IFIP Working Group 1.3 Foundations of System Specification Stanford June 29–July 1, 2000

Coalgebras in PVS

Sam Owre

owre@csl.sri.com

Computer Science Laboratory

SRI International

Menlo Park, CA

29^{th} June - 1^{st} July, 2000

Joint work with Falk Bartels, Harald Ruess, N. Shankar

Coalgebras in PVS

1. Introduction to PVS

2. Datatypes in PVS

3. Codatatypes in PVS

4. Related work

5. Future work

PVS

- Based on higher-order logic
- Rich type system with dependent function, tuple, and record types, predicate subtypes, and datatypes
- Modular structure via parameterized theories, providing a limited form of polymorphism
- More information at `http://pvs.csl.sri.com`

Datatypes in PVS

```
list [T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr: list): cons?
 END list
```

This corresponds to the initial algebra of the functor

$$1 + \mathtt{T} \times F \rightarrow F$$

Constructors: `null`, `cons`

Recognizers: `null?`, `cons?`

Accessors: `car`, `cdr`

This automatically generates a theory that defines the datatype

Generated `list` Theory

```
list_adt[T: TYPE]: THEORY
 BEGIN

  list: TYPE

  null?, cons?: [list -> boolean]

  null: (null?)

  cons: [[T, list] -> (cons?)]

  car: [(cons?) -> T]

  cdr: [(cons?) -> list]

  ...
 END list_adt
```

Generated Datatype Declarations

- Extensionality:

```
list_cons_extensionality: AXIOM
  FORALL (cons?_var: (cons?), cons?_var2: (cons?)):
    car(cons?_var) = car(cons?_var2)
         AND cdr(cons?_var) = cdr(cons?_var2)
     IMPLIES cons?_var = cons?_var2;
```

- Induction schema:

```
list_induction: AXIOM
  FORALL (p: [list -> boolean]):
    (p(null) AND
      (FORALL (cons1_var: T, cons2_var: list):
         p(cons2_var) IMPLIES p(cons(cons1_var, cons2_var))))
     IMPLIES (FORALL (list_var: list): p(list_var));
```

Generated Datatype Declarations

- `every` is defined when there are positive type parameters—`some` is defined dually:

```
every(p: PRED[T])(a: list):  boolean =
    CASES a
      OF null: TRUE,
         cons(cons1_var, cons2_var):
       p(cons1_var) AND every(p)(cons2_var)
      ENDCASES;
```

- Subterm relation:

```
<<(x, y: list):  boolean =
    CASES y
      OF null: FALSE,
         cons(cons1_var, cons2_var):
               x = cons2_var OR x << cons2_var
      ENDCASES;

 list_well_founded: AXIOM well_founded?[list](<<);
```

map theory

Generated when there are positive type parameters

```
list_adt_map[T: TYPE, T1: TYPE]: THEORY
 BEGIN
  IMPORTING list_adt
  map(f: [T -> T1])(a: list[T]):  list[T1] =
      CASES a
        OF null: null,
           cons(cons1_var, cons2_var):
             cons(f(cons1_var), map(f)(cons2_var))
        ENDCASES;
  ...
  every(R: [[T, T1] -> boolean])(x: list[T], y: list[T1]):  boolean =
      null?(x) AND null?(y) OR
       cons?(x) AND
        cons?(y) AND R(car(x), car(y)) AND every(R)(cdr(x), cdr(y));
 END list_adt_map
```

reduce theory

```
list_adt_reduce[T: TYPE, range: TYPE]: THEORY
 BEGIN
  IMPORTING list_adt[T]
  reduce(null?_fun: range, cons?_fun: [[T, range] -> range]):
  [list -> range] =
      LAMBDA (list_adtvar: list):
        CASES list_adtvar
          OF null: null?_fun,
             cons(cons1_var, cons2_var):
               cons?_fun(cons1_var,
                         reduce(null?_fun, cons?_fun)(cons2_var))
          ENDCASES;
  ...
 END list_adt_reduce
```

Datatype Restrictions

- Recursive references to the datatype must be positive - may not occur in the domain of a function type
- This yields polynomial functors
- Recursive references may also occur as positive type parameters of other datatypes:

```
d: DATATYPE
 BEGIN
  c(a: list[d], ...): c?
  ...
 END d
```

- No other recursive reference is currently allowed

Datatype Restrictions

- These restrictions are overly restrictive, for example, finite sets are disallowed

- Would be nice to allow any accessor type and generate TCCs that guarantee existence of a solution to the functor equation

- This probably needs a more expressiveness language - true polymorphism

Datatype Subtyping

- Would like to have, e.g., `list[nat]` $\subset$ `list[int]`
- Only possible across *positive* type parameters - must not occur in the domain of a function type
- `list[nat]` $\equiv$ `{l: list[int] | every(LAMBDA (x:int): x >= 0)(l)}`

Subtype difficulties

PVS disallows quantifying over the datatype in accessor types

```
trapped: DATATYPE
 BEGIN
  unit: unit?
  succ(a:{t: trapped | FORALL (x, y: trapped): x = y}): succ?
 END trapped
```

This is actually more restrictive than necessary - with another constructor the problem disappears

More on Datatypes

There is more to datatypes than what is presented here

- Dependent types
- Mutually recursive datatypes
- Proof support

This is described in a technical report available as a system manual from `http://pvs.csl.sri.com`

Codatatypes

Infinite sequences:

```
seq[T: TYPE]: CODATATYPE
 BEGIN
  scons(hd: T, tl: seq): scons?
 END seq
```

This corresponds to the final coalgebra of the functor

$$F \rightarrow \mathtt{T} \times F$$

As with datatypes, new theories are automatically generated

Generated Codatatype Declarations

Bisimulation relation:

```
bisimulation?(R: PRED[[seq, seq]]):  boolean =
    FORALL (x: seq, y: seq):
      R(x, y) IMPLIES
       scons?(x) AND scons?(y) AND hd(x) = hd(y) AND R(tl(x), tl(y));
```

Coinduction:

```
coinduction: FORMULA
  FORALL (B: (bisimulation?), x: seq, y: seq): B(x, y) => x = y;
```

some

Some is defined inductively:

```
some(p: PRED[T])(a: seq): INDUCTIVE boolean =
CASES a
  OF scons(scons1_var, scons2_var):
       p(scons1_var) OR some(p)(scons2_var)
  ENDCASES;
```

The following induction scheme is produced automatically:

```
some_induction: AXIOM
  FORALL (p: PRED[T], P: [seq -> boolean]):
    (FORALL (a: seq):
       CASES a
         OF scons(scons1_var, scons2_var):
              p(scons1_var) OR some(p)(scons2_var) AND P(scons2_var)
         ENDCASES
        IMPLIES P(a))
     IMPLIES (FORALL (a: seq): some(p)(a) IMPLIES P(a));
```

Coinduction

- To define `every`, we need a coinductive definition
- Almost trivial to implement from the code for inductive definitions
- The (non)-well-founded part of a relation:

```
T   : TYPE
<   : VAR pred[[T,T]]
u,v : VAR T

wf_part(<)(u): INDUCTIVE boolean =
  FORALL v: v < u IMPLIES wf_part(<)(v)
```

- Conversely:

```
nwf_part(<)(u): COINDUCTIVE boolean =
  EXISTS v: v < u AND nfw_part(<)(v)
```

Generated Codatatype Declarations

Every:

```
every(p: PRED[T])(a: seq): COINDUCTIVE boolean =
CASES a
  OF scons(scons1_var, scons2_var):
       p(scons1_var) AND every(p)(scons2_var)
  ENDCASES;
```

Coinduction schema automatically generated from above:

```
every_coinduction: AXIOM
  FORALL (p: PRED[T], P: [seq -> boolean]):
    (FORALL (a: seq):
       P(a) IMPLIES
        CASES a
          OF scons(scons1_var, scons2_var):
               p(scons1_var) AND
                 (every(p)(scons2_var) OR P(scons2_var))
          ENDCASES)
     IMPLIES (FORALL (a: seq): P(a) IMPLIES every(p)(a));
```

map

- Map is pretty much the same as for datatypes
- An overloaded relational every is also defined:

```
every(R: [[T, T1] -> boolean])(x: seq[T], y: seq[T1]):  boolean =
    scons?(x) AND
     scons?(y) AND R(hd(x), hd(y)) AND every(R)(tl(x), tl(y));
```

- This is needed to define the bisimulation relation when using codatatypes defined in terms of (co)datatypes:

```
cod: CODATATYPE
 BEGIN
  c(a: seq[cod], ...): c?
  ...
 END cod
```

Coreduce

- In `reduce`, a function is provided for each constructor that maps to the range
- Coreduce maps a given domain type to a range of the right "shape"
- This is accomplished in PVS using a datatype:

```
seq_codt_coreduce[T: TYPE, domain: TYPE]: THEORY
 BEGIN
  IMPORTING seq_codt[T]

   seq_struct: DATATYPE
   BEGIN
    inj_scons(inj_hd: T, inj_tl: domain): inj_scons?
   END seq_struct
  ...
```

Coreduce

```
coreduce(op: [domain -> seq_struct])(x: domain):
      {c: seq[T] | inj_scons?(op(x)) AND scons?(c)} =
    CASES op(x)
      OF inj_scons(inj_scons1_var, inj_scons2_var):
           scons(inj_scons1_var, coreduce(op)(inj_scons2_var))
      ENDCASES;
```

More Examples

```
% Finite or infinite sequences
sequ[T: TYPE]: CODATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr: sequ): cons?
 END sequ


% Finite or infinite depth binary tree
bintree[T: TYPE]: CODATATYPE
 BEGIN
  leaf(val: T): leaf?
  node(val: T, left, right: bintree): node?
 END bintree
```

More Examples

```
% Finite or infinite depth, infinite width trees
inftree[T: TYPE]: CODATATYPE
 BEGIN
  IMPORTING seq
  leaf(val: T): leaf?
  node(val: T, children: seq[inftree]): node?
 END inftree

Automaton[A: TYPE]: CODATATYPE
 BEGIN
  proc( acc: boolean, trans: [A -> Automaton]):proc?
 END Automaton

p_adic_number[p: posnat] : CODATATYPE
 BEGIN
  p_adic(hd: below[p], tl: p_adic_number): p_adic?
 END p_adic_number
```

Related Work

- Tom Melham provided recursive type definitions in HOL
- Larry Paulson provided (co)induction and (co)recursion in Isabelle
- C. Paulin-Mohring et al, provided forms of coalgebras in Coq
- Paul Miner's PhD Thesis *Hardware Verification using Coinductive Assertions* provides a definition of streams in PVS
- Bart Jacobs and Ulrich Hensel provided a coalgebraic theory of sequences in PVS

Related Work

- Bart Jacobs, et al, the LOOP (Logic of Object-Oriented Programming) project. Developed the Coalgebraic Class Specification Language (CCSL) that translates classes in object-oriented languages such as Java into logical theories for use in proof tools such as PVS or Isabelle

Future Work

- Exact formulation of restriction on datatype accessor quantifiers
- Formulate existence of functor solution as a TCC
- Add full polymorphism
- There is an asymmetry in this approach; reduce can be defined directly, but coreduce needs a datatype
- Define specialized proof strategies for automating coinductive proofs
- Development of corecursion schemes, possibly based on a suitable notion of productivity
- With minor changes to the generated code, can generate (co)algebras that are neither initial or final

Conclusion

- Coalgebras have been implemented in PVS
- The expressiveness of the language made aspects of this easy; much of the existing datatype code was reusable
- Send email to `owre@csl.sri.com` if you wish to experiment with codatatypes in PVS