Chapter 5

Operation Refinement

5.1 Introduction

Transformational algorithmic design (vulg. program calculation) involves the following steps:

1. Calculation of a concrete-level *simulation* of the specification, *i.e.* of the high-level operation we want to realize (or “get rid of abstraction functions”).

2. Calculation of an efficient version of such a simulation (or “change pattern recursion”).

3. Encoding of 2 in a concrete programming language (or “get rid of mathematics altogether!”).

5.1.1 Step 1 (simulation)

In general, let

$$\sigma : A \longrightarrow B$$

be a morphism specifying some operation involving data-types A and B. Suppose we have already calculated the representation A and B (data refinement):

$$\begin{array}{ccc} A & \xrightarrow{\sigma} & B \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle g} \\ A_1 & & B_1 \end{array}$$

The idea is to “close” this diagram,

$$\begin{array}{ccc} A & \xrightarrow{\sigma} & B \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle g} \\ A_1 & \xrightarrow[\sigma_1]{} & B_1 \end{array} \qquad (5.1)$$

so that σ_1 "simulates" the behaviour of σ at concrete-level,

$$g \cdot \sigma_1 = \sigma \cdot f \tag{5.2}$$

that is, the diagram commutes. We regard σ_1 as the "unknown" of equation (5.2). Note that there may be many solutions for σ_1 in equation (5.2) — the typical "one-abstract to many-concrete" relationship of (functional) refinement.

"Simulation" amounts to exhibiting the same observational behaviour:

$$\sigma\, a \stackrel{\mathrm{def}}{=} \begin{array}{ll} let & a_1 \in \{a' \in A_1 \mid f\, a' = a\} \\ in & g(\sigma_1\, a_1) \end{array}$$

The choice of a particular range representation function s (*i.e.* such that $g \cdot s = id$) will determine a particular solution:

$$\sigma_1 \quad = \quad s \cdot \sigma \cdot f \tag{5.3}$$

For injective g this choice is unique: $s = g^{-1}$. Then the idea is to calculate further

$$\begin{array}{lcl} \sigma_1 & = & s \cdot \sigma \cdot f \\ & = & \ldots \\ & \vdots & \\ & = & \ldots \sigma_1 \ldots \quad \text{/*}\textit{expression free of } f \textit{ and } g \text{ */} \end{array}$$

thus obtaining a (possibly recursive) simulation σ_1 which is no longer defined in terms of abstraction/representation functions. It is easy to see that (5.3) can be obtained by fusion laws such as catamorphism fusion (2.61) wherever f and g are expressed by catamorphisms.

In general, there may exist more than one solution and idea is to transform refinement equation (5.2) so that f eventually gives place to g and g is eventually "pulled" up to the outermost place on the right-hand side,

$$\begin{array}{lcl} g \cdot \sigma_1 & = & \sigma \cdot f \\ & = & \ldots \\ & \vdots & \\ & = & \ldots g \ldots \sigma_1 \ldots \\ & \vdots & \\ & = & g \cdot E(\ldots \sigma_1 \ldots) \end{array}$$

so that abstraction map g can be factored out from both sides of the obtained equation:

$$\sigma_1 \quad \stackrel{\mathrm{def}}{=} \quad E(\ldots \sigma_1 \ldots)$$

(Of course, there may exist another E' such that $g \cdot E'(\ldots \sigma_1 \ldots)$ can also be obtained by alternative calculation.)

5.1.2 Step 2 (algorithmic refinement)

This is the step concerned with "efficiency". This can be obtained in many ways which are dependent on the target machine architecture:

- Change of virtual data-structure, or changing the algorithmic control:
  - left-lists ($\mathsf{F}\,X = 1+X\times A$) lead to $\mathcal{O}(1)$ space complexity (vulg. recursion-removal transformations targetted at synthesizing `for/while`-loops from recursive equations.)
  - binary-trees ($\mathsf{F}\,X = 1 + A \times X^2$) lead to $\mathcal{O}(log\ n)$ time complexity (*e.g.* the 'quicksort' implementation of insertion sort).
  - (monoid) accumulations trim $\mathcal{O}(n^2)$ time complexity down to $\mathcal{O}(n)$ time complexity.
  - vital rôle of exponentials!
- 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 (for this purpose we will see Fokkinga's law and its well-known corollary, the "banana-split" law)

5.1.3 Step 3 (code generation)

Code generation consists of carrying "program" transformation even further,

$$\begin{array}{rcl} & \vdots & \\ \sigma_1 & = & \ldots E \ldots \\ & = & \ldots \\ & \vdots & \\ & = & [\![\mathsf{P}]\!] \end{array}$$

until the formal semantics $[\![\mathsf{P}]\!]$ of some executable piece of code P (in the target programming language) is found.

5.2 Examples of simulation calculation (step 1)

The most obvious illustration of step 1 is the simulation of functorial operations implicit in the abstraction function naturality condition (polymorphism), *e.g.*the "sets represented by lists" data refinement:

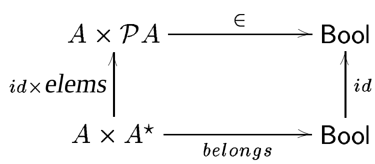
$$\begin{array}{ccccc} A & \qquad & A^\star & \xrightarrow{elems_A} & \mathcal{P}A \\ {\scriptstyle f}\downarrow & & {\scriptstyle f^\star}\downarrow & & \downarrow{\scriptstyle \mathcal{P}f} \\ B & & B^\star & \xrightarrow[elems_B]{} & \mathcal{P}B \end{array} \tag{5.4}$$

It is easy to see in this diagram an instance of diagram (5.1) — just rotate it anti-clockwise — for $\sigma = \mathcal{P} f$ and $\sigma_1 = f^\star$.

Let us now see a simple example of simulation calculation involving cata-fusion. Consider the fairly standard refinement equation

$$belongs(a, y) = a \in \mathit{elems}\, y \tag{5.5}$$

relative to the diagram of the "find" operation in the same "sets represented by lists" refinement:

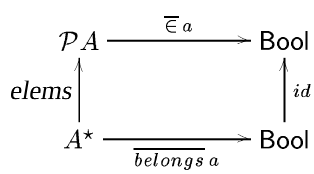


$$belongs = \in \cdot (id \times \mathit{elems})$$

We want to calculate $belongs$. We start by currying (5.5) on the first parameter:

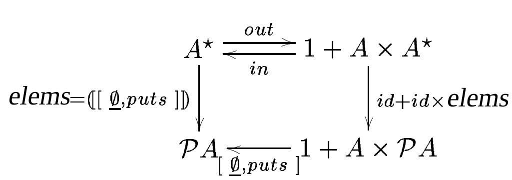
$$\begin{array}{ll} & belongs = \in \cdot (id \times \mathit{elems}) \\ \equiv & \quad \{ \text{ by (1.64) } \} \\ & \overline{belongs}\, a = \overline{\in \cdot (id \times \mathit{elems})}\, a \\ \equiv & \quad \{ \text{ by } \overline{f \cdot (g \times h)}\, a = ((\overline{f} \cdot g)\, a) \cdot h \ \} \\ & \overline{belongs}\, a = ((\overline{\in} \cdot id)\, a) \cdot \mathit{elems} \\ \equiv & \quad \{ \text{ identity } \} \\ & \overline{belongs}\, a = (\overline{\in}\, a) \cdot \mathit{elems} \end{array}$$

In a diagram:



$$\overline{belongs}\, a = (\overline{\in}\, a) \cdot \mathit{elems} \tag{5.6}$$

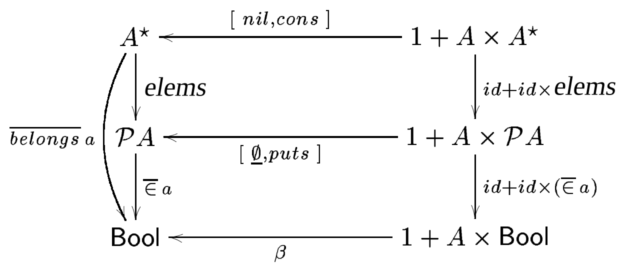
Recall that $\mathit{elems} = (\!|[\, \underline{\emptyset}, puts\,]|\!)_{\mathsf{F}}$, where $puts = \cup \cdot (\mathit{sings} \times id)$, $\mathit{sings}\, a = \{a\}$ (singleton set) and $\mathsf{F}\, X = 1 + A \times X$:



So, equation (5.6) is an opportunity to apply cata-fusion (2.61), by switching unknown from $belongs$ to β such that

$$\overline{belongs}\, a = (\!|\beta|\!)$$

holds, provided the bottom square below commutes:



Let $\beta = [\ \beta_1, \beta_2\ ]$. The detailed reasoning is as follows:

$$
\begin{array}{cl}
 & (\overline{\in}\, a) \cdot (\!|[\ \underline{\emptyset}, puts\ ]|\!) = (\!|\beta|\!) \\
\Leftarrow & \qquad \{ \text{ by cata-fusion (2.61)} \} \\
 & (\overline{\in}\, a) \cdot [\ \underline{\emptyset}, puts\ ] = \beta \cdot (id + id \times (\overline{\in}\, a)) \\
\equiv & \qquad \{ \text{ by } + \text{-}\mathit{fusion} \text{ (1.40)}, \beta = [\ \beta_1, \beta_2\ ] \text{ and } + \text{-absorption (1.41)} \} \\
 & [\ (\overline{\in}\, a) \cdot \underline{\emptyset}, (\overline{\in}\, a) \cdot puts\ ] = [\ \beta_1, \beta_2 \cdot (id \times (\overline{\in}\, a))\ ] \\
\equiv & \qquad \{ \text{ by } f \cdot \underline{c} = \underline{f\ c} \text{ and definition of } puts\ \} \\
 & \left\{ \begin{array}{l} \beta_1 = \underline{(\overline{\in}\, a)\, \emptyset} \\ \beta_2 \cdot (id \times (\overline{\in}\, a)) = (\overline{\in}\, a) \cdot \cup \cdot (\mathit{sings} \times id) \end{array} \right. \\
\equiv & \qquad \{ \text{ uncurrying and } a \in x \cup y = (a \in x) \vee (a \in y)\ \} \\
 & \left\{ \begin{array}{l} \beta_1 = \underline{a \in \emptyset} \\ \beta_2 \cdot (id \times (\overline{\in}\, a)) = \vee \cdot ((\overline{\in}\, a) \times (\overline{\in}\, a)) \cdot (\mathit{sings} \times id) \end{array} \right. \\
\equiv & \qquad \{ \times\text{-functor} \} \\
 & \left\{ \begin{array}{l} \beta_1 = \underline{\text{FALSE}} \\ \beta_2 \cdot (id \times (\overline{\in}\, a)) = \vee \cdot ((\overline{\in}\, a) \cdot \mathit{sings} \times (\overline{\in}\, a)) \end{array} \right. \\
\equiv & \qquad \{ \text{ by } a \in \{b\} = (a = b) \text{, that is, } (\overline{\in}\, a) \cdot \mathit{sings} = \overline{=}\, a\ \} \\
 & \left\{ \begin{array}{l} \beta_1 = \underline{\text{FALSE}} \\ \beta_2 \cdot (id \times (\overline{\in}\, a)) = \vee \cdot ((\overline{=}\, a) \times (\overline{\in}\, a)) \end{array} \right. \\
\equiv & \qquad \{ \text{ by introduction of two } id\ \} \\
 & \left\{ \begin{array}{l} \beta_1 = \underline{\text{FALSE}} \\ \beta_2 \cdot (id \times (\overline{\in}\, a)) = \vee \cdot ((\overline{=}\, a) \times id) \cdot (id \times (\overline{\in}\, a)) \end{array} \right. \\
\Leftarrow & \qquad \{ \text{ note the implication sign } \}
\end{array}
$$

$$\begin{cases} \beta_1 = \underline{\text{FALSE}} \\ \beta_2 = \vee \cdot ((\equiv a) \times id) \end{cases}$$

So, by cata-fusion (2.61) we have obtained

$$\overline{belongs}\, a = (\![\, \underline{\text{FALSE}}, \vee \cdot ((\equiv a) \times id)\,]\!) \tag{5.7}$$

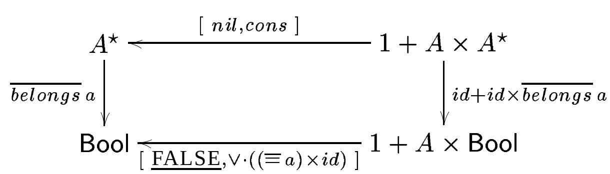
that is, going pointwise and uncurried:

$$belongs(a, y) \stackrel{\text{def}}{=} \begin{cases} y = [\,] & \Rightarrow \;\text{FALSE} \\ y \neq [\,] & \Rightarrow \; a = hd\, y \vee belongs(a, tl\, y) \end{cases} \tag{5.8}$$

In summary, we have achieved the purpose of step 1: $belongs$ is defined in terms of itself (abstraction/representation functions have vanished).

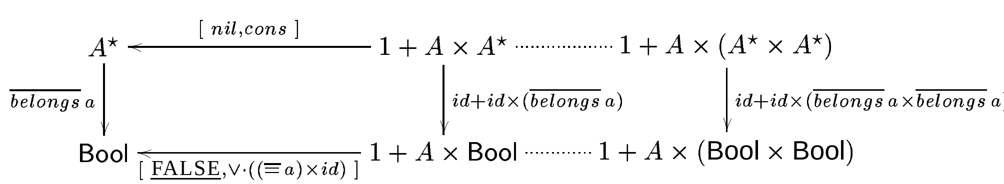
5.3 Algorithmic refinement: changing the virtual data-structure

The $belongs$ catamorphism (5.7,5.8) calculated in the previous section is described by diagram



and implements a linearly recursive search. Thus it is not particularly efficient. Moreover, the search will not stop once a is found in the list.

Linear search can be converted into the more efficient bilinear search by changing the pattern of recursion from lists to binary trees ($\mathsf{F}\, X = 1 + A \times X^2$), leading to $\mathcal{O}(log\ n)$ time complexity:



This change, which must not lose or mix information (it must be injective) is the same used in the 'quicksort' implementation of insertion sort:

$$id + part$$

where

$$
\begin{array}{lcll}
part(a,l) & \stackrel{\rm def}{=} & let & r = [x \mid x \leftarrow l \wedge x \leq a] \\
 & & & l = [x \mid x \leftarrow l \wedge x > a] \\
 & & in & (a,(r,l))
\end{array}
$$

Note that $id + part$ is in fact injective: it has $id + id \times (+\!\!+)$ as a left-inverse. So we are led to

$$
\begin{array}{ccccc}
A^\star & \xleftarrow{[\,nil, cons\,]} & 1 + A \times A^\star & \xleftarrow{id+id\times(+\!\!+)} & 1 + A \times (A^\star \times A^\star) \\
\Big\downarrow {\scriptstyle \overline{belongs\ a}} & & \Big\downarrow {\scriptstyle id+id\times(\overline{belongs\ a})} & & \Big\downarrow {\scriptstyle id+id\times(\overline{belongs\ a}\times\overline{belongs\ a})} \\
\mathsf{Bool} & \xleftarrow[{[\,\underline{\rm FALSE}, \vee\cdot((\equiv a)\times id)\,]}]{} & 1 + A \times \mathsf{Bool} & \xleftarrow[\alpha]{} & 1 + A \times (\mathsf{Bool} \times \mathsf{Bool})
\end{array}
$$

where α must be chosen so that the righthand side square commutes. It is easy to see that

$$\alpha \stackrel{\rm def}{=} id + id \times \vee$$

is a good choice, since

$$belongs(a, r +\!\!+ l) \quad \Leftrightarrow \quad belongs(a,r) \vee belongs(a,l)$$

All in all, we get the following bilinear version of $belongs$ (we omit the conversion from pointfree to pointwise notation):

$$
\begin{array}{lclll}
bbelongs(a,l) & \stackrel{\rm def}{=} & if\ \ y == [\,] & & \\
 & & then\ \ \mathrm{FALSE} & & \\
 & & else\ \ a = hd\, l \,\vee & let & r = [x \mid x \leftarrow tl\, l \wedge x \leq a] \\
 & & & & t = [x \mid x \leftarrow tl\, l \wedge x > a] \\
 & & & in & bbelongs(a,r) \vee bbelongs(a,t)
\end{array}
$$

This solution will run in logarithmic time at the cost of extra dynamic storage (binary recursion). Another drawback is that it still does not stop once a is found. In the next section we will see how to go in the opposite direction, *i.e.* by removing recursion and generating a `while` loop.

5.4 Left-linear recursion: calculation of `while/for` loops

We will be concerned with efficiency and want to eliminate recursion in (5.8). Let $\mathsf{Bool} \xleftarrow{beloop} A \times (A^\star \times \mathsf{Bool})$ be any function satisfying the following axiom:

$$beloop(x,(y,b)) = b \vee belongs(x,y) \tag{5.9}$$

i.e.

$$(\overline{beloop}\, x)(y, b) = b \vee (\overline{belongs}\, x) y$$

Let $beloop_x$ and $belongs_x$ abbreviate $\overline{beloop}\, x$ and $\overline{belongs}\, x$, respectively. It is easy to check that $beloop$ extends $belongs$ in the following way:

$$belongs_x\, y \quad = \quad beloop_x(y, \text{FALSE})$$

since algebraic structure

$$(\{\text{TRUE}, \text{FALSE}\}; \vee, \text{FALSE})$$

is a monoid. Moreover, for $y = [\,]$, we have

$$\begin{array}{rcl} beloop_x([\,], b) & = & b \vee belongs_x\, [\,] \\ & = & b \vee \text{FALSE} \\ & = & b \end{array} \tag{5.10}$$

as well as, for $y \neq [\,]$,

$$\begin{array}{rcl} beloop_x(y, b) & = & b \vee (x = hd\, y \vee belongs_x(tl\, y)) \\ & = & (b \vee (x = hd\, y)) \vee belongs_x(tl\, y) \\ & = & beloop_x(tl\, y, (b \vee (x = hd\, y))) \end{array} \tag{5.11}$$

cf. (5.9). Putting (5.10) and (5.11) together, we obtain

$$beloop_x(y, b) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} y = [\,] & \Rightarrow & b \\ y \neq [\,] & \Rightarrow & beloop_x(tl\, y, b \vee (x = hd\, y)) \end{array} \right. \tag{5.12}$$

Furthermore, we know that TRUE is the “zero” of $\vee$:

$$b \vee \text{TRUE} = \text{TRUE} \vee b = \text{TRUE}$$

So, via (5.9),

$$beloop_x(y, \text{TRUE}) = \text{TRUE}$$

i.e.

$$b \Rightarrow (beloop_x(y, b) = b)$$

We take advantage of this in the follow up of (5.12):

$$beloop_x(y, b) \stackrel{\text{def}}{=} \left\{ \begin{array}{rcl} y = [\,] \vee b & \Rightarrow & b \\ y \neq [\,] \wedge \neg b & \Rightarrow & beloop_x(tl\, y, (x = hd\, y)) \end{array} \right. \tag{5.13}$$

Finally, we put (5.10) together with (5.13) to obtain the denotational semantics of, respectively, the initialization and body of the following `while`-loop,

```
{  bool  found = 0;
   list  p;
   {      p = y;
          while   ((p != <>) && ¬ found)
                  { found = (x == head(p));        (5.14)
                    p = tail(p)
                  };
   }
}
```

encoded here in a kind of 'ad hoc' imperative C-like syntax. (At this level one may prefer `found` to `b` for easier perception of the meaning of the loop.)

Programming variables such as `found` are sometimes regarded as programming *tricks* produced by the intuition of able programmers. From the reasoning above we see that, rather than tricky, they have a sound *mathematical basis*, which is a consequence of the formal properties of the operators involved in their specification and calculation processes [1]. In a distributed/parallel environment, rules will go in the opposite way: the more recursive the better (*e.g.* double factorial instead of linear factorial).

This apparently academic exercise can be generalized to a vast **class** of algorithm specifications (in a sense, every programming loop "has" a hidden mathematical structure of this kind), as is shown next.

Generalization

In the general case, let

$$\begin{array}{rcl} f & : & \mathsf{F}Y \longrightarrow M \\ f & \stackrel{\rm def}{=} & p \rightarrow u, \theta \cdot \langle d, f \cdot e \rangle \end{array} \tag{5.15}$$

be an arbitrary function where the following abstract operators occur,

$$\begin{array}{rcl} p & : & \mathsf{F}Y \longrightarrow \mathsf{Bool} \\ e & : & \mathsf{F}Y \longrightarrow \mathsf{F}Y \\ d & : & \mathsf{F}Y \longrightarrow M \\ \theta & : & M \times M \longrightarrow M \\ u & : & 1 \longrightarrow M \end{array}$$

such that $(M; \theta, u)$ is a monoid, that is,

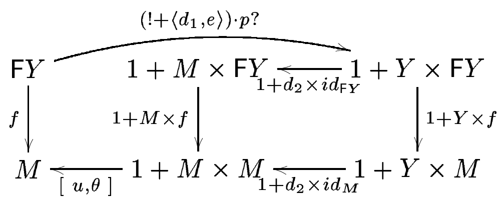
$$\theta \cdot \langle i, u \rangle = \theta \cdot \langle u, i \rangle = i$$
$$\theta \cdot \langle \theta \cdot \langle i, j \rangle, k \rangle = \theta \cdot \langle i, \theta \cdot \langle j, k \rangle \rangle$$

hold for arbitrary $\cdots \xrightarrow{i,j,k} M$.

Comments:

[1] See the program transformation literature for other rules and schemata of this kind, useful to eliminate undesired recursion [1, 2].

- Patterns such as (5.15) are called *program schemata*. Schema (5.15), which is called the *linear monadic schema* (LMS), is nothing but the following abstract hylomorphism,



  for $d = d_2 \cdot d_1$.

- Check that $belongs_x$ matches this schema for

LMS	$belongs_x$
Y	A
$\mathsf{F}Y$	$A^\star$
M	Bool
p	$=_{[\,]}$
e	*tl*
d_1	*hd*
d_2	$=_x$
θ	$\vee$
u	FALSE

Wishing to generalize our calculation of $belongs$ to f, we introduce a function

$$\mathsf{F}Y \times M \xrightarrow{floop} M$$

defined by

$$floop(..y..,r) \stackrel{\text{def}}{=} r\ \theta\ f(..y..) \tag{5.16}$$

which, because $(M;\theta,u)$ is a monoid, will satisfy:

$$f(..y..) = floop(..y..,u) \tag{5.17}$$

The following reasoning is a consequence of (5.16) and of the monoidal properties of $(M;\theta,u)$:

$$\begin{array}{rcl} floop(..y..,r) & = & r\ \theta\ f(..y..) \\ & = & r\ \theta \left\{ \begin{array}{rcl} p(..y..) & \Rightarrow & u \\ \neg(p(..y..)) & \Rightarrow & d(..y..)\ \theta\ f(..e\,y..) \end{array} \right. \\ & = & \left\{ \begin{array}{rcl} p(..y..) & \Rightarrow & r\ \theta\ u \\ \neg p(..y..) & \Rightarrow & r\ \theta\ (d(..y..)\ \theta\ f(..e\,y..)) \end{array} \right. \\ & = & \left\{ \begin{array}{rcl} p(..y..) & \Rightarrow & r \\ \neg p(..y..) & \Rightarrow & \underbrace{(r\ \theta\ d(..y..))\ \theta\ f(..e\,y..)}_{A} \end{array} \right. \end{array}$$

By instantiation of (5.16) we get

$$A = floop(..e\,y.., r\;\theta\; d(..y..))$$

In summary, we have

$$floop(..y.., r) \quad = \quad \begin{cases} p(..y..) & \Rightarrow \quad r \\ \neg p(..y..) & \Rightarrow \quad floop(..e\,y.., r\;\theta\; d(..y..)) \end{cases}$$

which, together with the "loop initialization" given by (5.17), matches the semantics of the following "while-loop schema",

```
{  M       r = u;
   Y       y' = y;
   while   (¬ p(..y'..))
           { r = r θ d(..y'..);
             y' = e(y')
           };
}
```

Should M have a "zero", that is some $a \in M$ such that

$$a\,\theta\,m = m\,\theta\,a = a$$

holds, further specialization of $floop$ becomes available:

$$\begin{aligned} floop(..y.., r) \quad &= \quad \begin{cases} p(..y..) & \Rightarrow \quad r \\ \neg p(..y..) & \Rightarrow \quad \begin{cases} r = a & \Rightarrow \quad a \\ r \neq a & \Rightarrow \quad floop(..e\,y.., r\;\theta\; d(..y..)) \end{cases} \end{cases} \\ &= \quad \begin{cases} p(..y..) \vee r = a & \Rightarrow \quad r \\ \neg p(..y..) \wedge r \neq a & \Rightarrow \quad floop(..e\,y.., r\;\theta\; d(..y..)) \end{cases} \end{aligned}$$

leading to something we can identify as the semantics of the following "while-loop schema",

```
{  M       r = u;
   Y       y' = y;
   while   (¬ p(..y'..)  && r!= a)
           { r = r θ d(..y'..);
             y' = e(y')
           };
}
```

of which that of $belongs$ (5.14) is an obvious instance.

5.4.1 Examples — Set and List Browsing

Recall Zermelo-Frænkl set-comprehension formula,

$$\{g\,x \mid \; x \in X \wedge p\,x\} \tag{5.18}$$

and list comprehension,

$$[g\,x \mid x \leftarrow L \wedge p\,x\,] \tag{5.19}$$

given $A \xrightarrow{g} B$, $X \subseteq A$, $L \in A^*$ and "filter" $A \xrightarrow{p} \mathsf{Bool}$.
It can be shown (*e.g.* by induction) that

- hylomorphism

$$\begin{array}{ccccc}
 & \multicolumn{3}{c}{\overset{(!+gets)\cdot =_\emptyset ?}{\longrightarrow}} & \\
2^A & & 1+2^B \times 2^A & \xleftarrow[1+d_2 \times id_{2^A}]{} & 1+A\times 2^A \\
\downarrow f & & \downarrow 1+2^B \times f & & \downarrow 1+A\times f \\
2^B & \xleftarrow[[\,\underline{\emptyset},\cup\,]]{} & 1+2^B \times 2^B & \xleftarrow[1+d_2 \times id_{2^B}]{} & 1+A\times 2^B
\end{array}$$

  where

$$d_2 = p \rightarrow sings \cdot g, \underline{\emptyset}$$

  for $sings\,x \stackrel{\rm def}{=} \{x\}$ — that is, recursive function

$$f\,y \stackrel{\rm def}{=} \left\{ \begin{array}{lcll}
y = \emptyset & \Rightarrow & \emptyset & \\
y \neq \emptyset & \Rightarrow & let & e \in y \\
 & & & a = \left\{ \begin{array}{ccc} p\,e & \Rightarrow & \{g\,e\} \\ \neg(p\,e) & \Rightarrow & \emptyset \end{array} \right. \\
 & & in & a \cup f(y - \{e\})
\end{array} \right.$$

  when applied to X, yields the same set as ZF-formula above;

- hylomorphism

$$\begin{array}{ccccc}
 & \multicolumn{3}{c}{\overset{(!+\langle hd,tl\rangle)\cdot =_{[\,]} ?}{\longrightarrow}} & \\
A^\star & & 1+B^\star \times A^\star & \xleftarrow[1+d_2 \times id_{A^\star}]{} & 1+A\times A^\star \\
\downarrow f & & \downarrow 1+B^\star \times f & & \downarrow 1+A\times f \\
B^\star & \xleftarrow[[\,\underline{[\,]},+\!\!+\,]]{} & 1+B^\star \times B^\star & \xleftarrow[1+d_2 \times id_{B^\star}]{} & 1+A\times B^\star
\end{array}$$

  where

$$d_2 = p \rightarrow sinl \cdot g, \underline{[\,]}$$

for $sinl\, x \stackrel{\rm def}{=} [x]$ — that is, recursive function

$$f\, y \stackrel{\rm def}{=} \left\{ \begin{array}{lll} y = [\,] & \Rightarrow & [\,] \\ y \neq [\,] & \Rightarrow & \begin{array}{ll} let & x = hd\, y \\ & a = \left\{ \begin{array}{lll} p\, x & \Rightarrow & [g\, x] \\ \neg(p\, x) & \Rightarrow & [\,] \end{array} \right. \\ in & a + f(tl\, y) \end{array} \end{array} \right.$$

when applied to L, yields the same list as formula (5.19) above.

Moreover, both recursive functions above are instances of linear-monadic scheme (5.15). So... they "are" loops!

Simple instantiation of (5.15) will show that second f above, for instance, is realized by the following abstract "`while`-loop",

```
{  B* r = [];
   A* y' = y;
   A x;
   while (y' != [])
       { x = head(y');
         y'= tail(y');
         if p(x) r = conc(r,g(x));
       }
}
```

or, being more "C-oriented" in parameterizing the function in terms of `stdin` and `stdout` and assuming suitable 'i/o' library functions `getA` and `putB`, by

```
{
   A x;
   while ((x = getA(stdin)) != EOF)
         { if p(x) putB(g(x)); }
}
```

Of course, a similar `while`-loop schema can be developed for (5.18).

Note how "fine-grained" these schemata look like when compared to relational model information browsing, and how they render "one-at-a-time" *data browsing* explicit.

A lot of programming practice can be captured by schema results such as above. Experience in this kind of calculation develops the ability to spot *while loops* and to write them straight away — one just has to search for the closest linear monadic schema around...

Recursive schemata which are not linear monadic may actually be *converted* into that form by program calculation. Laws such as the one presented next can be used for this purpose.

5.5 The mutual-recursion law

Let us consider the following pair of mutually dependent functions over $\mathbb{N}_0$ (written in the CAMILA notation):

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

Can any of these funtions — say g — be converted into a while loop? In pointfree notation we have

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

The mutual dependence can be made more explicit by forcing

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

The underlying inductive type is

$$
\mathbb{N}_0 \underset{in = [\,\underline{0}, suc\,]}{\cong} \underbrace{1 + \mathbb{N}_0}_{\mathsf{F}\,\mathbb{N}_0} \tag{5.20}
$$

which is such that $\mathsf{F}\, f = id + f$. So we can write

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

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

$$
\begin{array}{c}
f \cdot in = h \cdot \mathsf{F}\,\langle f, g \rangle \\
\wedge \\
g \cdot in = k \cdot \mathsf{F}\,\langle f, g \rangle
\end{array}
\quad \Rightarrow \quad \langle f, g \rangle = (\!|\langle h, k \rangle|\!) \tag{5.21}
$$

In terms of diagrams: from

$$
\begin{array}{ccc}
\mathsf{T} & \xleftarrow{in} & \mathsf{F}\,\mathsf{T} \\
{\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,\langle f,g \rangle} \\
A & \xleftarrow[h]{} & \mathsf{F}\,(A \times B)
\end{array}
\qquad
\begin{array}{ccc}
\mathsf{T} & \xleftarrow{in} & \mathsf{F}\,\mathsf{T} \\
{\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,\langle f,g \rangle} \\
B & \xleftarrow[k]{} & \mathsf{F}\,(A \times B)
\end{array}
$$

we get

$$
\begin{array}{ccc}
\mathsf{T} & \xleftarrow{in} & \mathsf{F}\,\mathsf{T} \\
{\scriptstyle \langle f,g \rangle}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,\langle f,g \rangle} \\
A \times B & \xleftarrow[\langle h,k \rangle]{} & \mathsf{F}\,(A \times B)
\end{array}
$$

Proof:

$$\begin{array}{ll} & \langle f, g \rangle \cdot in = \langle h, k \rangle \cdot \mathsf{F}\, \langle f, g \rangle \\ \equiv & \quad \{ \text{ by } \times\text{-fusion (1.24)} \} \\ & \langle f, g \rangle \cdot in = \langle h \cdot \mathsf{F}\, \langle f, g \rangle, k \cdot \mathsf{F}\, \langle f, g \rangle \rangle \\ \equiv & \quad \{ \text{ by hypothesis} \} \\ & \langle f, g \rangle \cdot in = \langle f \cdot in, g \cdot in \rangle \\ \equiv & \quad \{ \text{ by (reverse) } \times\text{-fusion (1.24)} \} \\ & \langle f, g \rangle \cdot in = \langle f, g \rangle \cdot in \\ \equiv & \quad \{ \text{ equality is reflexive} \} \\ & \text{TRUE} \end{array}$$

We can apply this law to the situation above by letting $h = [\,\underline{0}, \pi_2\,]$ and $k = [\,\underline{1}, +\,]$ therefore obtaining

$$\begin{array}{ll} & \langle f, g \rangle \\ = & \quad \{ \text{ Fokkinga law} \} \\ & (\!|\langle [\,\underline{0}, \pi_2\,], [\,\underline{1}, +\,] \rangle|\!) \\ = & \quad \{ \text{ exchange law} \} \\ & (\!|[\, \langle \underline{0}, \underline{1} \rangle, \langle \pi_2, + \rangle \,]|\!) \end{array}$$

which is CAMILA function

```
fg(n) = if n == 0 then <0,1>
        else let (p=fg(n.-1)
             in <p2(p),p1(p).+p2(p)>;
```

i.e.

```
fg(n) = do(a<-0,
           b<-1,
           while(~(n==0),
                 c<-a, a<-b,
                 b<-c .+ b,
                 n<-n.-1),
           <a,b>);
```

Since

$$g = \pi_2 \cdot \langle f, g \rangle$$

one has

```
g(n) = do(a<-0,
          b<-1,
          while(~(n==0),
                c<-a,
                a<-b,
                b<-c .+ b,
                n<-n.-1),
          b);
```

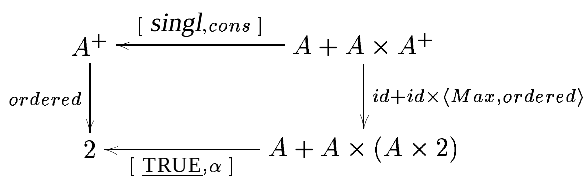
which is nothing but an iterative version of Fibonacci.

Another Example

Checking a list-invariant which ensures that a (non-empty) list is ordered:

$$ordered : A^+ \longrightarrow 2$$
$$ordered\,[a] = \text{TRUE}$$
$$ordered\,(cons(a,l)) = a > (Max\,l) \wedge (ordered\,l)$$

Assuming *singl* $a = [a]$ we can depict $ordered$ as follows:



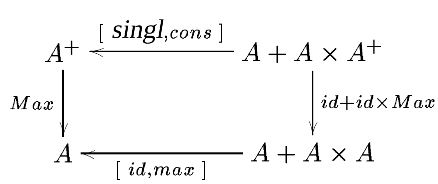
where

$$\alpha(a,(m,b)) \stackrel{\text{def}}{=} a > m \wedge b$$

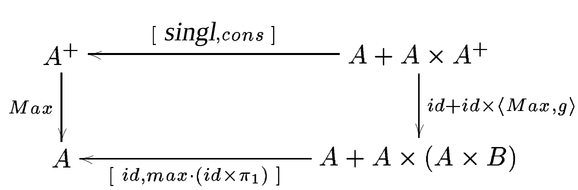
and where

$$Max = (\!|[\, id, max \,]|\!)$$

cf.



It is easy to check that the equation implicit in this diagram is the same as the one implicit in



for any $A^+ \xrightarrow{g} B$. For $B = 2$ and $g = ordered$ we are in position to apply Fokkinga's law and to obtain:

$$\begin{aligned} \langle Max, ordered \rangle &= (\!|\langle [\, id, max \cdot (id \times \pi_1)\,], [\, \underline{\text{TRUE}}, \alpha\,] \rangle|\!) \\ &= \qquad \{\ \textit{exchange law}\ (1.47)\} \\ &\quad\ (\!|[\ \langle id, \underline{\text{TRUE}} \rangle, \langle max \cdot (id \times \pi_1), \alpha \rangle\ ]|\!) \end{aligned}$$

Of course, $ordered = \pi_2 \cdot \langle Max, ordered \rangle$. Calling aux to the above synthesized catamorphism, we end up with the following realization of $ordered$:

$$\begin{aligned} ordered\, l \quad = \quad & let \quad (a, b) = aux\, l \\ & in \quad\ b \end{aligned}$$

where

$$\begin{aligned} & aux : \ A^+ \longrightarrow A \times 2 \\ & ordered\, [a] = (a, \text{TRUE}) \\ & ordered\, (cons(a, l)) = \ let \quad (m, b) = aux\, l \\ & \qquad\qquad\qquad\qquad\quad\ in \quad\ (max(a, m), (a > m \wedge b)) \end{aligned}$$

5.6 "Banana-split": a corollary of the mutual-recursion law

Let $h = i \cdot \mathsf{F}\, \pi_1$ and $k = j \cdot \mathsf{F}\, \pi_2$ in (5.21). Then

$$\begin{aligned} & f \cdot in = (i \cdot \mathsf{F}\, \pi_1) \cdot \mathsf{F}\, \langle f, g \rangle \\ \equiv & \qquad \{\ \text{composition is associative and } \mathsf{F} \text{ is a functor} \} \\ & f \cdot in = i \cdot \mathsf{F}\, (\pi_1 \cdot \langle f, g \rangle) \\ \equiv & \qquad \{\ \text{by } \times\text{-cancellation (1.20)} \} \\ & f \cdot in = i \cdot \mathsf{F}\, f \\ \equiv & \qquad \{\ \text{by cata-cancellation} \} \\ & f = (\!|i|\!) \end{aligned}$$

Similarly, from $k = j \cdot \mathsf{F}\, \pi_2$ we get

$$g = (\!|j|\!)$$

Then, from (5.21), we get

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

that is

$$\langle (\!|i|\!), (\!|j|\!) \rangle = (\!|(i \times j) \cdot \langle \mathsf{F}\, \pi_1, \mathsf{F}\, \pi_2 \rangle|\!) \tag{5.22}$$

by (reverse) $\times$-absorption (1.25).

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" $(\!|(i \times j) \cdot \langle \mathsf{F}\,\pi_1, \mathsf{F}\,\pi_2 \rangle|\!)$. The need for this kind of calculation arises very often. Consider, for instance, 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:

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

Function $average$ will do two independent traversals of the argument list before division $(/)$ takes place. Banana-split fuses such two traversals into a single one, thus leading to a function which: (a) runs twice as fast (b) can be converted into a *while loop* by introduction of accummulation parameters (such as seen above).

5.7 Exercises

Exercise 5.1 Isomorphism

$$\mathbb{N} \cong 1^\star$$

in exercise 4.6 suggests a strange (but valid) representation for natural numbers: number n is represented by list $\underbrace{[NIL, \ldots, NIL]}_{n \text{ vezes}}$. Suppose that you want to implement, over such a model, natural number addition and (partial) subtraction.

1. Identify which list-operations simulate such operations by drawing the corresponding refinement diagram.
2. Describe the behaviour of your simulation for subtraction $n - m$ wherever $m > n$.

□