An Introduction to Data Refinement

Formal Methods II, 2002/03

J.N. Oliveira

FM software design process

- Formal specification — “what” the intended software system should do
- Implementation — machine code produced instructing the hardware about “how” to do it

In general, there is more than one way in which a particular machine can accomplish “what” the specifier bore in mind:

- Relationship between specifications and implementations is **one-to-many**
- Specifications are more abstract than implementations.

Data refinement

Principle of **data abstraction**: A abstracts B wherever

- A surjective abstraction function $A \xleftarrow{F} B$ can be found:

$$\mathit{img}\, F = id \tag{1}$$

  F is thus **simple** but possibly partial.

- Any **entire** subrelation R of F° is said to be a representation for F. So $R \subseteq F^\circ$.

Representation relations

- It follows that R is **injective**, since *ker* $R \subseteq$ *ker* F° and *ker* $F^\circ =$ *img* $F = id$.
- So, no two different abstract values $a, a' \in A$ get mixed up along the representation process.
- Altogether, *ker* $R = id$ because $id \subseteq$ *ker* R (R is entire).
- It also follows that R is a **right-inverse** of F, that is

$$F \cdot R \;=\; id \tag{2}$$

This is proved by circular inclusion

$$F \cdot R \subseteq id \subseteq F \cdot R$$

in the next slide.

Right invertibility

$$
\begin{array}{ll}
 & F \cdot R \subseteq id \wedge id \subseteq F \cdot R \\
\equiv & \qquad \{ \; \textit{img}\ F = id \text{ and converses } \} \\
 & F \cdot R \subseteq F \cdot F^{\circ} \wedge id \subseteq R^{\circ} \cdot F^{\circ} \\
\equiv & \qquad \{ \; \textit{ker}\ R = id \} \\
 & F \cdot R \subseteq F \cdot F^{\circ} \wedge R^{\circ} \cdot R \subseteq R^{\circ} \cdot F^{\circ} \\
\Leftarrow & \qquad \{ \; (F \cdot) \text{ and } (R^{\circ} \cdot) \text{ are monotone } \} \\
 & R \subseteq F^{\circ} \wedge R \subseteq F^{\circ} \\
\equiv & \qquad \{ \; R \subseteq F^{\circ} \text{ is assumed } \} \\
 & \text{TRUE}
\end{array}
$$

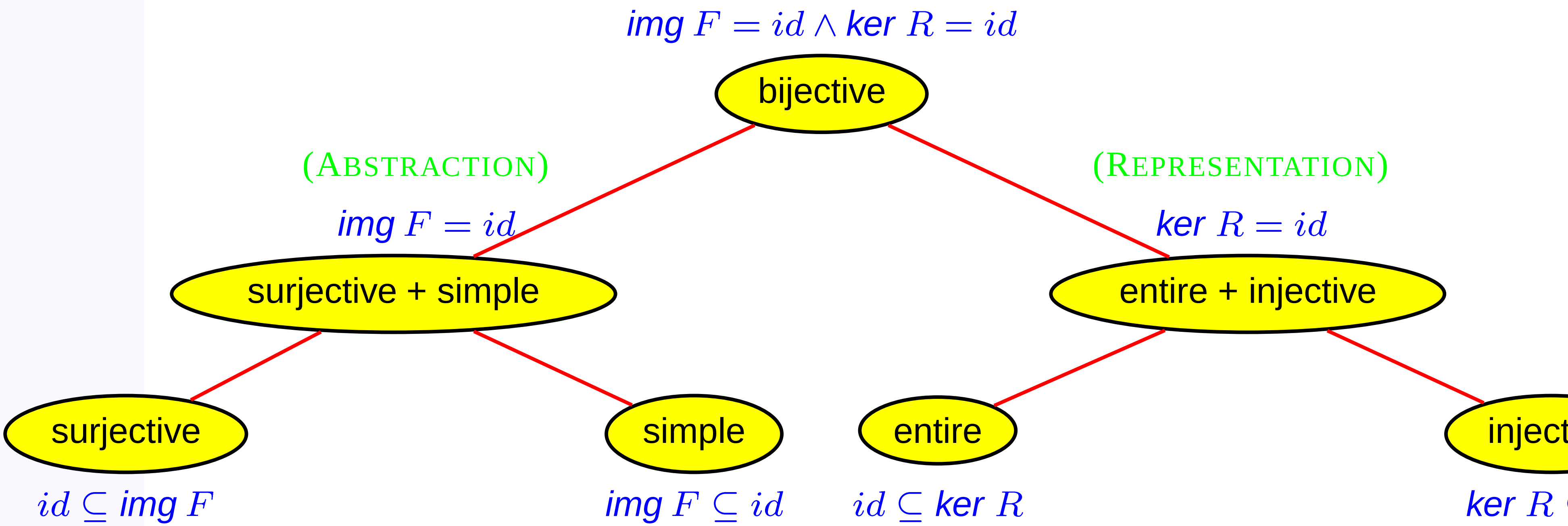
Refinement inequations

$$A \underset{F}{\overset{R}{\rightleftarrows}} B \quad (\leq) \qquad \text{such that} \quad F \cdot R = id_A$$

This inequation has several informal interpretations:

- A is "smaller" than B
- B is able to "represent" A
- B is "abstracted" by A
- A is "implemented" by B
- B is a refinement ("refines") A

In a diagram



img F = id ∧ ker R = id
bijective
(ABSTRACTION)
img F = id
surjective + simple
(REPRESENTATION)
ker R = id
entire + injective
surjective
id ⊆ img F
simple
img F ⊆ id
entire
id ⊆ ker R
inject
ker R

Example

Representing finite **sets** by finite **lists**:

$F =$ *elems*

$$\begin{tikzcd}
 & {[1,2]} \arrow[dl] \\
\{1,2\} & {[2,1]} \arrow[l] \\
 & {[1,2,1]} \arrow[ul]
\end{tikzcd}$$

Among the many $R \subseteq F^{\circ}$, we may choose the following:

Relational representation

```
Listify : set of nat -> seq of nat
Listify(s) ==
      if s = {} then []
                 else let e in set s
                      in [e] ^ Listify(s \ {e});
```

Intuitively,

$$\text{rng}\, Listify = [\![noRepeats]\!]$$

where

```
noRepeats(s) == card elems s = len s
```

Functional representation

```
listify : set of nat -> seq of nat
listify(s) ==
      if s = {} then []
                else let e = minset(s)
                     in [e] ^ listify(s \ {e});
```

Intuitively,

$$\textit{rng}\; listify = [\![IsOrdered]\!] \cdot [\![noRepeats]\!]$$

Concrete invariants

- Wherever

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] \end{tikzcd}$$

such that $R \subseteq F^\circ$ and *rng* $R = [\![\phi]\!]$

we say that ϕ is the **concrete invariant** induced by R.

- In case R is a function, and because it always is injective, one has

$$A \;\; \tilde{=} \;\; B_\phi$$

where B_ϕ denotes the subset of B which satisfies concrete-invariant ϕ.

Example of a partial abstraction

Every element of datatype A can be represented by a "pointer":

$$\begin{tikzcd} A \arrow[r, bend left, "i_1"] \arrow[r, phantom, "\leq"] & A+1 \arrow[l, bend left, "i_1^\circ"] \end{tikzcd}$$

- Simplicity of the abstraction is ensured by a known fact: the converse of an injective relation is simple.
- Concrete invariant: $\phi = [\underline{\mathsf{TRUE}}, \underline{\mathsf{FALSE}}]$

Another partial abstraction

Finite mappings “are” (simple) finite relations:

$$\begin{tikzcd} \mathtt{map}\ A\ \mathtt{to}\ B \arrow[rr, bend left, "mkr"] \arrow[rr, phantom, "\leq"] & & \mathtt{set\ of}\ (A * B) \arrow[ll, bend left, "mkf = mkr^{\circ}"] \end{tikzcd}$$

VDM-SL:

```
mkr : map A to B -> set of (A * B)
mkr(f) == { mk_(a,f(a)) | a in set dom f };

mkf : set of (A * B) -> map A to B
mkf(r) == { p.#1 |-> p.#2 | p in set r }
pre isSimple(r);
```

(Guess the concrete invariant.)

A foundamental iso abstraction

$$
\begin{tikzcd}
A \rightharpoonup B \arrow[r, bend left, "tot"] \arrow[r, phantom, "\stackrel{\sim}{=}"] & (B+1)^A \arrow[l, bend left, "untot"]
\end{tikzcd}
\tag{3}
$$

where, for types A, B and `JustB::value:B`,

```
tot: map A to B -> A -> [JustB]
tot(sigma)(a) ==
    if a in set dom(sigma) then mk_JustB(sigma(a)) else nil;

untot: (A -> [JustB]) -> map A to B
untot(f) == { a |-> b | a: A, b: B & f(a) = mk_JustB(b) };
```

Pointfree $untot = (i_1^\circ \cdot)$

As checked next:

$$
\begin{array}{ll}
 & untot\ f = i_1^\circ \cdot f \\
\equiv & \{ \text{ relations as set comprehensions} \} \\
 & untot\ f = \{(b, a) \mid a \in A, b \in B : b(i_1^\circ \cdot f)a\} \\
\equiv & \{ \text{ using rule } (f\ b)Ra \equiv b(f^\circ \cdot R)a \ \} \\
 & untot\ f = \{(b, a) \mid a \in A, b \in B : i_1\, b = f\ a\} \\
\equiv & \{ \text{ VDM-SL notation} \} \\
 & untot\ f = \texttt{\{a|->b|a:A,b:B \& f(a)=mk\_JustB(b)\}}
\end{array}
$$

Easy consequence of $tot/untot$:

$$
\begin{tikzcd} A \arrow[r, bend left] \arrow[r, phantom, "\doteq"] & A^1 \arrow[l, bend left] \end{tikzcd}
$$

extends to partial functions as follows:

$$
\begin{tikzcd} A+1 \arrow[r, bend left, "r"] \arrow[r, phantom, "\doteq"] & 1 \rightharpoonup A \arrow[l, bend left, "f"] \end{tikzcd}
$$

(guess f and r).

That is, the "singleton" finite map is a disguise of a "pointer" structure.

Properties of $\leq$:

Reflexivity

$$\begin{tikzcd} A \arrow[r, bend left, "id"] \arrow[r, phantom, "\leq"] & A \arrow[l, bend left, "id"] \end{tikzcd} \quad \textit{cf.} \quad id \cdot id = id$$

Transitivity

$$\begin{tikzcd} A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"] \arrow[r, phantom, "\wedge"] & B \arrow[r, bend left, "S"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "G"] \arrow[r, phantom, "\Rightarrow"] & A \arrow[r, bend left, "S \cdot R"] \arrow[r, phantom, "\leq"] & C \arrow[l, bend left, "F \cdot G"] \end{tikzcd}$$

Proof of transitivity

It is enough to show that composition preserves simplicity and surjectiveness:

$$
\begin{array}{ll}
 & \textit{img}\,(F \cdot G) = id \\
\equiv & \quad \{ \text{ expanding and converses} \} \\
 & F \cdot (\textit{img}\, G) \cdot F^\circ = id \\
\equiv & \quad \{ \; G \text{ is simple and surjective} \} \\
 & \textit{img}\, F = id \\
\equiv & \quad \{ \; F \text{ is simple and surjective} \} \\
 & id = id
\end{array}
$$

Also note that $S \cdot R \subseteq (F \cdot G)^\circ$ by monotonicity.

Structural data refinement

$$
\begin{tikzcd}
A \arrow[r, bend left, "R"] \arrow[r, phantom, "\leq"] & B \arrow[l, bend left, "F"]
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
\mathsf{F}\,A \arrow[r, bend left, "\mathsf{F}\,R"] \arrow[r, phantom, "\leq"] & \mathsf{F}\,B \arrow[l, bend left, "\mathsf{F}\,F"]
\end{tikzcd}
$$

where F is an arbitrary relator (functor):

$$
\begin{array}{ll}
 & id \\
= & \quad \{ \text{ functors commute with } id \} \\
 & \mathsf{F}\,id \\
= & \quad \{ \ R \text{ is right-inverse of } F \} \\
 & \mathsf{F}\,(F \cdot R) \\
= & \quad \{ \text{ functors commute with composition} \} \\
 & (\mathsf{F}\,F) \cdot (\mathsf{F}\,R)
\end{array}
$$

Refining finite sets (I)

$$\mathcal{P}A \quad \tilde{\leq} \quad A \rightharpoonup 1$$

Calculation:

$$\begin{array}{ll} & A \rightharpoonup 1 \\ \tilde{\leq} & \quad \{\ tot \text{ representation } \} \\ & (1+1)^A \\ \tilde{\leq} & \quad \{\text{ basic}\} \\ & 2^A \\ \tilde{\leq} & \quad \{\ 2^A \text{ is isomorphic to } \mathcal{P}A\ \} \\ & \mathcal{P}A \end{array}$$

Refining finite sets (Ia)

$$\mathcal{P}A \underset{dom}{\overset{set2fm}{\doteq\!\!\!\!\!\sim}} A \rightharpoonup 1$$

VDM-SL

```
set2fm : set of A -> map A to Nil
set2fm(s) == { a |-> nil | a in set s };
```

Pointfree

$$set2fm \stackrel{\mathrm{def}}{=} (!\cdot)$$

Right-invertibility

Calculation:

$$
\begin{array}{ll}
& \mathit{dom} \cdot set2fm = id \\
\equiv & \quad \{ \ \} \\
& \mathit{dom}\,(set2fm\ s) = s \\
\equiv & \quad \{ \ \} \\
& \mathit{dom}\,(! \cdot s) = s \\
\equiv & \quad \{\ !\text{ is a function, } \mathit{dom}\,(f \cdot R) = \mathit{dom}\,R \} \\
& \mathit{dom}\,s = s \\
\equiv & \quad \{\ s \text{ is coreflexive} \} \\
& s = s
\end{array}
$$

Refining finite sets (II)

List (cf. example before):

$$
\begin{tikzcd}
\texttt{set of}\, A \arrow[rr, bend left] \arrow[r, phantom, "\leq"] & {} & \texttt{seq of}\, A \arrow[ll, bend left, "elems"]
\end{tikzcd}
$$

Index A:

$$
\begin{tikzcd}
\texttt{set of}\, A \arrow[rr, bend left] \arrow[r, phantom, "\leq"] & {} & \texttt{map nat to}\, A \arrow[ll, bend left, "rng"]
\end{tikzcd}
$$

Refining finite sets (III)

Classify A by B ($B \supset \{\}$):

$$\texttt{set of } A \underset{dom}{\overset{\longrightarrow}{\underset{\longleftarrow}{\leq}}} \texttt{map } A \texttt{ to } B$$

Quantify A ("multisets"):

$$\texttt{set of } A \underset{dom}{\overset{\longrightarrow}{\underset{\longleftarrow}{\leq}}} \texttt{map } A \texttt{ to nat}$$

Refining finite maps (I)

```
JustB::value:B;
JustC::value:C;
BorC = JustB | JustC ;
```

$$\begin{tikzcd}[column sep=huge] \mathtt{map}\ (BorC)\ \mathtt{to}\ A \arrow[r, bend left] \arrow[r, phantom, "\hat{\cong}"] & (\mathtt{map}\ B\ \mathtt{to}\ A) \times (\mathtt{map}\ C\ \mathtt{to}\ A) \arrow[l, bend left, "peither"] \end{tikzcd}$$

```
peither: (map B to A) * (map C to A) -> map BorC to A
peither(m,n) == { mk_JustB(b) |-> m(b) | b in set dom m} munion
                { mk_JustC(c) |-> n(c) | c in set dom n};
```

Refining finite maps (Ia)

$$
\begin{tikzcd}
(B+C) \rightharpoonup A \arrow[rr, bend left, "unpeither"] & \cong & (B \rightharpoonup A) \times (C \rightharpoonup A) \arrow[ll, bend left, "peither"]
\end{tikzcd}
$$

where

$$peither(\sigma, \tau) \;=\; [\sigma, \tau]$$

for $[R, S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ)$, that is

$$peither = \cup \cdot ((\cdot i_1^\circ) \times (\cdot i_2^\circ))$$

Refining finite maps (II)

$$\begin{tikzcd} A \rightharpoonup (B + C) \arrow[rr, bend left, "uncojoin"] & \leq & (A \rightharpoonup B) \times (A \rightharpoonup C) \arrow[ll, bend left, "cojoin"] \end{tikzcd}$$

where

$$cojoin = \cup \cdot ((i_1 \cdot) \times (i_2 \cdot))$$

NB: $cojoin$ is partial since the union of two partial functions not always is a partial function.

Refining finite maps (IIa)

Note the representation function:

```
uncojoin : map A to BorC -> (map A to B) * (map A to C)
uncojoin(f) ==
    mk_( { a |-> f(a).value
               | a in set dom f & is_JustB(f(a)) },
         { a |-> f(a).value
               | a in set dom f & is_JustC(f(a)) }
       );
```

The finite map bifunctor

- Note the $(\cdot i_1^\circ)$s, $(i_1 \cdot)$s, etc
- In general, for an injective f and any g, define bifunctor

$$f \rightharpoonup g \stackrel{\text{def}}{=} (g\cdot) \cdot (\cdot f^\circ)$$

that is

$$(f \rightharpoonup g)\sigma = g \cdot \sigma \cdot f^\circ$$

- So, we could have written *e.g.*

$$peither = \cup \cdot ((i_1 \rightharpoonup id) \times (i_2 \rightharpoonup id))$$

Refining finite maps (III)

$$\begin{tikzcd} \mathtt{map}\ A\ \mathtt{to}\ B * C \arrow[rr, bend left] & \leq & (\mathtt{map}\ A\ \mathtt{to}\ B) \times (\mathtt{map}\ A\ \mathtt{to}\ C) \arrow[ll, bend left, "\bowtie"] \end{tikzcd}$$

where (writing `join` for $\bowtie$)

```
join :(map A to B) * (map A to C) -> map A to (B * C)
join(m,n) == { a |-> mk_(m(a),n(a))
                | a in set dom m inter dom n };
```

Refining finite maps (IIIa)

$$\begin{tikzcd} A \rightharpoonup B \times C \arrow[r, phantom, "\leq"] \arrow[r, bend left, "unjoin"] & (A \rightharpoonup B) \times (A \rightharpoonup C) \arrow[l, bend left, "\bowtie"] \end{tikzcd}$$

where

$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \langle \sigma, \tau \rangle$$

where $\langle R, S \rangle \stackrel{\text{def}}{=} (\pi_1^\circ \cdot R) \cap (\pi_2^\circ \cdot S)$. A right-inverse of $join$ is

$$unjoin \stackrel{\text{def}}{=} \langle id \rightharpoonup \pi_1, id \rightharpoonup \pi_2 \rangle$$

Refining finite maps (IV)

How do we extend

$$
\begin{tikzcd}
B^{C \times A} \arrow[r, bend left, "curry"] \arrow[r, phantom, "\doteq"] & (B^A)^C \arrow[l, bend left, "uncurry"]
\end{tikzcd}
$$

to partial functions? Case $B := B + 1$

$$(B+1)^{C \times A} \doteq ((B+1)^A)^C$$

$$\equiv \qquad \{ \text{ that is } \}$$

$$(C \times A) \rightharpoonup B \doteq (A \rightharpoonup B)^C$$

Refining finite maps (IVa)

In general:

$$
\begin{tikzcd}
(C \times A) \rightharpoonup B \arrow[r, bend left, "pcurry"] \arrow[r, phantom, "\leq"] & C \rightharpoonup (A \rightharpoonup B) \arrow[l, bend left, "unpcurry"]
\end{tikzcd}
$$

```
unpcurry : map C to (map A to B) -> map (C * A) to B
unpcurry(f) ==
     merge { let g=f(a)
             in { mk_(a,b) |-> g(b) | b in set dom g }
           | a in set dom f };
```

Refining finite maps (IVb)

```
pcurry : map (C * A) to B -> map C to (map A to B)
pcurry(f) ==
      let y = { x.#1 | x in set dom f }
      in { a |-> { p.#2 |-> f(p)
                 | p in set dom f & p.#1=a }
         | a in set y };
```

Transposing relations

Let $B := 2$ in the $curry/uncurry$ isomorphism and obtain

$$\begin{tikzcd} \mathcal{P}(A \times C) \arrow[rr, bend left, "\Lambda"] \arrow[rr, phantom, "\cong"] & & (\mathcal{P}A)^C \arrow[ll, bend left, "\Lambda^\circ"] \end{tikzcd}$$

where

$$f = \Lambda R \;\equiv\; R = \in \cdot f \qquad (4)$$

and $A \xleftarrow{\in} \mathcal{P}A$ is the membership relation.

Transposing finite relations

$$\begin{tikzcd} \mathtt{set\ of}\,(C * A) \arrow[r, bend left, "collect"] \arrow[r, phantom, "\leq"] & \mathtt{map}\ C\ \mathtt{to\ set\ of}\ A \arrow[l, bend left, "discollect"] \end{tikzcd}$$

```
collect : set of (C * A) -> map C to set of A
collect(r) == { c |-> { q.#2 | q in set r & c=q.#1 }
                  | c in set { p.#1 | p in set r } };

discollect : map C to set of A -> set of (C * A)
discollect(f) == dunion { { mk_(c,a) | a in set f(c) }
                        | c in set dom f };
```

Refining finite maps (V)

Last but not least

$$
\begin{tikzcd}
A \rightharpoonup D \times (B \rightharpoonup C) \arrow[r, bend left, "unnjoin"] & (A \rightharpoonup D) \times ((A \times B) \rightharpoonup C) \arrow[l, bend left, "\bowtie_n"]
\end{tikzcd}
\qquad A \rightharpoonup D \times (B \rightharpoonup C) \;\leq\; (A \rightharpoonup D) \times ((A \times B) \rightharpoonup C) \tag{5}
$$

where

$$\bowtie_n \;\stackrel{\mathrm{def}}{=}\; \bowtie \cdot \langle \pi_1, \dagger \cdot ((id \rightharpoonup \underline{\emptyset}) \times pcurry) \rangle \tag{6}$$

and

$$unnjoin \;\stackrel{\mathrm{def}}{=}\; (id \times unpcurry) \cdot unjoin$$

Recursive data refinement

How does one refine recursive VDM-SL models such as *e.g.*

```
FS :: D: map Id to Node; -- FS means file system
Node = File | FS;        -- a Node is either a file
                         --   or a directory
Id = seq of char;        -- node identifiers
File :: F: seq of token  -- sequential files
```

that is, $FS = \mu\mathsf{F}$ for $\mathsf{F}\,X = Id \rightharpoonup (File + X)$:

$$
\begin{tikzcd}
\mu\mathsf{F} \arrow[rr, bend left, "out"] \arrow[rr, phantom, "\cong"] & & Id \rightharpoonup (File + \mu\mathsf{F}) \arrow[ll, bend left, "in"]
\end{tikzcd}
$$

Recursive data refinement

or . . .

```
DecTree :: question: What
           answers: map Answer to DecTree
What = seq of char;
Answer = seq of char;
```

that is, $DecTree = \mu$F in

$$DecTree \;\;\hat{\cong}\;\; What \times (Answer \rightharpoonup DecTree)$$

for F $X = What \times (Answer \rightharpoonup X)$

Recursion "removal"

Given

$$
\begin{tikzcd}
\mu\mathsf{F} \arrow[rr, bend left, "out"] \arrow[rr, phantom, "\stackrel{\sim}{=}"] & & \mathsf{F}\,\mu\mathsf{F} \arrow[ll, bend left, "in"]
\end{tikzcd}
$$

one has

$$
\begin{tikzcd}
\mu\mathsf{F} \arrow[rr, bend left] \arrow[rr, phantom, "\leq"] & & (K \rightharpoonup \mathsf{F}\,K) \times K \arrow[ll, bend left, "F"]
\end{tikzcd}
\tag{7}
$$

for K a domain of "pointers" such that $K \stackrel{\sim}{=} \mathbb{N}$.

Abstraction function

- Main rôle in representation is played by a (partial) F-coalgebra $\mathsf{F}\,K \xleftarrow{\sigma} K$, assumed as a (finite) piece of "linear storage", a "heap" or a "database".
- $\overline{F}$ (the transpose of abstraction F) is of type $(K \rightharpoonup \mathsf{F}\,K) \longrightarrow K \longrightarrow \mu\mathsf{F}$ and one can build hylomorphism

$$
\begin{tikzcd}
\mu\mathsf{F} & & K \arrow[ll, "\overline{F}\sigma"'] \arrow[d, "\sigma"] \\
\mathsf{F}(\mu\mathsf{F}) \arrow[u, "in"] & & \mathsf{F}\,K \arrow[ll, "\mathsf{F}(\overline{F}\sigma)"]
\end{tikzcd}
\qquad
\overline{F}\sigma = in \cdot \mathsf{F}(\overline{F}\sigma) \cdot \sigma
$$

Partiality of implementation

$F(\sigma, k) = (\overline{F}\sigma)k$ will be undefined wherever

- $k \notin$ *dom* σ
- σ is not “closed” over itself (see below)
- σ is non-well-founded (see below)

Thus concrete invariant

$$\phi(\sigma, k) \stackrel{\text{def}}{=} k \in \textit{dom}\, \sigma \wedge (closed\ \sigma) \wedge (wellf\ \sigma)$$

In order to define $closed\ \sigma$ and $wellf\ \sigma$ we need σ's accessibility relation $\prec_\sigma$ (next slide).

Accessibility and membership

Accessibility relation for σ:

$$K \xleftarrow{\prec_\sigma} K$$

$$\prec_\sigma \stackrel{\mathrm{def}}{=} \in_{\mathsf{F}} \cdot \sigma$$

where $K \xleftarrow{\in_{\mathsf{F}}} \mathsf{F}\, K$ extends $K \xleftarrow{\in} \mathcal{P} K$ inductively over polynomial functors, as follows:

$$\begin{array}{rcl}
\in_{\mathcal{P}} & \stackrel{\mathrm{def}}{=} & \in \\
\in_C & \stackrel{\mathrm{def}}{=} & \bot \\
\in_{\lambda X.X} & \stackrel{\mathrm{def}}{=} & id \\
\in_{\mathsf{F} \times \mathsf{G}} & \stackrel{\mathrm{def}}{=} & (\in_{\mathsf{F}} \cdot \pi_1) \cup (\in_{\mathsf{G}} \cdot \pi_2) \\
\in_{\mathsf{F} + \mathsf{G}} & \stackrel{\mathrm{def}}{=} & [\in_{\mathsf{F}}, \in_{\mathsf{G}}]
\end{array}$$

Example

Let F $X = 1 + A \times X$. Then,

$$
\begin{array}{ll}
& \in_{1+A\times X} \\
= & \{ \ \in \text{ for coproduct bifunctor } \} \\
& [\in_1, \in_{A\times X}] \\
= & \{ \ \in \text{ for constant and product (bi)functors } \} \\
& [\bot, (\in_A \cdot \pi_1) \cup (\in_{\lambda X.X} \cdot \pi_2)] \\
= & \{ \ \in \text{ for constant and identity functor } \} \\
& [\bot, (\bot \cdot \pi_1) \cup (id \cdot \pi_2)] \\
= & \{ \ \bot \text{ and } [R, S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \ \} \\
& \pi_2 \cdot i_2^\circ
\end{array}
$$

Example (pointfree)

$$
\begin{array}{ll}
& k \in_{1+A\times X} x \\
\equiv & \qquad \{ \text{ calculation above } \} \\
& k(\pi_2 \cdot i_2^\circ)x \\
\equiv & \qquad \{ \text{ relational composition } \} \\
& k(\pi_2)(a, k') \wedge x = i_2(a, k') \\
\equiv & \qquad \{ \text{ trivia } \} \\
& x = i_2(a, k') \wedge k = k' \\
\equiv & \qquad \{ \text{ trivia } \} \\
& x = i_2(a, k)
\end{array}
$$

Accessibility (example)

Pointer reachability in case of a “linear” heap

$(1 + A \times K) \xleftarrow{\sigma} K$:

$$k_1 \prec_\sigma k_2 \quad \equiv \quad k_2 \in \textsf{\textit{dom}}\, \sigma \wedge (\sigma\ k_2) = i_2(a, k_1)$$

In a drawing:

$$\begin{array}{cc} & \begin{array}{c||cc} K & A & K \end{array} \\ \sigma = & \left|\begin{array}{c||c|c} k_1 & \ldots & k_0 \\ \hline k_2 & a & k_1 \\ \hline & \ldots & \ldots \end{array}\right| \end{array} \qquad \begin{array}{l} k_1 \prec_\sigma k_2 \\ k_0 \prec_\sigma k_1 \end{array}$$

Closure and wellfoundedness

Let $\prec_\sigma^+$ denote the transitive closure of $\prec_\sigma$. Then,

- $closed\ \sigma = \mathit{rng}\ \prec_\sigma^+ \subseteq \mathit{dom}\ \sigma$ that is, all reacheable k are defined.
- $wellf\ \sigma = (\prec_\sigma^+) \cap id = \bot$, that is, $\prec_\sigma^+$ is irreflexive (no cycles)