Exercises in Coalgebraic Specification

Bart Jacobs

Department of Computer Science,

University of Nijmegen

Email: `bart@cs.kun.nl`

URL: `http://www.cs.kun.nl/~bart`

and also

URL: `http://www.cs.kun.nl/~bart/LOOP/`

Contents

The product of two sets A, B is written as

$$A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}$$

with first and second projections

$$A \xleftarrow[x \,\longleftarrow\!\shortmid\, (x,y)]{\pi} A \times B \xrightarrow[(x,y) \,\longmapsto\, y]{\pi'} B$$

For $f\colon C \longrightarrow A$ and $g\colon C \longrightarrow B$ there is a tuple function

$$C \xrightarrow{\langle f, g \rangle} A \times B \qquad \text{given by} \qquad z \longmapsto (f(z), g(z))$$

Bijective correspondence

Projecting and tupleing yield:

$$\frac{C \longrightarrow A \times B}{C \longrightarrow A \qquad C \longrightarrow B}$$

Functoriality

For $h\colon C \longrightarrow A$ and $k\colon D \longrightarrow B$ there is a product function

$$C \times D \xrightarrow{h \times k} A \times B \qquad \text{given by} \qquad (x, y) \longmapsto (h(x), k(y))$$

Note that: $h \times k = \langle h \circ \pi, k \circ \pi' \rangle$.

Empty product

We write $1 = \{*\}$ for a (chosen) singleton set.

It can be understood as the product of an empty collection of sets.

Properties

- For each set A, there is a unique function

$$A \dashrightarrow 1$$

- Functions

$$1 \longrightarrow A$$

  correspond to elements of A.

The coproduct (or disjoint union) of two sets A, B is written as

$$A + B = \{(x, 0) \mid x \in A\} \cup \{(y, 1) \mid y \in B\}$$

with coprojection (or injection) functions

$$A \xrightarrow[x \longmapsto (x,0)]{\kappa} A + B \xleftarrow[(y,1) \longleftarrow y]{\kappa'} B$$

For functions $f \colon A \longrightarrow C$ and $g \colon B \longrightarrow C$ there is a cotuple

$$A + B \xrightarrow{[f,g]} C \qquad \text{given by} \qquad \begin{cases} (x,0) \longmapsto f(x) \\ (y,1) \longmapsto g(y) \end{cases}$$

Often we shall use a CASES notation for $[f, g]$:

$$\begin{array}{rl} z \longmapsto & \mathsf{CASES}\ z\ \mathsf{OF} \\ & \quad \kappa x \mapsto f(x) \\ & \quad \kappa' y \mapsto g(y) \\ & \mathsf{ENDCASES} \end{array}$$

Bijective correspondence

Coprojecting and cotupleing yield:

$$\frac{A + B \longrightarrow C}{\overline{A \longrightarrow C \qquad B \longrightarrow C}}$$

Functoriality

For $h: C \longrightarrow A$ and $k: D \longrightarrow B$ there is a coproduct function

$$C + D \xrightarrow{\;h+k\;} A + B \qquad \text{given by} \qquad z \longmapsto \begin{array}{l} \text{CASES } z \text{ OF} \\ \quad \kappa x \mapsto \kappa(h(x)) \\ \quad \kappa' y \mapsto \kappa'(k(y)) \\ \text{ENDCASES} \end{array}$$

Thus: $h + k = [\kappa \circ h, \kappa' \circ k]$.

Note

Coprojections κ, κ' are injective and "disjoint": $\kappa a \neq \kappa' b$.

The exponent or function space of sets A, B is

$$B^A = \{f \colon A \to B\}.$$

Bijective correspondence ("Currying")

$$\frac{C \longrightarrow B^A}{C \times A \longrightarrow B}$$

Functoriality

For a fixed set A and $h \colon B \longrightarrow C$ there is an exponent function

$$B^A \xrightarrow{\;h^A\;} C^A \qquad \text{given by} \qquad f \longmapsto h \circ f.$$

An operation $X \longmapsto F(X)$ on sets is called functorial if it extends to functions.
Usually, the same notation $F(-)$ is used for this extension to functions.

Domains, codomains, identities and compositions should be preserved:

- For $f\colon X \longrightarrow Y$, $F(f)\colon F(X) \longrightarrow F(Y)$;
- $F(id_X) = id_{F(X)}$;
- $F(g \circ f) = F(g) \circ F(f)$.

This operation F is then called a functor.

Often the action of a functor on functions is "obvious", and only the action on sets is given.

Examples of functors

- The identity functor $X \longmapsto X$.
- For a fixed set A, the constant functor $X \longmapsto A$. It maps each function $f\colon X \longrightarrow Y$ to the identity $id_A\colon A \longrightarrow A$.
- For each set A, the operations

$$X \longmapsto A \times X \qquad \text{and} \qquad X \longmapsto A + X \qquad \text{and} \qquad X \longmapsto X^A$$

are functorial. The first two send $f\colon X \longrightarrow Y$ to

$$A \times X \xrightarrow{id_A \times f} A \times Y \qquad \text{given by} \qquad (a, x) \longmapsto (a, f(x))$$

and to

$$A + X \xrightarrow{id_A + f} A + Y \qquad \text{given by} \qquad \begin{cases} \kappa a \longmapsto \kappa a \\ \kappa' x \longmapsto \kappa'(f(x)) \end{cases}$$

These functors are denoted by $A \times (-)$, $A + (-)$ and $(-)^A$.
- The composition $X \mapsto G(F(X))$ of two functors F, G.

Polynomial functors include:

- The identity functor;
- The constant functors $X \longmapsto A$, for an arbitrary set A;
- The product $X \longmapsto T_1(X) \times T_2(X)$ of two polynomial functors T_1, T_2;
- The coproduct $X \longmapsto T_1(X) + T_2(X)$ of two polynomial functors T_1, T_2;
- The exponent $X \longmapsto T_1(X)^A$ for a polynomial functors T_1 and an arbitrary set A.

Example

$$X \longmapsto 1 + (A + X)^B + (X \times X)^C.$$

Let T be a polynomial functor. An algebra consists of a set A together with a function

$$T(A) \xrightarrow{a} A$$

A coalgebra consists of a set C with a function

$$C \xrightarrow{c} T(C)$$

Example

- The unit, multiplication, and inverse operation of a group G form an algebra

  $$1 + (G \times G) + G \longrightarrow G$$

  for the functor $T(X) = 1 + (X \times X) + X$.

- The head and tail operation on the set A^ω of infinite sequences of elements from a set A form a coalgebra

  $$A^\omega \longrightarrow A \times A^\omega$$

  for the functor $T(X) = A \times X$.

The functor is the interface of a (co)algebra.

In "Computer Science" style:

BEGIN AbelianGroup
OPERATIONS
$\mathsf{zero}\colon 1 \longrightarrow X$
$\mathsf{add}\colon X \times X \longrightarrow X$
$\mathsf{inv}\colon X \longrightarrow X$
ASSERTIONS
$\mathsf{unit}\colon [\ \forall x \in X.\ \mathsf{add}(x, \mathsf{zero}(*)) = x \wedge \mathsf{add}(\mathsf{zero}(*), x) = x\ ]$
$\mathsf{assoc}\colon [\ \forall x, y, z \in X.\ \mathsf{add}(x, \mathsf{add}(y, z)) = \mathsf{add}(\mathsf{add}(x, y), z)\ ]$
$\mathsf{inv}\colon [\ \forall x \in X.\ \mathsf{add}(x, \mathsf{inv}(x)) = \mathsf{zero}(*)\ ]$
$\mathsf{comm}\colon [\ \forall x, y \in X.\ \mathsf{add}(x, y) = \mathsf{add}(y, x)\ ]$
END AbelianGroup

AbelianGroup interface functor

$$X \longmapsto 1 + (X \times X) + X$$

So that models (i.e. Abelian groups) are algebras satisfying the assertions.

BEGIN VectorSpace[K: FIELD]
  EXTENDS AbelianGroup
  OPERATIONS
    scalar_mult: $K \times X \longrightarrow X$
  ASSERTIONS
    scalar_group_add: [$\forall a \in K.\, \forall x, y \in X.$
      scalar_mult$(a,$ add$(x, y))$ $=$ add(scalar_mult(a, x), scalar_mult(a, y))]
    scalar_field_add: [$\forall a, b \in K.\, \forall x \in X.$
      scalar_mult$(a + b, x)$ $=$ add(scalar_mult(a, x), scalar_mult(b, x))]
    scalar_field_mult: [$\forall a, b \in K.\, \forall x \in X.$
      scalar_mult$(a \cdot b, x)$ $=$ scalar_mult$(a,$ scalar_mult$(b, x))$]
    scalar_field_unit: [$\forall x \in X.$ scalar_mult$(1, x) = x$]
END VectorSpace

Interface functor: $X \longmapsto (1 + (X \times X) + X) + (K \times X)$.

BEGIN BinaryTreeSpace$[A\colon$ TYPE$]$
  METHODS
    label: $X \longrightarrow A$
    children: $X \longrightarrow 1 + (X \times X)$
    size: $X \longrightarrow 1 + \mathbb{N}$
  ASSERTIONS
    size_def: $[\ \forall x \in X.\ \mathsf{size}(x) =$
      CASES children(x) OF
        $\kappa u \mapsto \kappa' 1$
        $\kappa'(v_1, v_2) \mapsto$ CASES size(v_1) OF
          $\kappa u \mapsto \kappa *$
          $\kappa' n \mapsto$ CASES size(v_2) OF
            $\kappa u \mapsto \kappa *$
            $\kappa' m \mapsto \kappa'(1 + n + m)$
          ENDCASES
        ENDCASES
      ENDCASES $]$
END BinaryTreeSpace

Interface functor

$$\mathcal{BT}(X) = A \times (1 + (X \times X)) \times (1 + \mathbb{N})$$

Models

Coalgebras

$$X \xrightarrow{c \,=\, \langle \mathsf{label}, \mathsf{children}, \mathsf{size} \rangle} \mathcal{BT}(X)$$

satisfying the assertions of the specification.

Example models

State space $=$ finite binary A-labeled binary trees;
infinite ones;
finite plus infinite ones,
each with appropriate operations.

Bisimilarity

Coalgebras are dynamical systems with a black-box state space.

Equality of states is often not appropriate, but equality as far as we can observe is.

This is called bisimilarity, and written as $\underline{\leftrightarrow}$.

Bisimulation

Bisimilarity is defined as the greatest bisimulation.

A bisimulation for a coalgebra $c\colon X \longrightarrow T(X)$ is a relation $R \subseteq X \times X$ which is closed under c.

There are several ways to formalise this. We choose the logical approach (using induction on T), since it is most convenient in reasoning.

Definition

For a coalgebra $c\colon X \longrightarrow T(X)$ a relation $R \subseteq X \times X$ is a bisimulation if

$$(x, y) \in R \implies (c(x), c(y)) \in \mathsf{Rel}(T)(R)$$

where the relation lifting $\mathsf{Rel}(T)(R) \subseteq T(X) \times T(X)$ is defined by induction:

T	$\mathsf{Rel}(T)(R)(z, w)$ for $z, w : T(X)$
X	$R(z, w)$
C	$z =_C w$
$T_1 \times T_2$	$\mathsf{Rel}(T_1)(R)(\pi\, z, \pi\, w) \;\wedge\; \mathsf{Rel}(T_2)(R)(\pi'\, z, \pi'\, w)$
$T_1 + T_2$	$\begin{cases} \text{if } (z, w) = (\kappa\, z_1, \kappa\, w_1) & \mathsf{Rel}(T_1)(R)(z_1, w_1) \\ \text{if } (z, w) = (\kappa'\, z_2, \kappa'\, w_2) & \mathsf{Rel}(T_2)(R)(z_2, w_2) \\ \text{otherwise} & \perp \end{cases}$
T_1^A	$\forall a : A\,.\, \mathsf{Rel}(T_1)(R)(z(a), w(a))$

Bisimilarity

Definition
Consider two states $x, y \in X$ of a coalgebra $c\colon X \longrightarrow T(X)$.

$$x \mathrel{\underline{\leftrightarrow}} y \stackrel{\text{def}}{\Longleftrightarrow} \exists R.\, (R \text{ is a bisimulation for } c \;\wedge\; R(x, y))$$

Then:

- $\underline{\leftrightarrow}$ is the greatest bisimulation.
- bisimilar states are observationally indistinguishable

The definition of $\underline{\leftrightarrow}$ is fully determined by the interface type/functor of a specification. Therefore it is tailor-made for each class specification.

Recall the interface functor $\mathcal{BT}$ for binary trees:

$$\mathcal{BT}(X) = A \times (1 + (X \times X)) \times (1 + \mathbb{N})$$

For a coalgebra $c = \langle \mathsf{label}, \mathsf{children}, \mathsf{size} \rangle \colon X \longrightarrow \mathcal{BT}(X)$, a relation $R \subseteq X \times X$ is a bisimulation if

$$R(x, y) \Longrightarrow \begin{cases} \mathsf{label}(x) = \mathsf{label}(y) \\ \wedge \\ ((\mathsf{children}(x) = \mathsf{children}(y) = \kappa *) \vee \\ \quad (\mathsf{children}(x) = \kappa'(x_0, x_1) \wedge \mathsf{children}(y) = \kappa'(y_0, y_1) \wedge \\ \quad R(x_0, y_0) \wedge R(x_1, y_1))) \\ \wedge \\ \mathsf{size}(x) = \mathsf{size}(y) \end{cases}$$

$$
\begin{array}{l}
\textbf{BEGIN}\ \mathsf{BinaryTreeSpace}[A\colon \mathsf{TYPE}] \\
\quad \textbf{METHODS} \\
\qquad \mathsf{label}\colon X \longrightarrow A \\
\qquad \mathsf{children}\colon X \longrightarrow 1 + (X \times X) \\
\qquad \mathsf{size}\colon X \longrightarrow 1 + \mathbb{N} \\
\qquad \mathsf{mirror}\colon X \longrightarrow X \\
\quad \textbf{ASSERTIONS} \\
\qquad \mathsf{size\_def}\colon [\ \cdots\ ] \\
\qquad \mathsf{label\_mirror}\colon [\ \forall x \in X.\, \mathsf{label}(\mathsf{mirror}(x)) = \mathsf{label}(x)\ ] \\
\qquad \mathsf{children\_mirror}\colon [\ \forall x \in X. \\
\qquad\quad \mathsf{CASES}\ \mathsf{children}(x)\ \mathsf{OF} \\
\qquad\qquad \kappa u \mapsto \mathsf{children}(\mathsf{mirror}(x)) = \kappa * \\
\qquad\qquad \kappa'(v_1, v_2) \mapsto\ \mathsf{CASES}\ \mathsf{children}(\mathsf{mirror}(x))\ \mathsf{OF} \\
\qquad\qquad\qquad \kappa w \mapsto \mathsf{false} \\
\qquad\qquad\qquad \kappa'(z_1, z_2) \mapsto\ (z_1 \underline{\leftrightarrow} \mathsf{mirror}(v_2)) \wedge \\
\qquad\qquad\qquad\qquad (z_2 \underline{\leftrightarrow} \mathsf{mirror}(v_1)) \\
\qquad\qquad\quad \mathsf{ENDCASES} \\
\qquad\quad \mathsf{ENDCASES}\ ] \\
\textbf{END}\ \mathsf{BinaryTreeSpace}
\end{array}
$$

Lemma

1. $\mathsf{size}(\mathsf{mirror}(x)) = \mathsf{size}(x)$
2. $\mathsf{mirror}(\mathsf{mirror}(x)) \underline{\leftrightarrow} x$.

Proof of 2.

Show that

$$R = \{(x, y) \mid y \underline{\leftrightarrow} \mathsf{mirror}(\mathsf{mirror}(x))\}$$

is a bisimulation. Then $R \subseteq \underline{\leftrightarrow}$, i.e.

$$y \underline{\leftrightarrow} \mathsf{mirror}(\mathsf{mirror}(x)) \implies x \underline{\leftrightarrow} y$$

Take $y = \mathsf{mirror}(\mathsf{mirror}(x))$ and use that $\underline{\leftrightarrow}$ is reflexive. $\square$

Informally

In the theory of dynamical systems, an invariant is a predicate on the state space, which, once it holds, continues to hold no matter which operations are applied.

Briefly, all operations maintain an invariant.

Importance

- Invariants express safety properties of specifications.

  E.g. that a certain integer-valued attribute will always be non-negative.
- Invariants are used for the definition of $\Box$ and $\Diamond$.
- Invariants are needed for refinements.

For a coalgebra $c\colon X \longrightarrow T(X)$ a predicate $P \subseteq X$ is an invariant if

$$x \in P \implies c(x) \in \mathsf{Pred}(T)(P)$$

where the predicate lifting $\mathsf{Pred}(T)(P) \subseteq T(X)$ is defined by induction:

T	$\mathsf{Pred}(T)(P)(z)$ for $z : T(X)$
X	$P(z)$
C	$\top$
$T_1 \times T_2$	$\mathsf{Pred}(T_1)(P)(\pi\ z) \;\wedge\; \mathsf{Pred}(T_2)(P)(\pi'\ z)$
$T_1 + T_2$	$\begin{cases} \text{if } z = \kappa\, z_1 & \mathsf{Pred}(T_1)(P)(z_1) \\ \text{if } z = \kappa'\, z_2 & \mathsf{Pred}(T_2)(P)(z_2) \end{cases}$
T_1^A	$\forall x : A\ .\ \mathsf{Pred}(T_1)(P)(z(x))$

We consider the BinaryTree specification with mirroring. Its interface functor is:

$$X \longmapsto A \times (1 + (X \times X)) \times (1 + \mathbb{N}) \times X$$

For a coalgebra $c = \langle \mathsf{label}, \mathsf{children}, \mathsf{size}, \mathsf{mirror} \rangle$ of this functor, a predicate $P \subseteq X$ is an invariant if:

$$P(x) \Longrightarrow \begin{cases} (\mathsf{children}(x) = \kappa'(x_0, x_1) \Rightarrow P(x_0) \wedge P(x_1)) \\ \wedge \\ P(\mathsf{mirror}(x)) \end{cases}$$

Example

$$\mathsf{FinSize}(x) \Longleftrightarrow (\mathsf{size}(x) \neq \kappa *)$$

Properties

Proposition

1. Predicate lifting $\mathsf{Pred}(T) \colon \mathcal{P}(X) \longrightarrow \mathcal{P}(T(X))$ preserves arbitrary intersections.

   (Or, equivalently, it has a left Galois adjoint $\mathcal{P}(T(X)) \longrightarrow \mathcal{P}(X)$.)

2. Invariants are closed under $\wedge$ and $\forall$:
   - If P and Q are invariants then so is $P \wedge Q$.
   - If $P(i)$ is an invariant for each $i \in I$, then so is $\forall i \in I.\, P(i)$.

Suppose we wish to add a method which tells whether an element occurs in the unfolding of a tree:

$$
\begin{array}{l}
\textbf{BEGIN}\ \mathsf{BinaryTreeSpace}[A\colon \mathsf{TYPE}] \\
\quad \textbf{METHODS} \\
\qquad \mathsf{label}\colon X \longrightarrow A \\
\qquad \mathsf{children}\colon X \longrightarrow 1 + (X \times X) \\
\qquad \mathsf{size}\colon X \longrightarrow 1 + \mathbb{N} \\
\qquad \mathsf{elem}\colon X \times A \longrightarrow \{\mathsf{false}, \mathsf{true}\} \\
\quad \textbf{ASSERTIONS} \\
\qquad \mathsf{naieve\_elem\_def}\colon [\ \forall x \in X.\, \forall a \in A.\ \mathsf{elem}(x,a) \iff \\
\qquad\quad (\mathsf{label}(x) = a) \vee\ \mathsf{CASES}\ \mathsf{children}(x)\ \mathsf{OF} \\
\qquad\qquad\qquad\qquad\qquad \kappa y \mapsto \mathsf{false} \\
\qquad\qquad\qquad\qquad\qquad \kappa'(x_1, x_2) \mapsto \mathsf{elem}(x_1, a) \vee \mathsf{elem}(x_2, a) \\
\qquad\qquad\qquad\qquad\quad \mathsf{ENDCASES}\ ] \\
\textbf{END}\ \mathsf{BinaryTreeSpace}
\end{array}
$$

The set $A^{\{0,1\}^\star}$ of all infinite binary A-labelled trees forms a model with, for $x \in A^{\{0,1\}^\star}$,

$$\begin{aligned}
\mathsf{children}(x) &= \kappa'(\lambda\alpha \in \{0,1\}^\star.\, x(0\cdot\alpha),\ \lambda\alpha \in \{0,1\}^\star.\, x(1\cdot\alpha)) \\
\mathsf{label}(x) &= x(\langle\rangle) \\
\mathsf{elem}(x,a) &= \mathsf{true}\ !!!
\end{aligned}$$

What we would like to be able to define instead is:

$$\mathsf{elem}(x,a) \iff (\text{there is a successor tree } y \text{ of } x \text{ with } \mathtt{label}(y) = a)$$

That is:

$$\mathsf{elem}(x,a) \iff \Diamond(\{y \mid \mathsf{label}(y) = a\})(x)$$

Definition

With respect to a coalgebra $c\colon X \longrightarrow T(X)$, define for a prediate $P \subseteq X$,

$$(\Box P)(x) \overset{\text{def}}{\Longleftrightarrow} \exists Q.\, (Q \text{ is an invariant for } c \;\wedge\; Q \subseteq P \;\wedge\; Q(x))$$

And:

$$\Diamond P \;=\; \neg\Box\neg P.$$

Then:

- $\Box P$ is the greatest invariant contained in P. It is henceforth P.
- $\Diamond P$ holds for those states that are reachable from P, expressing eventually P.

The definitions of invariant, $\Box$ and $\Diamond$ are tailor-made for each class specification, because they are fully determined by the interface type/functor of the specification.

BinaryTree revisited

Now we can define elem: $X \times A \longrightarrow \{\mathsf{false}, \mathsf{true}\}$ as:

$$\mathsf{elem}(x, a) \iff \Diamond(\{y \mid \mathsf{label}(y) = a\})(x)$$

And prove:

$$\mathsf{elem}(\mathsf{mirror}(x), a) \iff \mathsf{elem}(x, a)$$

This definition is at the right level of abstraction. It would be precisely the same for ternary trees, with operation

$$\mathsf{children}\colon X \longrightarrow 1 + (X \times X \times X)$$

because $\Diamond$ would change accordingly, again with meaning: for some future state …

$$
\begin{array}{l}
\textbf{BEGIN}\ \mathsf{Queue}[A\colon \mathsf{TYPE}] \\
\quad \textbf{METHOD}\ \ \mathsf{size}\colon X \longrightarrow \mathbb{Z} \\
\qquad\qquad\qquad\ \ \mathsf{push}\colon X \times A \longrightarrow X \\
\qquad\qquad\qquad\ \ \mathsf{pop}\colon X \longrightarrow 1 + (A \times X) \\
\quad \textbf{ASSERTION} \\
\qquad \mathsf{size\_push}\colon [\ \forall x \in X.\, \forall a \in A.\, \mathsf{size}(\mathsf{push}(x,a)) = \mathsf{size}(x) + 1\ ] \\
\qquad \mathsf{size\_pop}\colon [\ \forall x \in X.\ \ \mathsf{CASES}\ \mathsf{pop}(x)\ \mathsf{OF} \\
\qquad\qquad\qquad\qquad\qquad\quad \kappa * \mapsto \mathsf{size} = 0 \\
\qquad\qquad\qquad\qquad\qquad\quad \kappa'(a,y) \mapsto \mathsf{size}(y) = \mathsf{size}(x) - 1 \\
\qquad\qquad\qquad\qquad\qquad \mathsf{ENDCASES}\ ] \\
\qquad \mathsf{pop\_push}\colon [\ \forall x \in X.\, \forall a \in A.\ \ \mathsf{CASES}\ \mathsf{pop}(\mathsf{push}(x,a))\ \mathsf{OF} \\
\qquad\qquad\qquad\qquad \kappa * \mapsto \mathsf{false} \\
\qquad\qquad\qquad\qquad \kappa'(b,y) \mapsto\ \ \mathsf{CASES}\ \mathsf{pop}(x)\ \mathsf{OF} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \kappa * \mapsto b = a \wedge y \underline{\leftrightarrow} x \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \kappa'(c,z) \mapsto b = c \wedge y \underline{\leftrightarrow} \mathsf{push}(z,a) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ENDCASES}\ \mathsf{ENDCASES}\ ] \\
\quad \textbf{CONSTRUCTOR}\ \ \mathsf{empty}\colon X \\
\quad \textbf{CREATION}\ \ \mathsf{pop\_empty}\colon [\ \mathsf{pop}(\mathsf{empty}) = \kappa *\ ] \\
\textbf{END}\ \mathsf{Queue}
\end{array}
$$

We wish to prove the safety and progress properties:

$$\Box(\lambda y : \mathsf{Self} : \mathsf{size}(y) \geq 0 \wedge (\mathsf{size}(y) = 0 \Rightarrow \mathsf{pop}(y) = \kappa *))(\mathsf{empty})$$

And:

$$\Box(\lambda y : \mathsf{Self}.\ \mathsf{size}(y) \geq 0 \wedge \Diamond(\lambda z : \mathsf{Self}.\ \mathsf{size}(z) = 0)(y))(\mathsf{empty})$$

The proofs use that the following is a Queue invariant:

$$\begin{array}{l} \mathsf{size}(x) \geq 0 \\ \wedge \\ \forall n < \mathsf{size}(x).\ \mathsf{iterate}(\mathsf{pop}, n)(x) \neq \kappa * \\ \wedge \\ \mathsf{iterate}(\mathsf{pop}, \mathsf{size}(x))(x) = \kappa * . \end{array}$$

Properties of $\Box$ and $\Diamond$

$\Box\colon \mathcal{P}(X) \to \mathcal{P}(X)$ **is an interior operator**

- $\Box P \subseteq P$
- $\Box P \subseteq \Box\Box P$
- $P \subseteq Q \implies \Box P \subseteq \Box Q$.

$\Diamond\colon \mathcal{P}(X) \to \mathcal{P}(X)$ **is a closure operator**

- $P \subseteq \Diamond P$
- $\Diamond\Diamond P \subseteq \Diamond P$
- $P \subseteq Q \implies \Diamond P \subseteq \Diamond Q$.

Earlier we saw for a polynomial functor T that the predicate lifting operation:

$$(P \subseteq X) \longmapsto (P^T = \mathrm{Pred}(T)(P) \subseteq T(X))$$

preserves all intersections. Equivalently, this operation has a left adjoint

$$(Q \subseteq T(X)) \longmapsto (Q_T \subseteq X)$$

which can also be defined by induction on T. We thus have a Galois connection:

$$Q_T \subseteq P \quad \Longleftrightarrow \quad Q \subseteq P^T$$

For a coalgebra $c : X \longrightarrow T(X)$ and a predicate $P \subseteq X$, define one-step NextTime and LastTime operators:

$$\text{Nexttime}\, P = \{x \in X \mid P^T(c(x))\}$$

$$\text{Lasttime}\, P = (\{c(x) \mid x \in P\})_T$$

The Galois connection $(-)_T \dashv (-)^T$ yields another Galois connection between NextTime and LastTime:

$$\mathsf{Lasttime}\, P \subseteq Q \iff P \subseteq \mathsf{Nexttime}\, Q$$

on the poset $\mathcal{P}X$ of predicates on the state space X of a coalgebra $c \colon X \to T(X)$.

Definition [von Karger 98]

A Galois algebra is a complete Boolean algebra with a Galois connection Lasttime $\dashv$ Nexttime.

Proposition

Every coalgebra of a polynomial functor yields a Galois algebra: $c \colon X \to T(X)$ yields $(\mathcal{P}X, \mathsf{Nexttime})$.

In an arbitrary Galois algebra one interprets, for example,

$$
\begin{array}{rcl}
\mathsf{WeakNexttime}\,P & = & \mathsf{Nexttime}\,P \\
\mathsf{StrongLasttime}\,P & = & \mathsf{Lasttime}\,P \\
\mathsf{StrongNexttime}\,P & = & \neg\mathsf{Nexttime}\neg P \\
\mathsf{WeakLasttime}\,P & = & \neg\mathsf{Lasttime}\neg P \\
\mathsf{Henceforth}\,P \;=\; \Box P & = & \nu S.\,(P \wedge \mathsf{nexttime}\,S) \\
\mathsf{StrongEarlier}\,P & = & \mu S.\,(P \vee \mathsf{Lasttime}\,S)
\end{array}
$$

etc.

Von Karger proves that all axioms and rules of Computation Tree Logic (CTL) are valid in Galois algebras, and that all axioms and rules of Linear Temporal Logic (LTL) are valid in Galois algebras satisfying certain linearity conditions.

Examples

- For a Queue coalgebra $c = \langle \mathsf{size}, \mathsf{push}, \mathsf{pop} \rangle \colon X \longrightarrow \mathbb{Z} \times X^A \times (1 + (A \times X))$,

$$
\begin{array}{rcl}
(\mathsf{Nexttime}\, P)(x) & \Longleftrightarrow & \forall a \in A.\ P(\mathsf{push}(x,a)) \wedge \\
& & \forall a \in A, y \in X.\ \mathsf{pop}(x) = \kappa'(a,y) \Rightarrow P(y) \\[1em]
(\mathsf{Lasttime}\, P)(x) & \Longleftrightarrow & \exists a \in A, y \in X.\ \mathsf{push}(y,a) = x \ \wedge\ P(y) \\
& & \vee\ \exists a \in A, y \in X.\ \mathsf{pop}(y) = \kappa'(a,x) \ \wedge\ P(y)
\end{array}
$$

- For a deterministic automaton $c = \langle \delta, \downarrow \rangle \colon X \to X^A \times \{0,1\}$,

$$
\begin{array}{rcl}
(\mathsf{Nexttime}\, P)(x) & \Longleftrightarrow & \forall a \in A, y \in X.\ x \xrightarrow{a} y \Rightarrow P(y). \\
(\mathsf{Lasttime}\, P)(x) & \Longleftrightarrow & \exists a \in A, y \in X.\ y \xrightarrow{a} x \ \wedge\ P(y).
\end{array}
$$

  where $x \xrightarrow{a} y$ is used for $\delta(x)(a) = y$.

Thus we can apply the theory of Galois algebras for every class/coalgebra.

Adding $\Box$ and $\Diamond$ greatly increase the expressive power of the logic. But more could be asked.

The least μ and greatest ν fixed point operators on monotone functions on predicates further increase the expressiveness. With these operators from the μ-calculus we can express what it means for a binary tree to have finite or infinite behaviour:

$$\begin{array}{rcl} \mathsf{finite} & = & \mu P.\,(\{x \mid \begin{array}[t]{l} \mathsf{CASES}\ \mathsf{children}(x)\ \mathsf{OF} \\ \quad \kappa y \mapsto \mathsf{true} \\ \quad \kappa'(x_1, x_2) \mapsto P(x_1)\ \wedge\ P(x_2) \\ \mathsf{ENDCASES}\,\}\,) \end{array} \end{array}$$

$$\begin{array}{rcl} \mathsf{infinite} & = & \nu P.\,(\{x \mid \begin{array}[t]{l} \mathsf{CASES}\ \mathsf{children}(x)\ \mathsf{OF} \\ \quad \kappa y \mapsto \mathsf{false} \\ \quad \kappa'(x_1, x_2) \mapsto P(x_1)\ \vee\ P(x_2) \\ \mathsf{ENDCASES}\,\}\,) \end{array} \end{array}$$

Rôle of refinements

- A refinement is a relative model of one "abstract" specification assuming a model of a "concrete" model, using the same state space.
- Typically, the concrete specification has more implementation details and less underspecification than the abstract one.
- Refinements are important in the step-wise construction of an implementation:

$$\boxed{\text{Requirement Spec}} \dashrightarrow \boxed{\text{Spec}_1} \xrightarrow{\text{refinements}} \boxed{\text{Spec}_n} \dashrightarrow \boxed{\text{Implementation}}$$

Consider abstract and concrete coalgebraic specifications $\mathcal{A}$ and $\mathcal{C}$ with

- interface functors $\mathcal{IA}$ and $\mathcal{IC}$.
- assertion predicates $\mathcal{A}$-Assert and $\mathcal{C}$-Assert.
- initial states $\mathsf{new}_{\mathcal{A}}$ and $\mathsf{new}_{\mathcal{C}}$, satisfying creation assertions $\mathcal{A}$-Create and $\mathcal{C}$-Create.

Definition (simple)

$\mathcal{C}$ is a refinement of $\mathcal{A}$ if for each "concrete" coalgebra $c\colon X \longrightarrow \mathcal{IC}(X)$ and state **new**$\colon X$ there are an "abstract" coalgebra $c'\colon X \longrightarrow \mathcal{IA}(X)$ and state **new**$'\colon X$ with:

$$\forall x \in X.\, \mathcal{C}\text{-Assert}(c)(x) \;\wedge\; \mathcal{C}\text{-Create}(c)(\mathbf{new})$$
$$\Longrightarrow$$
$$\forall x \in X.\, \mathcal{A}\text{-Assert}(c')(x) \;\wedge\; \mathcal{A}\text{-Create}(c')(\mathbf{new'})$$

In practice the simple version of refinement is not powerful enough.

Definition (advanced)

$\mathcal{C}$ is a refinement of $\mathcal{A}$ if for each "concrete" coalgebra $c\colon X \longrightarrow \mathcal{IC}(X)$ and state **new**: X there are an "abstract" coalgebra $c'\colon X \longrightarrow \mathcal{IA}(X)$ and state **new**$'$: X together with an $\mathcal{A}$-invariant $P \subseteq X$ for c' such that

$$\forall x \in X.\, \mathcal{C}\text{-Assert}(c)(x) \;\wedge\; \mathcal{C}\text{-Create}(c)(\mathbf{new})$$
$$\Longrightarrow$$
$$\forall x \in X.\, P(x) \Rightarrow \mathcal{A}\text{-Assert}(c')(x) \;\wedge\; P(\mathbf{new}') \;\wedge\; \mathcal{A}\text{-Create}(c')(\mathbf{new'})$$

i.e.

$$\forall x \in X.\, \mathcal{C}\text{-Assert}(c)(x) \;\wedge\; \mathcal{C}\text{-Create}(c)(\mathbf{new})$$
$$\Longrightarrow$$
$$\Box(\mathcal{A}\text{-Assert}(c'))(\mathbf{new}') \;\wedge\; \mathcal{A}\text{-Create}(c')(\mathbf{new'})$$

Then: c' restricts to a subcoalgebra $P \longrightarrow \mathcal{IA}(P)$ forming an abstract model.

Recall Queue example

$$
\begin{array}{l}
\textbf{BEGIN}\ \mathsf{Queue}[A\colon \mathsf{TYPE}] \\
\quad \textbf{METHOD}\ \ \mathsf{size}\colon X \longrightarrow \mathbb{Z} \\
\qquad\qquad\qquad\ \mathsf{push}\colon X \times A \longrightarrow X \\
\qquad\qquad\qquad\ \mathsf{pop}\colon X \longrightarrow 1 + (A \times X) \\
\quad \textbf{ASSERTION} \\
\qquad \mathsf{size\_push}\colon [\ \forall x \in X.\, \forall a \in A.\, \mathsf{size}(\mathsf{push}(x,a)) = \mathsf{size}(x) + 1\ ] \\
\qquad \mathsf{size\_pop}\colon [\ \forall x \in X.\ \ \mathsf{CASES}\ \mathsf{pop}(x)\ \mathsf{OF} \\
\qquad\qquad\qquad\qquad\qquad\qquad \kappa * \mapsto \mathsf{size} = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \kappa'(a,y) \mapsto \mathsf{size}(y) = \mathsf{size}(x) - 1 \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{ENDCASES}\ ] \\
\qquad \mathsf{pop\_push}\colon [\ \forall x \in X.\, \forall a \in A.\ \ \mathsf{CASES}\ \mathsf{pop}(\mathsf{push}(x,a))\ \mathsf{OF} \\
\qquad\qquad\qquad\qquad\quad \kappa * \mapsto \mathsf{false} \\
\qquad\qquad\qquad\qquad\quad \kappa'(b,y) \mapsto\ \ \mathsf{CASES}\ \mathsf{pop}(x)\ \mathsf{OF} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \kappa * \mapsto b = a \wedge y \underline{\leftrightarrow} x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \kappa'(c,z) \mapsto b = c \wedge y \underline{\leftrightarrow} \mathsf{push}(z,a) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{ENDCASES}\ \mathsf{ENDCASES}\ ] \\
\quad \textbf{CONSTRUCTOR}\ \ \mathsf{empty}\colon X \\
\quad \textbf{CREATION}\ \ \mathsf{pop\_empty}\colon [\ \mathsf{pop}(\mathsf{empty}) = \kappa *\ \ ] \\
\textbf{END}\ \mathsf{Queue}
\end{array}
$$

$$
\begin{array}{l}
\textbf{BEGIN}\ \textsf{Stack}[A\colon \textsf{TYPE}] \\
\quad \textbf{METHOD}\ \ \textsf{size}\colon X \longrightarrow \mathbb{Z} \\
\qquad\qquad\qquad \textsf{push}\colon X \times A \longrightarrow X \\
\qquad\qquad\qquad \textsf{top}\colon X \longrightarrow 1 + (A \times X) \\
\quad \textbf{ASSERTION} \\
\qquad \textsf{size\_push}\colon [\ \forall x \in X.\, \forall a \in A.\, \textsf{size}(\textsf{push}(x,a)) = \textsf{size}(x) + 1\ ] \\
\qquad \textsf{size\_top}\colon [\ \forall x \in X.\ \ \textsf{CASES top}(x)\ \textsf{OF} \\
\qquad\qquad\qquad\qquad\qquad \kappa * \mapsto \textsf{size} = 0 \\
\qquad\qquad\qquad\qquad\qquad \kappa'(a,y) \mapsto \textsf{size}(y) = \textsf{size}(x) - 1 \\
\qquad\qquad\qquad\qquad \textsf{ENDCASES}\ ] \\
\qquad \textsf{top\_push}\colon [\ \forall x \in X.\, \forall a \in A.\ \ \textsf{CASES top}(\textsf{push}(x,a))\ \textsf{OF} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \kappa * \mapsto \textsf{false} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \kappa'(b,y) \mapsto b = a \wedge y \underline{\leftrightarrow} x \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsf{ENDCASES}\ ] \\
\quad \textbf{CONSTRUCTOR}\ \ \textsf{empty}\colon X \\
\quad \textbf{CREATION}\ \ \textsf{top\_empty}\colon [\ \textsf{top}(\textsf{empty}) = \kappa *\ \ ] \\
\textbf{END}\ \textsf{Stack}
\end{array}
$$

Stack refines Queue 1

Goal

Refinement of Queue by Stack (via invariant).

Therefore, assume a Stack coalgebra

$$c = \langle \mathsf{size}, \mathsf{push}, \mathsf{top} \rangle \colon X \longrightarrow \mathbb{Z} \times X^A \times (1 + (A \times X))$$

satisfying the assertions, together with an initial state **new**: X.

Auxilary function remove_last: $X \times \mathbb{N} \longrightarrow 1 + X$

$$
\begin{array}{rl}
\mathsf{remove\_last}(x, n) = & \textbf{IF}\ n = 0 \\
& \textbf{THEN}\ \ \mathsf{CASES}\ \mathsf{top}(x)\ \mathsf{OF} \\
& \qquad\qquad \kappa * \mapsto \kappa * \\
& \qquad\qquad \kappa'(a, y) \mapsto \kappa' y \\
& \qquad\quad \mathsf{ENDCASES} \\
& \textbf{ELSE}\ \ \mathsf{CASES}\ \mathsf{top}(x)\ \mathsf{OF} \\
& \qquad\quad \kappa * \mapsto \kappa * \\
& \qquad\quad \kappa'(a, y) \mapsto\ \mathsf{CASES}\ \mathsf{remove\_last}(x, n-1)\ \mathsf{OF} \\
& \qquad\qquad\qquad\qquad\quad \kappa * \mapsto \kappa * \\
& \qquad\qquad\qquad\qquad\quad \kappa' z \mapsto \kappa'(\mathsf{push}(z, a)) \\
& \qquad\qquad\qquad\qquad\ \ \mathsf{ENDCASES} \\
& \qquad\ \ \mathsf{ENDCASES}
\end{array}
$$

Refinement mapping

$$\textbf{new'} \stackrel{\text{def}}{=} \textbf{new}$$

$$
\begin{array}{rcl}
c'(x) & \stackrel{\text{def}}{=} & (\\
& & \quad \mathsf{size}(x), \\
& & \quad \lambda a \in A.\, \mathsf{push}(x, a), \\
& & \quad \textbf{IF } \mathsf{size} \leq 0 \\
& & \quad \textbf{THEN } \kappa * \\
& & \quad \textbf{ELSE } \ \mathsf{CASES\ iterate}(\mathsf{top}, \mathsf{size}(x) - 1)(x) \ \mathsf{OF} \\
& & \qquad\qquad \kappa * \mapsto \kappa * \\
& & \qquad\qquad \kappa'(a, y) \mapsto \ \mathsf{CASES\ remove\_last}(x, \mathsf{size}(x) - 1) \ \mathsf{OF} \\
& & \qquad\qquad\qquad\qquad \kappa * \mapsto \kappa * \\
& & \qquad\qquad\qquad\qquad \kappa' z \mapsto \kappa'(a, z) \\
& & \qquad\qquad\qquad \mathsf{ENDCASES} \\
& & \qquad\quad \mathsf{ENDCASES} \\
& &)
\end{array}
$$

Invariant $P \subseteq X$

$$
\begin{array}{rcl}
P(x) & \stackrel{\mathrm{def}}{=} & \mathsf{size}(x) \geq 0 \\
& & \wedge \\
& & \forall n < \mathsf{size}(x).\ \mathsf{iterate}(\mathsf{top}, n)(x) \neq \kappa * \\
& & \wedge \\
& & \mathsf{iterate}(\mathsf{top}, \mathsf{size}(x))(x) = \kappa * \\
& & \wedge \\
& & \forall n < \mathsf{size}(x).\ \mathsf{remove\_last}(x, n) \neq \kappa * \\
& & \wedge \\
& & \mathsf{remove\_last}(x, \mathsf{size}(x)) = \kappa *
\end{array}
$$

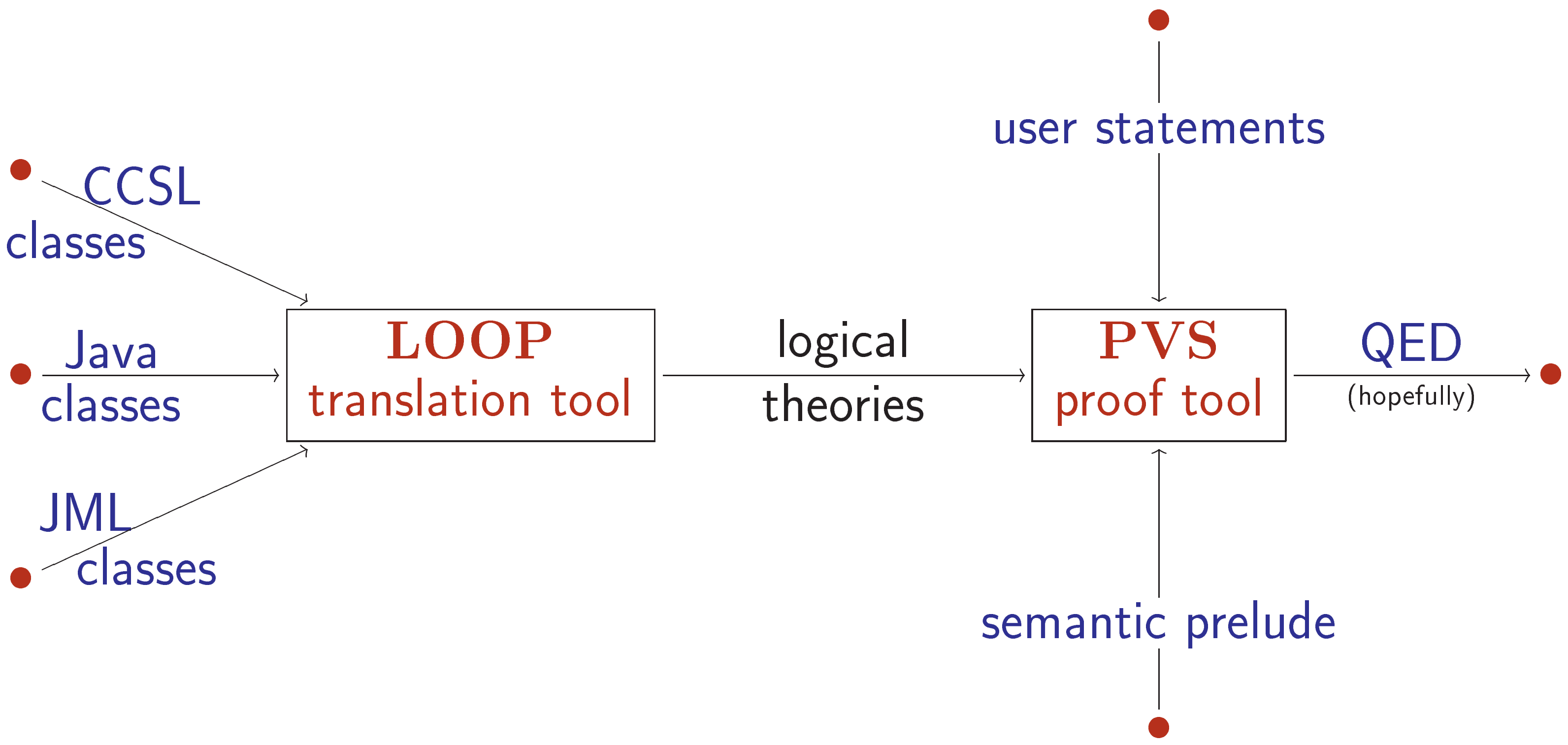
Refinement proof

1. Since top(**new**) $= \kappa *$ we get size(**new**) $= 0$, so that it easy to see that $P($**new**$)$ and pop(**new**) $= \kappa *$.

2. The proof that $P(x)$ implies the abstract assertions is more difficult, and requires several auxiliary lemmas. E.g. for 'size_pop' we need

$$\mathsf{remove\_last}(x, n) = \kappa' y \implies \mathsf{size}(y) + 1 = \mathsf{size}(x)$$

(This whole proof has been done formally using the proof tool PVS.)



The LOOP tool is implemented in Ocaml and has an Emacs interface.

It is possible to use also Isabelle as a back-end theorem prover.

CCSL, for Coalgebraic Class Specification Language is an experimental specifcation language for classes in object-oriented languages.

It involves parameterisation (by types and values), inheritance, aggregation.

The LOOP compiler automatically generates tailor-made definitions of invariant, bisimulation, $\Box$ (henceforth) and $\Diamond$ (eventually) for the CCSL classes.

The LOOP tool provides a semantics for CCSL classes in terms of coalgebras in the logic of the back-end proof tool. (Similarly for Java and JML).

The LOOP tool is developed jointly at the universities of Nijmegen and Dresden, by:

Joachim van den Berg, Marieke Huisman, Bart Jacobs, Erik Poll (Nijmegen)

Ulrich Hensel, Hendrik Tews, Jan Rothe (Dresden).

Further topics and conclusions

- Coalgebras can form the basis for a specification formalism, with a assertion logic with bisimulations, invariants and temporal operators.

- Crucial in the definition of this logic are predicate and relation lifting.

- A formal language CCSL, for Coalgebraic Class Specification Language exists, together with an associated compiler. It translates classes in CCSL into logical theories of the proof tool PVS. This allows tool-assisted reasoning about class specifications, involving models, theories, and refinements.

- Also Java classes can be modeled as coalgebras with assertions.

See also: `http://www.cs.kun.nl/~bart/LOOP`