Reflexividade: $\vdash \subseteq id$, i.e.

$$S \vdash S$$

Transitividade: $\vdash \cdot \vdash \subseteq \vdash$, i.e.

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

Anti-simetria: $\vdash \cap \vdash^\circ \subseteq id$

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

monotonia-F:

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

Refinamento passo-a-passo

As leis de $\vdash$ tornam possível refinar uma especificação inicial, S, ao longo de vários passos,

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

cada qual introduzindo cada vez mais definição e/ou determinismo, e frequentemente resultando numa função (=algoritmo determinístico totalmente definido):

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

Que fazer depois de f?

$g \vdash f$ de novo

- Formalmente, $g \vdash f \equiv g = f$, ou seja, a especif. g é **extensionalmente** equivalente à implementação f.
- Mas há mais: em geral, olhamos f como sendo "mais **eficiente**" que g.
- A eficiência pode ser formalizada apenas na disciplina de **complexidade algorítmica** (fora do âmbito desta disciplina)
- Estudaremos leis funcionais que acrescentam eficiência e que generalizam regras bem conhecidas de geração e inter-combinação de ciclos (`while`).

Refinamento por fusão, etc

- Refinamento por inter-combinação de "ciclos sequenciais": regras de **fusão** e de **absorção**:

  "Desflorestação" — remoção de estruturas de dados intermédias

- Refinamento por inter-combinação de "ciclos paralelos": eliminação de **recursão mútua**:

  Veremos a propósito a lei de Fokkinga e o seu bem conhecido corolário, a lei "banana-split".

Eliminação de recursão mútua

Considere o seguinte par de funções mutuamente dependentes:

```
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);
```

Poderá alguma destas funções — p.e. g — ser convertida num ciclo while?
Em notação sem variáveis:

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

Duas equações em f e g

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

cf.

$$\mathbb{N}_0 \underset{in}{\overset{\cong}{\rightleftarrows}} \underbrace{1 + \mathbb{N}_0}_{\mathsf{F}\,\mathbb{N}_0}$$

$$in = [id, suc]$$

sendo tal que $\mathsf{F}\, f = id + f$. Assim, (por absorção-+) podemos escrever

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

A lei da recursão mútua

Esta situação é tratada pela chamada lei da recursão mútua, também chamada "lei de Fokkinga":

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

ou seja, em geral

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

Demonstração

$$\begin{array}{ll} & \langle f, g\rangle = (\!|\langle h, k\rangle|\!) \\ \equiv & \quad \{ \text{ universal-cata } \} \\ & \langle f, g\rangle \cdot in = \langle h, k\rangle \cdot \mathsf{F}\,\langle f, g\rangle \\ \equiv & \quad \{ \text{ fusão-}\times \text{ duas vezes (esq e dir)} \} \\ & \langle f \cdot in, g \cdot in\rangle = \langle h \cdot \mathsf{F}\,\langle f, g\rangle, k \cdot \mathsf{F}\,\langle f, g\rangle\rangle \\ \equiv & \quad \{ \text{ igualdade estrutural do "split" } \} \\ & \left\{ \begin{array}{l} f \cdot in = h \cdot \mathsf{F}\,\langle f, g\rangle \\ g \cdot in = k \cdot \mathsf{F}\,\langle f, g\rangle \end{array} \right. \end{array}$$

Exemplo

Seja $h = [id, \pi_2]$ e $k = [\underline{1}, +]$ do exemplo acima:

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

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

Exemplo

Dado que $fg = \langle f, g \rangle$, obtemos $g = \pi_2 \cdot fg$. Por outro lado, é fácil extrair g de

```
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);
```

como a função de Fibonacci padrão:

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

Resumo: calculámos $\pi_2 \cdot fg$ como uma versão **linear** da Fibonacci ($g = \pi_2 \cdot fg$).

Corolário: "banana-split" (1)

Considere a função que calcula a média duma lista não-vazia de números naturais:

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

sum e $length$ são ambas catamorfismos sobre $\mathbb{N}^+$:

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

A função avg faz duas travessias independentes pela lista argumento antes de ser feita a divisão $(/)$. Poderemos evitá-lo? A "banana-split" fundirá essas duas travessias.

Corolário: "banana-split" (2)

Seja $h = i \cdot \mathsf{F}\,\pi_1$ e $k = j \cdot \mathsf{F}\,\pi_2$ da lei de recursão mútua. Então

$$
\begin{array}{ll}
 & f \cdot in = (i \cdot \mathsf{F}\,\pi_1) \cdot \mathsf{F}\,\langle f, g\rangle \\
\equiv & \quad \{ \text{ a composição é associativa e F é um functor} \} \\
 & f \cdot in = i \cdot \mathsf{F}\,(\pi_1 \cdot \langle f, g\rangle) \\
\equiv & \quad \{ \text{ por cancelamento-} \times \} \\
 & f \cdot in = i \cdot \mathsf{F}\,f \\
\equiv & \quad \{ \text{ por cancelamento-cata} \} \\
 & f = (\!|i|\!)
\end{array}
$$

Corolário: "banana-split" (3)

De igual modo, de $k = j \cdot \mathsf{F}\,\pi_2$ segue-se que $g = (\!|j|\!)$. Então, pela lei da recursão mútua, temos

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

ou seja

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

Esta lei fornece-nos uma ferramenta muito útil para inter-combinação de **ciclos "paralelos"**: os "ciclos" $(\!|i|\!)$ e $(\!|j|\!)$ são fundidos num único "ciclo" $(\!|(i \times j) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2\rangle|\!)$.

Generalidade da "banana-split"

A banana-split funde duas travessias ("ciclos") sobre uma estrutura de dados, no sentido **geral**. Por exemplo,

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

faz ainda sentido no caso de árvores binárias de folhas, para

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

Novamente, sum e $length$ podem ser fundidas (bi-recursivamente).

Refinamento total de dados

Refinamento simultâneo de algoritmos e de dados: dadas

- uma especificação $B \xleftarrow{S} A$
- uma função de abstracção $A \xleftarrow{F_1} C$
- uma relação de representação $D \xleftarrow{R_2} B$

então diremos que $C \xleftarrow{I} D$ implementa S sse

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

Análise da equação

- A equação de refinamento acima vai ser resolvida em ordem a I (a incógnita) e apresentará, em geral, mais do que uma solução.
- $S \vdash F_1 \cdot I \cdot R_2$ significa

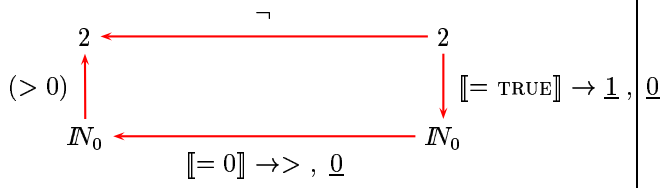
$$F_1 \cdot I \cdot R_2 \cdot dom\, S \subseteq S \quad \wedge \quad dom\, S \subseteq dom\, (F_1 \cdot I \cdot R_2)$$

- Caso $F_i = R_i = id$ ($i = 1, 2$ reduzir-se-á ao refinamento algorítmico:

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

pois não há refinamento de dados envolvido.

Exemplo



recordar

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

Note-se como o não-determinismo da implementação é resolvido pela não-injectividade de (> 0).

Em VDM

Especificação:

```
s(b) == not b;
```

Abstracção e representação:

```
f1(n) == n > 0;
r2(b) == if b then 1 else 0;
```

Implementação:

```
I(a:nat) r:nat
post a = 0 and r > 0 or a > 0 and r = 0;
```

Resolução de equações de refinamento

Uma vez que $dom\,(S \cdot R) = dom\,(dom\,S \cdot R)$, o segundo ponto acima rescreve-se como

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

Caso F_1 (f_1) seja inteira:

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

Caso a especificação S e F_1 (f_1) sejam inteiras e $R_2 = f_2^\circ$, I será inteira e tal que

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

Soluções funcionais

Caso em que todas as entidades numa equação de refinamento são funções totais (notem-se as minúsculas):

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

- Exemplo: $i = f^\star$ implementará $s = \mathcal{P} f$ sob o refinamento de dados $f_1 = f_2 = elems$.
- $i = f^\star$ não é solução única. Estas surgirão sempre que f_1 seja um iso (f_1° é função):

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

  Isto apela-nos a calcular i usando fusão-cata sobre o tipo indutivo da implementação, D.

Exemplo

Conjuntos como refinamento de listas:

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

($f_1 = id$, $f_2 = elems$):

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

Como $elems = (\!|ins|\!)$ e a função a obter são catas de listas $(a\ pertence) = (\!|\beta|\!)$, por fusão-cata a equação de refinamento será verdadeira se for verdadeiro $(a \in) \cdot ins = \beta \cdot (id + id \times (a \in))$.

Exemplo (cont.)

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

- Uma vez que $a \in \emptyset = \text{FALSE}$, calculamos $\beta_1 = \underline{\text{FALSE}}$.
- Ficamos com

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

  De $a \in \{x\} \cup s = (a \in \{x\}) \vee (a \in s)$, inferimos $\beta_2(x, b) \equiv a = x \vee b$.

- Enfim:

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

Cálculo de ciclos `while/for`

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

> *J.N. Oliveira. Operation refinement, Junho de 2000. Departamento de Informática, Universidade do Minho. Capítulo de livro em preparação.*