Coalgebraic Theories of Sequences in PVS*

Ulrich Hensel
Inst. Theor. Inf., TU Dresden
D-01062 Dresden,
Germany.
hensel@tcs.inf.tu-dresden.de

Bart Jacobs
Dep. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen,
The Netherlands.
bart@cs.kun.nl

July 17, 1998

Abstract

This paper explains the setting of an extensive formalisation of the theory of sequences (finite and infinite lists of elements of some data type) in the Prototype Verification System PVS. This formalisation is based on the characterisation of sequences as a final coalgebra, which is used as an axiom. The resulting theories comprise standard operations on sequences like composition (or concatenation), filtering, flattening, and their properties. They also involve the prefix ordering and proofs that sequences form an algebraic complete partial order. The finality axiom gives rise to various reasoning principles, like bisimulation, simulation, invariance, and induction for admissible predicates. Most of the proofs of equality statements are based on bisimulations, and most of the proofs of prefix order statements use simulations. Some significant aspects of these theories are described in detail.

This coalgebraic formalisation of sequences is presented as a concrete example that shows the importance and usefulness of coalgebraic modeling and reasoning. Hopefully, it will help to convey the view that coalgebraic data types should form an intrinsic part of (future) languages for programming and reasoning. Therefore, some suggestions for an appropriate syntax for coalgebraic datatypes are included.

The use of sequences as a final coalgebra is demonstrated in two (standard) applications: a refinement result for automata involving sequences of actions, and a coalgebraic definition plus correctness proof for an insert operation on ordered sequences.

1 Introduction

Formal verification always involves a certain amount of theory development. Theories are needed to adequately describe one's application domain (with appropriate operations). If there is no useful library of theories at hand, theory development may form a significant part of a verification project. It is therefore important that standard theories are available for frequently occurring structures. Finite lists, for example, have a well-developed theory, which is so often used that it forms a standard part of (the "prelude" or "basic library" of) almost all proof tools.

This paper contributes to the theory of sequences, *i.e.* to the theory of finite and infinite lists (of elements of a fixed data set). Good theory development should satisfy some quality criteria. We think it should be (1) tool-independent, (2) general purpose, and (3) describing a "standard" theory. The theory of sequences that we present here is developed in the verification system PVS [ORSvH95, ORR+96, RSC96], but its basic notions are firmly founded on standard mathematical theory, which can be expressed in the language of any sufficiently expressive proof tool. Further, regarding criterion (2), a theory of sequences is certainly of general use, since sequences play an important rôle in many verification projects. For example, the behavior of

*This paper was written during a visit of Ulrich Hensel to the Computing Science Institute of the University of Nijmegen.

systems (or of automata or processes) is often expressed in terms of sequences, forming executions, runs, or traces; and lazy lists are important in describing and reasoning about the behavior of (lazy) functional programs. Finally, as to (3), describing sequences as a final coalgebra is certainly standard in category theory, just like finite lists are standardly described as an initial algebra. For those readers who are not familiar with these coalgebraic techniques, the approach that we present is hopefully providing a similar standard.

The formalisation of the theory of sequences and infinite objects in general is an active research area. Streams or infinite lists received attention in the context of dataflow programming languages. Leclerc and Paulin-Mohring [LP94, Pau96] suggest an impredicative encoding using existential types in COQ and derive explicit corecursion combinators. Johnson and Miner [MJ96] develop stream theories axiomatically but do not state finality explicitly. Theories of sequences are directly encoded in a variety of frameworks, see *e.g.* [Age94, CP96, NS95, Reg95, MN97, DGM97, DG97]. Closest to our approach is perhaps Paulsons formalisation of coinductive lazy lists in ISABELLE/HOL [Pau97a]. He establishes an encoding of codatatypes as greatest fixed point of monotone operators on a suitable domain. His corecursion combinator identifies lazy lists as final coalgebras.

The approach presented here isolates final coalgebras as a characterisation of sequences. The abstraction from the particular encoding of sequences (possibly as a final coalgebra) avoids clutter and provides conceptual clarity. Our results are in this respect implementation independent. Finality immediately gives us so-called coinductive definition principles and coinductive proof principles for sequences. We explain how these principles are used to obtain a wide variety of operations on sequences and how to prove certain properties about them. We think that our formalisation of sequences is one of the most extensive applications of coalgebraic (or coinductive) techniques to date.

It is not our intention to explain coalgebras and coinduction in general—therefore we refer to a tutorial [JR97] on these matters. Here we concentrate on the special case of sequences, which we will explain in detail (without using category theory). But we do hope that this particular example will illustrate the importance and usefulness of coalgebraic data types more generally, and that it will have an influence on future languages for programming and reasoning. When these languages are equipped with definition mechanisms both for algebraic and for coalgebraic data types (as suggested in [Hag87] and realised in the experimental programming language CHARITY [CF92, CS95]), then one can use (and reason about) coalgebraic data structures like trees with infinitely many branches, and possibly infinite depth with the same ease as algebraic structures like finitely branching trees. What is useful to have is a language which does not only provide users with coalgebraic modeling techniques, but also with coalgebraic (also called coinductive) reasoning techniques. As a possible step towards such languages with coalgebraic data types we present in Section 6 some suggestions for a syntax and for associated reasoning principles (notably involving bisimulations).

This paper is organised as follows. In Section 2 we describe some aspects of the mathematical theory of sequences by introducing several standard operations and their properties. The axiomatic description of sequences as a final coalgebra and the corresponding coinductive definition and proof principles appear in Section 3. In Section 4 we give a tour through our sequence formalisation in PVS. And in Section 5 we submit our theories to two sample challenges. Section 6 contains a sketch of a general description of coalgebraic datatypes, with associated reasoning principles. We conclude with a brief comparison to other sequence formalisation in Section 7.

2 Some standard operations on sequences (and their properties)

We fix an arbitrary set A and write A^∞ for the set of finite and infinite sequences of elements of A. Thus, every element $\sigma \in A^\infty$ is either of the form $\sigma = \langle a_1, \ldots, a_n \rangle$, for finitely many elements $a_1, \ldots, a_n \in A$, or of the form $\sigma = \langle a_1, a_2, \ldots \rangle$ for a (countably) infinite series of elements

$a_i \in A$. For such sequences σ we will write σ_i for the ith element a_i. As a first step towards our formalisation, we investigate some characteristic operations and results for this set of sequences A^∞.

For example, for a finite or infinite sequence σ there are the "at" and "tail" operations:

$$\mathsf{at}(\sigma, n) = \begin{cases} \sigma_n, \text{ the } n\text{th element, if it exists} \\ \bot, \text{ for undefined, otherwise.} \end{cases} \qquad \text{and} \qquad \mathsf{tail}(\sigma, n) = (\sigma_n, \sigma_{n+1}, \sigma_{n+2}, \ldots)$$

yielding the nth element in σ, if it exists, and yielding the elements of σ starting from position n. The outcome of $\mathsf{tail}(\sigma, n)$ is the empty sequence if σ has less than n elements. These operations at and tail can be defined by induction on n. Some standard properties for at and tail are:

$$\mathsf{at}(\mathsf{tail}(\sigma, n), m) = \mathsf{at}(\sigma, n+m) \qquad \text{and} \qquad \mathsf{tail}(\mathsf{tail}(\sigma, n), m) = \mathsf{tail}(\sigma, n+m)$$

which can be proved easily by induction. Also, one can characterise equality of sequences in terms of at as,

$$\sigma = \tau \qquad \text{if and only if} \qquad \forall n \in \mathbb{N}\ \mathsf{at}(\sigma, n) = \mathsf{at}(\tau, n).$$

There is a similar operation which sends a sequence σ and a natural number n to the finite list of elements in σ up-to position n. Or an operation which takes a sequence σ and two natural numbers n, m and produces the sublist of elements in σ from n to $n+m$.

It is standard that functions can be extended to sequences: for a function $f\colon A \to B$, one can define a function $\mathsf{seq\_map}(f)$ (or f^∞) from A^∞ to B^∞ by sending a sequence (a_i) to the sequence $(f(a_i))$, obtained by applying f elementwise. This function $\mathsf{seq\_map}(f)$ commutes appropriately with the at and tail operations. And $\mathsf{seq\_map}$ is functorial in the sense that it preserves identities and compositions.

We write $A^\star$ for the set of finite sequences (or lists) of elements of A, and $A^{\mathbb{N}}$ for the set of infinite sequences (*i.e.* of functions $\mathbb{N} \to A$). There are obvious (injective) inclusion functions $A^\star \hookrightarrow A^\infty$ and $A^{\mathbb{N}} \hookrightarrow A^\infty$. And one can prove that each sequence $\sigma \in A^\infty$ is either in the image of $A^\star \hookrightarrow A^\infty$ or in the image of $A^{\mathbb{N}} \hookrightarrow A^\infty$. In general, it turns out to be a bad strategy to define operations on sequences by distinguishing whether the input sequence is finite or infinite: it leads to much complication and unnecessary duplication.

An important operation on sequences is composition (also called concatenation) $\mathsf{comp}\colon A^\infty \times A^\infty \to A^\infty$. The sequence $\mathsf{comp}(\sigma, \tau)$ consists of all elements of σ prefixed to τ. It may be described as:

$$\mathsf{comp}(\sigma, \tau) = \begin{cases} \sigma & \text{if } \sigma \text{ is infinite} \\ a_1 \cdot a_2 \cdots a_n \cdot \tau & \text{if } \sigma \text{ is a finite sequence } (a_1, a_2, \ldots, a_n). \end{cases}$$

(but we shall see a better definition in the next section which does not distinguish whether σ is infinite or not). Some useful properties are: the empty sequence is a neutral element (both on the left and on the right), composition is associative, composition of two finite lists (as sequences) is the same as the result of appending the lists, *etc.* Also, the composition operation allows us to define the prefix order on sequences as:

$$\sigma \leq \tau \qquad \text{if and only if} \qquad \exists \rho \in A^\infty\ \mathsf{comp}(\sigma, \rho) = \tau.$$

It is not hard to see that $\leq$ forms a partial order on A^∞, with the empty sequence as bottom element. This order is complete in the sense that each ascending chain $\sigma_1 \leq \sigma_2 \leq \cdots$ of sequences has a least upper bound. Also, it can be shown that each sequence is the least upper bound of its finite prefixes. This entails that $(A^\infty, \leq)$ forms an algebraic complete partial order (see *e.g.* [DP96]).

An important operation on sequences is filtering. The filter function takes a predicate p on A and a sequence $\sigma \in A^\infty$ and produces the sequence $\mathsf{filter}(p, \sigma)$ containing only those elements of σ (in original order) which satisfy p. This filtering is often used in describing the external behavior of systems by filtering out the internal behavior, see Section 5. Standard properties are:

$$\begin{aligned} \mathsf{filter}(p, \mathsf{comp}(\sigma, \tau)) &= \mathsf{comp}(\mathsf{filter}(p, \sigma), \mathsf{filter}(p, \tau)), \quad \text{if } \sigma \text{ is finite} \\ \mathsf{filter}(p, \mathsf{filter}(q, \sigma)) &= \mathsf{filter}(p \wedge q, \sigma). \end{aligned}$$

A consequence of the last result is that filtering is idempotent. Notice that for an infinite sequence σ, the result $\mathsf{filter}(p, \sigma)$ of filtering with predicate p may be either finite or infinite.

Another standard operation is flattening, which we consider (for convenience) on sequences of finite lists only. For a sequence $S \in (A^\star)^\infty$ of finite lists, $\mathsf{flatten}(S)$ is the sequence obtained from S by removing the inner braces. It is a non-trivial result that flattening commutes with filtering, in the sense that:

$$\mathsf{filter}(p, \mathsf{flatten}(S)) = \mathsf{flatten}(\mathsf{seq\_map}(\lambda\sigma\ \mathsf{filter}(p, \sigma))(S)).$$

That is, filtering the flattened S is the same as first filtering the lists (with the appropriate filter function for lists) in S individually, and then flattening the resulting sequence of lists.

This concludes our brief overview of sequences. In the next section we shall discuss many of these aspects in an axiomatic setting. There, we shall also see some important proof principles for sequences.

3 Sequences as a final coalgebra

What is characteristic of *finite* lists (of elements of A) is that they can all be represented by a finite term generated from the empty list "nil" and from a "cons" operation. The "cons" operation prefixes a single element to an existing list. In contrast, it is not possible to construct all sequences in such finite manner. But we do have a characteristic "destruction" operation on sequences—to be called next—which, given an arbitrary sequence $\sigma \in A^\infty$, tells us if the sequence is empty, and, if not, produces the head of σ (which is in A) and the remainder (the tail) of σ (which is in A^∞ again). If there is a tail, this next operation may be applied again, yielding possibly a second element in A together with the tail of the tail. A sequence is thus finite if and only if iterating this next operation stops at some stage.

We describe this destruction operation for sequences in greater detail[1]. Therefore we need the "lift" operation $\mathsf{lift}(\bot)$ which adds a new element $\bot$ to a set. Thus, $\mathsf{lift}(X) = X \cup \{\bot\}$, where $\bot \notin X$ stands for "undefined". Now we can identify partial functions $Y \to X$ with total functions $Y \to \mathsf{lift}(X)$.

In particular, the destruction operation on sequences is a function

$$A^\infty \xrightarrow{\quad\mathsf{next}\quad} \mathsf{lift}(A \times A^\infty)$$

defined by

$$\mathsf{next}(\sigma) = \begin{cases} \bot & \text{if } \sigma \text{ is the empty sequence } \langle\rangle \\ (a, \sigma') & \text{if } \sigma \text{ is non-empty, with head } a \text{ and tail } \sigma', \textit{ i.e.} \text{ if } \sigma = a \cdot \sigma'. \end{cases}$$

This function next is of great importance: virtually everything that we will say about sequences will be said in terms of next. In fact, this is the essence of the coalgebraic description of sequences: the next operation tells us what is directly observable about a sequence—whether it is empty or not, and if not, what its head and tail are; and by iteratively applying next (to tails of sequences) each element in a sequence appears at some stage. This approach does not distinguish whether a sequence is finite or infinite.

We illustrate this rôle of next in some examples, with the natural numbers as data set $A = \mathbb{N}$. Consider the function $\mathsf{up\_from}\colon \mathbb{N} \to \mathbb{N}^\infty$ which sends a natural number $n \in \mathbb{N}$ to the (infinite) sequence $n, n+1, n+2, \ldots$. In terms of next, this sequence $\mathsf{up\_from}(n)$ is determined by:

$$\mathsf{next}(\mathsf{up\_from}(n)) = (n, \mathsf{up\_from}(n+1)). \tag{1}$$

[1] To be categorically precise, we investigate A^∞ as a final coalgebra of the functor $X \mapsto 1 + (A \times X)$ on the category of sets and functions. What we say applies to sequences in the ordinary universe of sets, and not to sequences in the domain theoretic universe of domains and strict functions, as formalised in HOLCF, see [Reg95, MN97].

Similarly, there is a function $\mathsf{down\_from}\colon \mathbb{N} \to \mathbb{N}^\infty$ which maps a number n to the (finite) sequence $n \perp 1, n \perp 2, \ldots, 0$. It is determined by:

$$\mathsf{next}(\mathsf{down\_from}(n)) = \begin{cases} \perp & \text{if } n = 0 \\ (n \perp 1, \mathsf{down\_from}(n \perp 1)) & \text{otherwise.} \end{cases} \tag{2}$$

Bisimulations are special relations on sequences (in the present context), that can be used to prove the equality of two sequences in a step-by-step manner. Formally, a bisimulation is a relation $R \subseteq A^\infty \times A^\infty$ which satisfies the following property. For all sequences $\sigma, \tau \in A^\infty$, if $R(\sigma, \tau)$ then

- either both $\mathsf{next}(\sigma)$ and $\mathsf{next}(\tau)$ are equal to $\perp$;
- or both $\mathsf{next}(\sigma)$ and $\mathsf{next}(\tau)$ are not $\perp$, say $\mathsf{next}(\sigma) = (a, \sigma')$ and $\mathsf{next}(\tau) = (b, \tau')$; in this case it is required that $a = b$ and that σ' and τ' are related by R, *i.e.* that $R(\sigma', \tau')$ holds.

Two sequences $\sigma, \tau \in A^\infty$ are called *bisimilar* if there is a bisimulation $R \subseteq A^\infty \times A^\infty$ with $R(\sigma, \tau)$. In that case one often writes $\sigma \mathrel{\underline{\leftrightarrow}} \tau$.

The bisimulation proof principle states that bisimilar sequences are equal, *i.e.* that

$$\sigma \mathrel{\underline{\leftrightarrow}} \tau \qquad \Rightarrow \qquad \sigma = \tau. \tag{3}$$

This means that in order to show that two sequences σ and τ are equal, it suffices to come up with a bisimulation relation R with $R(\sigma, \tau)$. This is an extremely useful principle: it reduces a "global" task of showing that two potentially infinite structures are equal to a "local" task, namely of showing that a particular relation R is a bisimulation. The latter only requires us to prove something about the next step[2] in a sequence (and not about the whole sequence). But it can be a non-trivial matter to actually find an appropriate relation R which does the job. Usually, there is an obvious candidate, but sometimes the obvious relation has to be strengthened[3] in an appropriate way.

The operation $\mathsf{next}\colon A^\infty \to \mathsf{lift}(A \times A^\infty)$ plays a crucial rôle in the definition of a bisimulation relation. Actually showing that a particular relation R is a bisimulation is easiest if one deals with sequences which are determined in terms of next—like $\mathsf{up\_from}(n)$ and $\mathsf{down\_from}(n)$ described above. The finality principle that we will introduce now allows us to define such functions. This principle holds in the universe of sets and functions. It will be required as an axiom in our formalisation of sequences in PVS, and this single axiom will form the basis for all subsequent theories.

Fact 1 *The operation* $\mathsf{next}\colon A^\infty \to \mathsf{lift}(A \times A^\infty)$ *satisfies the following "finality" property. For each set* X *with an operation* $\mathsf{struct}\colon X \to \mathsf{lift}(A \times X)$ *of the same shape, there is a unique function* $f\colon X \to A^\infty$ *satisfying*

1. *if* $\mathsf{struct}(x) = \perp$, *then* $\mathsf{next}(f(x)) = \perp$;
2. *if* $\mathsf{struct}(x) = (a, x')$, *then* $\mathsf{next}(f(x)) = (a, f(x'))$.

In this situation one says that f *is defined* by coinduction. *In our formalisation we write* $\mathsf{coreduce}(\mathsf{struct})$ *for this function* f *(in analogy with the* reduce *operation in* PVS *for inductive definitions). The two conditions 1. and 2. thus allow us to calculate with* $\mathsf{next} \circ \mathsf{coreduce}(\mathsf{struct})\colon X \to \mathsf{lift}(A \times A^\infty)$.

A few words on terminology: functions of the form $\mathsf{struct}\colon X \to \mathsf{lift}(A \times X)$ are examples of *coalgebras*. To be precise, they are examples of coalgebras of the endofunctor[4] $X \mapsto \mathsf{lift}(A \times X)$, see [JR97] for more details. Such a coalgebra $\mathsf{struct}\colon X \to \mathsf{lift}(A \times X)$ may be understood as a

[2] This aspect of "single step" proof obligations is like in the induction step from n to $n+1$ in induction proofs.

[3] Again, this is as for induction proofs, where one sometimes needs to strengthen the statement that one actually wishes to prove in order to get the induction method going. This strengthening is called *induction loading*.

[4] On the category of sets and (total) functions.

machine with state space X and with transition function struct, telling us for each state $x \in X$ if we can move to a next state from x: if $\mathsf{struct}(x) = \bot$, then there is no next state, and if $\mathsf{struct}(x) = (a, x')$, then x' is the successor state of x, and in moving from x to x' we can observe a. The function next is also a coalgebra. It is in fact a very special coalgebra: the above statement tells us that next is the *final* coalgebra in the sense that for an arbitrary coalgebra struct there is precisely one function f which forms a "homomorphism of coalgebras" from struct to next. Diagrammatically:

$$\begin{array}{ccc} A^\infty & \xrightarrow{\mathsf{next}} & \mathsf{lift}(A \times A^\infty) \\ {\scriptstyle \mathsf{coreduce}(\mathsf{struct})}\,\vdots & & \uparrow\,{\scriptstyle \mathsf{lift}(id \times \mathsf{coreduce}(\mathsf{struct}))} \\ X & \xrightarrow[\mathsf{struct}]{} & \mathsf{lift}(A \times X) \end{array}$$

The two conditions 1. and 2. express that f is such a homomorphism. For each state $x \in X$, the resulting sequence $f(x) \in A^\infty$ can be understood as the observable behavior which is obtained by running the machine $\mathsf{struct}\colon X \to \mathsf{lift}(A \times X)$ with $x \in X$ as start state.

With this machine picture in mind, we can easily convince ourselves of the truth of Fact 1. Suppose we have a set X with such a function $\mathsf{struct}\colon X \to \mathsf{lift}(A \times X)$. Then, for an arbitrary element $x \in X$, we can apply struct and observe the outcome $\mathsf{struct}(x)$. Either it is $\bot$, or it is of the form (a, x'), where a is in A and x' is a new element of X. This can be continued: $\mathsf{struct}(x')$ is either $\bot$, or of the form (a', x'') with $a' \in A$ and $x'' \in X$. And-so-forth. In this way we get our sequence $f(x) \in A^\infty$ associated with x: it records all the successive elements of A which appear by iteratively applying struct to x and to its successors $x', x'', \ldots$. This yields a finite or infinite sequence, depending on whether struct will ever hit $\bot$. The sequence $f(x)$ thus records the observable behavior starting from x. The two requirements 1. and 2. determine f in this manner, guaranteeing uniqueness.

The above unique existence statement is formulated in terms of sets and functions. But the formulation is such that it can easily be ported to a sufficiently expressive (typed) logic, like simply typed higher order logic (as used in the HOL [GM93] or ISABELLE/HOL [Pau94, Pau97a] proof tools) or dependently typed higher order logic (as used in PVS [ORR+96, RSC96]), or even in type theoretic languages (as used in tools like LEGO [LP92] and COQ [BBC+97]). The fact that our formalisation was carried out in PVS is basically inessential.

The finality fact involves two aspects, namely *existence* of such a function $f\colon X \to A^\infty$ and *uniqueness*. Existence will be used as a *definition principle*: it allows us to define functions of the form $X \to A^\infty$ with sequences as outputs, by putting an appropriate structure function $\mathsf{struct}\colon X \to \mathsf{lift}(A \times X)$ on the domain X of the function that we wish to define. This function then appears as "unfolding" of struct. Uniqueness will be used as a *proof principle*: it allows us to show that two functions $f, g\colon X \to A^\infty$ are equal, by showing that they both satisfy requirements 1. and 2. (with respect to the same function struct). Uniqueness is actually equivalent to the bisimulation proof principle, that we described earlier[5].

In the remainder of this section we shall describe some standard operations on sequences (like in the previous section) by using only this finality fact 1. The actual formalisation of sequences (which will be described in the next section) follows the same approach.

First we illustrate how the two functions $\mathsf{up\_from}, \mathsf{down\_from}\colon \mathbb{N} \to \mathbb{N}^\infty$ can be defined by coinduction (in the situation with data set $A = \mathbb{N}$). In both cases we take $X = \mathbb{N}$, because both functions have $\mathbb{N}$ as domain. We then need two functions

$$\mathbb{N} \xrightarrow{\mathsf{up\_from\_struct}} \mathsf{lift}(\mathbb{N} \times \mathbb{N}) \qquad \text{and} \qquad \mathbb{N} \xrightarrow{\mathsf{down\_from\_struct}} \mathsf{lift}(\mathbb{N} \times \mathbb{N})$$

so that we get $\mathsf{up\_from} = \mathsf{coreduce}(\mathsf{up\_from\_struct})$ and $\mathsf{down\_from} = \mathsf{coreduce}(\mathsf{down\_from\_struct})$

[5] See the sketch of the PVS proof in Section 4.

by coinduction. These structure functions are defined as

$$\mathsf{up\_from\_struct}(n) = (n, n+1) \qquad \text{and} \qquad \mathsf{down\_from\_struct}(n) = \begin{cases} \bot & \text{if } n = 0 \\ (n \perp 1, n \perp 1) & \text{otherwise.} \end{cases}$$

It may be clear that unfolding these structure maps gives rise to down_from and up_from.

Next we present a more complicated example. The composition (or concatenation) function $\mathsf{comp}\colon A^\infty \times A^\infty \to A^\infty$ can be defined by coinduction. The idea is that $\mathsf{comp}(\sigma, \tau)$ is the sequence obtained by prefixing σ to τ. For a coinductive definition we need an appropriate structure operation of the form

$$(A^\infty \times A^\infty) \xrightarrow{\quad\mathsf{comp\_struct}\quad} \mathsf{lift}(A \times (A^\infty \times A^\infty))$$

which tells us what we can directly observe and how to proceed. Clearly, if both σ and τ are empty sequences, then $\mathsf{comp}(\sigma, \tau)$ is empty as well; if σ is empty, but τ is not, then $\mathsf{comp}(\sigma, \tau)$ is non-empty, with its first element equal to the first element of τ; and if σ is non-empty, then $\mathsf{comp}(\sigma, \tau)$ is non-empty with first element equal to the first element of σ. This leads to the following definition.

$$\mathsf{comp\_struct}(\sigma, \tau) = \begin{cases} \bot & \text{if } \mathsf{next}(\sigma) = \bot \text{ and } \mathsf{next}(\tau) = \bot \\ (a, (\sigma, \tau')) & \text{if } \mathsf{next}(\sigma) = \bot \text{ and } \mathsf{next}(\tau) = (a, \tau') \\ (a, (\sigma', \tau)) & \text{if } \mathsf{next}(\sigma) = (a, \sigma'). \end{cases}$$

Then $\mathsf{comp}\colon A^\infty \times A^\infty \to A^\infty$ is defined as $\mathsf{coreduce}(\mathsf{comp\_struct})$. Notice that if σ is infinite, then we remain in the third case and the outcome $\mathsf{comp}(\sigma, \tau)$ will behave just like (or, be equal to) σ.

Let $1 = \{\emptyset\}$ be a singleton set. The empty sequence empty_seq, as a function $\mathsf{empty\_seq}\colon 1 \to A^\infty$ can be defined by coinduction as $\mathsf{empty\_seq} = \mathsf{coreduce}(\mathsf{undef})(\emptyset)$ where $\mathsf{undef}\colon 1 \to \mathsf{lift}(A \times 1)$ is the everywhere undefined function $\mathsf{undef}(x) = \bot$. Then, by definition, $\mathsf{next}(\mathsf{empty\_seq}) = \bot$.

Two results that we may wish to prove at this stage are that the empty sequence is a neutral element on the left and on the right for composition, *i.e.* that $\mathsf{comp}(\mathsf{empty\_seq}, \sigma) = \sigma$ and $\mathsf{comp}(\sigma, \mathsf{empty\_seq}) = \sigma$. We shall illustrate two proof-techniques: we prove the first equation by exploiting the uniqueness in the finality fact, and the second one by using a bisimulation relation.

In order to prove $\mathsf{comp}(\mathsf{empty\_seq}, \sigma) = \sigma$ by uniqueness, we consider two functions $A^\infty \to A^\infty$, namely the identity function id, and the function f given by $f(\sigma) = \mathsf{comp}(\mathsf{empty\_seq}, \sigma)$. Our aim is to show $f = id$, by showing that both requirements 1. and 2. hold for $X = A^\infty$ and $\mathsf{struct} = \mathsf{next}$. This is obvious for the identity function id, so that $id = \mathsf{coreduce}(\mathsf{next})$. Checking the requirements for f involves some computation.

- if $\mathsf{next}(\sigma) = \bot$, then $\mathsf{next}(f(\sigma)) = \mathsf{next}(\mathsf{comp}(\mathsf{empty\_seq}, \sigma)) = \bot$, since the composition function comp is defined as $\mathsf{coreduce}(\mathsf{comp\_struct})$ and $\mathsf{comp\_struct}(\mathsf{empty\_seq}, \sigma) = \bot$.
- if $\mathsf{next}(\sigma) = (a, \sigma')$, then

  $$\mathsf{next}(f(\sigma)) = \mathsf{next}(\mathsf{comp}(\mathsf{empty\_seq}, \sigma)) = (a, \mathsf{comp}(\mathsf{empty\_seq}, \sigma')) = (a, f(\sigma'))$$

  since $\mathsf{comp\_struct}(\mathsf{empty\_seq}, \sigma) = (a, (\mathsf{empty\_seq}, \sigma'))$.

The second equation $\mathsf{comp}(\sigma, \mathsf{empty\_seq}) = \sigma$ will be proved by showing that the relation

$$R = \{(\mathsf{comp}(\sigma, \mathsf{empty\_seq}), \sigma) \mid \sigma \in A^\infty\} \subseteq A^\infty \times A^\infty$$

is a bisimulation. Therefore we need to prove that:

- $\mathsf{next}(\mathsf{comp}(\sigma, \mathsf{empty\_seq})) = \bot$ if and only if $\mathsf{next}(\sigma) = \bot$. This follows easily from the definition of comp_struct;
- if $\mathsf{next}(\sigma) = (a, \sigma')$, then $\mathsf{next}(\mathsf{comp}(\sigma, \mathsf{empty\_seq})) = (a, \sigma'')$ with $R(\sigma'', \sigma')$. But we can compute σ'' as $\mathsf{comp}(\sigma', \mathsf{empty\_seq})$, so that $R(\sigma'', \sigma')$ clearly holds.

As one can see these two proof methods involve basically the same steps. This is not surprising, since the bisimulation proof principle is equivalent to the uniqueness requirement in Fact 1. In both cases one says that the statement is proved by *coinduction*. We leave it to the interested reader to prove (by hand) that composition is associative, *e.g.* by using the bisimulation relation[6]

$$R = \{(\mathsf{comp}(\sigma, \mathsf{comp}(\tau, \rho)), \mathsf{comp}(\mathsf{comp}(\sigma, \tau), \rho)) \mid \sigma, \tau, \rho \in A^\infty\}.$$

An interesting consequence of Fact 1 is that the function $\mathsf{next}\colon A^\infty \to \mathsf{lift}(A \times A^\infty)$ is an isomorphism[7]. Its inverse $\mathsf{next\_inv}\colon \mathsf{lift}(A \times A^\infty) \to A^\infty$ can be defined by coinduction. Therefore we have to put a structure map on the domain $\mathsf{lift}(A \times A^\infty)$ of the function $\mathsf{next\_inv}$ that we wish to define. This map thus has to be of the form:

$$\mathsf{lift}(A \times A^\infty) \xrightarrow{\mathsf{next\_inv\_struct}} \mathsf{lift}(A \times \mathsf{lift}(A \times A^\infty))$$

It is defined as:

$$\mathsf{next\_inv\_struct}(x) = \begin{cases} \bot & \text{if } x = \bot \\ (a, \mathsf{next}(\sigma)) & \text{if } x = (a, \sigma). \end{cases}$$

Now we obtain $\mathsf{next\_inv}$ as $\mathsf{coreduce}(\mathsf{next\_inv\_struct})$.

We can prove the first equation $\mathsf{next\_inv} \circ \mathsf{next} = id\colon A^\infty \to A^\infty$ by coinduction: the identity on A^∞ is $\mathsf{coreduce}(\mathsf{next})$, as we have just seen, so it suffices to show that the composite $\mathsf{next\_inv} \circ \mathsf{next}$ also satisfies the two requirements in Fact 1 for the structure map next. The second equation $\mathsf{next} \circ \mathsf{next\_inv} = id\colon \mathsf{lift}(A \times A^\infty) \to \mathsf{lift}(A \times A^\infty)$ is proved by calculating next applied to $\mathsf{coreduce}$. For $x \in \mathsf{lift}(A \times A^\infty)$ we can distinguish two cases:

- $x = \bot$. Then

$$\begin{array}{lcl} (\mathsf{next} \circ \mathsf{next\_inv})(x) & = & \mathsf{next}(\mathsf{coreduce}(\mathsf{next\_inv\_struct})(\bot)) \\ & = & \bot, \qquad \text{since } \mathsf{next\_inv\_struct}(\bot) = \bot \\ & = & x. \end{array}$$

- $x = (a, \sigma)$. Then

$$\begin{array}{lcl} (\mathsf{next} \circ \mathsf{next\_inv})(x) & = & \mathsf{next}(\mathsf{coreduce}(\mathsf{next\_inv\_struct})(a, \sigma)) \\ & = & (a, \mathsf{next\_inv}(\mathsf{next}(\sigma))), \\ & & \qquad \text{since } \mathsf{next\_inv\_struct}(a, \sigma) = (a, \mathsf{next}(a, \sigma)) \\ & = & (a, \sigma), \\ & & \qquad \text{using the first equation} \\ & = & x. \end{array}$$

Once we know that the destructor next is an isomorphism, we can define the empty sequences simply as $\mathsf{empty\_seq} = \mathsf{next\_inv}(\bot)$, and the prefix operation as $\mathsf{cons\_seq}(a, \sigma) = \mathsf{next\_inv}(a, \sigma)$.

There is also a *simulation* proof principle for sequences. Just like bisimulations are useful for proving equalities $\sigma = \tau$ between sequences, simulations are very convenient for proving inequalities $\sigma \leq \tau$, where $\leq$ is the prefix ordering which is defined via composition. In order to show $\sigma \leq \tau$, it suffices to show $R(\sigma, \tau)$ for some simulation $R \subseteq A^\infty \times A^\infty$. This R is a simulation in case for all sequences ρ_1, ρ_2, if $R(\rho_1, \rho_2)$ then: if $\mathsf{next}(\rho_1) = (a, \rho_1')$, then $\mathsf{next}(\rho_2) = (b, \rho_2')$ with $a = b$ and $R(\rho_1', \rho_2')$. The difference with a bisimulation is thus that if the first sequence ρ_1 satisfies $\mathsf{next}(\rho_1) = \bot$, then $\mathsf{next}(\rho_2) = \bot$ need not hold. The simulation proof principle can be derived from Fact 1, and is used for most of the results we prove about the prefix order on sequences.

There is a well-known *inductive* proof principle for sequences, which only applies to so-called admissible predicates, see e.g. [Reg95, MN97]. These are predicates which are closed under least upper bounds of chains: $P \subseteq A^\infty$ is admissible if for each ascending chain $\sigma_1 \leq \sigma_2 \leq \sigma_3 \leq \cdots$ of

[6] The sketch of the PVS proof can be found in Section 4.

[7] This property holds for initial algebras and final coalgebras, see for example [JR97].

sequences with $P(\sigma_i)$ for all $i \in \mathbb{N}$, P also holds for the least upper bound of the sequence. If one wishes to prove $\forall \sigma \in A^\infty\ P(\sigma)$ for such admissible predicates P it suffices to prove $P(\sigma)$ for all *finite* sequences σ—*e.g.* by induction on the length. This principle follows from Fact 1 because it can be derived that each sequence is the least upper bound of its finite prefixes. We derive this proof principle, but we do not use it in our formalisation. In other contexts [MN97, Reg95] it is frequently used, because there are syntactic criteria which guarantee that certain predicates are admissible. They are described as "propagations of admissibility" in [Reg95, Subsection 3.5], and form part of the theory HOLCF developed there.

Another definition and proof principle for sequences involves *invariants*. These are predicates $P \subseteq A^\infty$ which satisfy for all $\sigma \in A^\infty$,

$$P(\sigma) \text{ and } \mathsf{next}(\sigma) = (a, \sigma') \text{ implies } P(\sigma').$$

For an arbitrary predicate $P \subseteq A^\infty$ it can be shown that there is a greatest invariant $\mathsf{gi}(P) \subseteq P$ and a least invariant $\mathsf{li}(P) \supseteq P$. In general, the existence of least and greatest invariants is not a consequence of the finality fact. However, if predicates P on X form a complete lattice then so do invariants [Rut96]. This is the case in many logics including PVS. Greatest invariants are a special instance of coinductive definitions using a greatest fixed point operator as in [Pau97a, Pau97b]. Finality of sequences and the fact that homomorphisms preserve invariants [Jac97] provide an appropriate reasoning principle. In order to establish $\mathsf{gi}(P)(\mathsf{coreduce}(\mathsf{struct})(x))$ for a certain structure operation $\mathsf{struct}\colon X \to \mathsf{lift}(A \times X)$ and state $x \in X$ it suffices to determine a struct–invariant $Q \subseteq X$ such that $Q(x)$ implies $P(\mathsf{coreduce}(\mathsf{struct})(x))$. This, perhaps novel, proof principle is demonstrated in Subsection 4.1 and Section 5 in more detail.

This concludes our brief tour of the axiomatic description of sequences as a final coalgebra. We should emphasise that the (extremely useful) proof principles involving (bi)simulation and invariants are naturally part of this coalgebraic setting. In the next section we transfer this axiomatic description from the logic of sets to the logic of types in the verification system PVS, and use it as a basis for an elaborate, fully verified theory of sequences.

4 Coalgebraic sequences in PVS

PVS is a multi–purpose specification and verification system which comprises an expressive language, a theorem prover with powerful built in decision procedures and a user interface facilitating the management of large theories and proofs. The PVS–language is based on higher order logic with predicate subtypes and dependent types. It features, moreover, a mechanism to define inductive datatypes such as finite lists or binary trees of finite depth. Once such a datatype is declared the system generates automatically the corresponding inductive definition schemes and proof principles.

Our coalgebraic formalisation of sequences follows a similar (but dual) approach. The codatatype sequences is essentially characterised by the output signature (codomain type) of the destructor `next`. After this type is fixed we provide associated coinductive definition and proof schemes. These are solely based on the finality axiom for sequences[8] explained in Section 3. We actually do by hand what a system with a codatatype mechanism should do automatically. These basic definitions and theorems are discussed in Subsection 4.1. In particular we explain how the coinductive proof techniques are derived and illustrate the use of the corecursion operator `coreduce`.

In Section 2 we briefly mentioned that sequences form an algebraic complete partial order *w.r.t.* the prefix ordering. This yields additional proof techniques: simulation and induction for admissible predicates which are discussed in Subsection 4.2.

In Subsection 4.3 we introduce the special operations filter and flatten which are useful in many application areas for sequences.

[8] As induction is based on the initiality of an (abstract) datatype in PVS.

Considering the size of the theory package (275 theorems including tcc's[9]) we cannot discuss every aspect of our development. We rather give the reader an impression on the coinductive nature of the sequence formalisation. Therefore we focus on the coinductive construction of sequences and explain the various proof methods by short proof sketches. The complete theory and proof files are publicly available on the WWW [HJ97www].

The subsequent theory and proof fragments use PVS–syntax. Although we touch the main concepts of the PVS–language "on the fly" the reader unfamiliar to PVS may wish to consult the standard literature [ORS93, RSC96].

4.1 Basic definitions

We introduce sequences as an uninterpreted type in a theory parameterised by a type `A`.

```
Seq_defn[A : TYPE] : THEORY
BEGIN
  IMPORTING Lift_prop
  Seq : TYPE
  next : [Seq -> Lift[[A,Seq]]]
END Seq_defn
```

A sequence is determined (as discussed in Section 3) by the observation via the destructor `next` which appears in the above definition as the only basic operation on sequences. This theory expresses one aspect of Fact 1, namely that the sequences together with the destructor operation form a coalgebra. Before we add the finality requirement for this coalgebra (in a separate theory) let us investigate the imported type `Lift`.

```
Lift[X : TYPE] : DATATYPE
BEGIN
  bot : bot?
  up(down : X) : up?
END Lift
```

This is a simple example for a PVS–datatype declaration (see also Section 6). The datatype `Lift` has a parameter type `X`, two constructors `bot` and `up`, the accessor `down`, and two recogniser predicates `bot?` and `up?`. An element `y` of `Lift[X]` is either of the form `y=up(x)` for an element `x` in `X` or of the form `y=bot`. The recogniser predicates distinguish these forms; for instance we have `up?(y)=TRUE` if and only if `y=up(x)` for some `x` in `X`. The accessor (or destructor) `down` maps elements of the form `up(x)` back to `x` in `X`. Using predicate subtypes in PVS it has the type `down:[(up?)->X]`.

The `Lift` datatype provides the lift operation from Section 3, which is needed to describe the partiality of `next` in terms of total functions. Partial functions can be described alternatively using predicate subtypes (as `down`). The representation with `Lift` is simpler and has the advantage that composition of partial functions is very easy to express. The methodological reason for using `Lift` here is, however, more significant: it emphasises the fact that we deal with coalgebras where the codomain of the defining operation is structured. In general a similar development can be undertaken with an arbitrary datatype in the position of `Lift` (see Section 6).

The term `next(s)` for a sequence `s` can now take the intended values: `next(s)=bot` expresses that `s` is empty and `next(s)=up(a,t)` produces the top element `a` and the remainder `t` of a nonempty sequence `s`: these `a` and `t` can be accessed using down as `a=proj_1(down(next(s)))` and `t=proj_2(down(next(s)))`.

We are now in the position to formalise Fact 1 as an axiom for sequences in the theory

```
Seq_ax[A,X : TYPE] : THEORY
```

[9] Typechecking expressions which involve predicate subtypes often lead to nontrivial type correctness conditions which have the same importance as lemmas in the theories, see *e.g.* the definition of `bisim_struct`.

```
BEGIN
  IMPORTING Seq_defn
  struct : VAR [X -> Lift[[A,X]]]
  ...
END Seq_ax
```

In this theory, the conditions 1. and 2. are coded into the predicate

```
 struct_map?(struct) : PRED[[X -> Seq[A]]] =
    LAMBDA(f : [X -> Seq[A]]) : FORALL(x:X) :
      (bot?[[A,X]](struct(x)) IMPLIES bot?[[A,Seq[A]]](next(f(x))))
        AND
      (up?[[A,X]](struct(x)) IMPLIES next(f(x)) =
        up[[A,Seq[A]]](proj_1(down(struct(x))), f(proj_2(down(struct(x))))))
```

where `struct` is an arbitrary operation (coalgebra) on a state set (carrier) `X`. Our approach isolates finality of sequences in one axiom

```
  seq_finality : AXIOM
    EXISTS(f:[X->Seq[A]]) : struct_map?(struct)(f) AND
      FORALL(g:[X->Seq[A]]) : struct_map?(struct)(g) IMPLIES g = f
```

and therefore localises the possible introduction of inconsistencies[10]. For the actual application of finality as a construction and proof method we provide explicitly the corecursion combinator and its basic properties.

```
  coreduce(struct) : {f:[X->Seq[A]] | struct_map?(struct)(f) AND
      FORALL(g:[X->Seq[A]]) : struct_map?(struct)(g) IMPLIES g = f}
```

Of course, this cannot be typechecked automatically—the generated type correctness condition (tcc) requires us to prove that the predicate subtype is nonempty. It can be discharged using the finality axiom.

The function `coreduce` is the only way to define functions taking values in the (uninterpreted) type `Seq`. In order to do so one has to provide a structure map `struct` on an arbitrary state set `X`. Such a structure map can be considered as a machine or automaton with transition function `struct`, whose behavior (given a start state in `X`) is a sequence. Thus, in order to produce a sequence we have to come up with a machine on an appropriate state set. The operation `coreduce` then unravels or unfolds the single step `struct` recursively, producing a sequence (as explained in Section 3). This aspect is revealed in the following (rewrite) lemma which is extracted from the type of `coreduce`

```
  next : LEMMA
    FORALL(x:X) : next(coreduce(struct)(x)) =
      IF bot?(struct(x)) THEN bot
      ELSE up(proj_1(down(struct(x))),
              coreduce(struct)(proj_2(down(struct(x)))))
      ENDIF
```

The uniqueness part of the finality axiom can easily be transformed into the simple coinduction proof methods

[10] This is only of general interest: the validity of the axiom can be shown in PVS: for example, by taking `Seq[A]` to be the type { `f : [nat -> Lift[A]] | FORALL(n : nat) : bot?(f(n)) IMPLIES bot?(f(n+1))` } with suitable `next` coalgebra, we explicitly proved the validity of the `seq_finality` axiom, see the theories `SeqImpl` and `SeqImplFinality`. Of course, other implementations can be used as well. Our motivation for using an axiom (instead of theorem) is to emphasise (1) that no part of the theory that we develop depends on an actual implementation of sequences, and (2) that our approach can be axiomatised easily.

```
  coreduce_unique : LEMMA
    FORALL(g:[X->Seq[A]]) :
      struct_map?(struct)(g) IMPLIES g = coreduce(struct)

  struct_map_unique : LEMMA
    FORALL(g,h:[X->Seq[A]]) :
      struct_map?(struct)(g) AND struct_map?(struct)(h) IMPLIES g = h
```

These proof principles are quite powerful as they can be used to derive the equality of two sequences by showing that they are generated by the same automaton starting from the same state. However, in practice in turns out that coming up with this particular automaton may be difficult.

An equivalent but often more straightforward proof principle is proof by bisimulation which we derive in the theory

```
Bisim[A,X1,X2 : TYPE] : THEORY
BEGIN
  IMPORTING Seq_ax
  struct1 : VAR [X1 -> Lift[[A,X1]]]
  struct2 : VAR [X2 -> Lift[[A,X2]]]
  ...
END Bisim
```

The bisimulation principle in this theory is actually more general than the proof method explained in Section 3: two sequences are equal if they are generated (using `coreduce`) from two bisimilar states of distinct machines `struct1` and `struct2`. A bisimulation is a relation `R:PRED[[X1,X2]]` on the state sets which is appropriately closed under the application of the structure maps: for states `x1` and `x2` with `R(x1,x2)`, one should have:

1. `struct1(x1)=bot IFF struct2(x2) = bot`
2. `struct1(x1)=up(a1,x1')` and `struct2(x2)=up(a2,x2')` implies `a1 = a2` and `R(x1',x2')`

These requirements are packaged in the predicate

```
  bisimulation?(struct1,struct2) : PRED[PRED[[X1,X2]]] = ...
```

Two machine states are bisimilar if there exist such a bisimulation relating them:

```
  bisim?(struct1,struct2) : PRED[[X1,X2]] =
    LAMBDA(x1:X1,x2:X2) : EXISTS(R:PRED[[X1,X2]]) :
      bisimulation?(struct1,struct2)(R) AND R(x1,x2)
```

The bisimulation proof principle is stated in

```
  bisim_finality : LEMMA
    FORALL(x1:X1,x2:X2) : bisim?(struct1,struct2)(x1,x2) IFF
      Kernel(struct1,struct2)(x1,x2)
```

where the Kernel takes the standard definition

```
  Kernel(struct1,struct2) : PRED[[X1,X2]] =
    LAMBDA(x1:X1,x2:X2) : coreduce(struct1)(x1) = coreduce(struct2)(x2)
```

It is the relation consisting of those states which produce the same sequences. To prove one direction of the lemma one checks that `Kernel` is a bisimulation, which is easily done by using the `next` rewrite rule. The other direction requires a bit more work. First of all a bisimulation, viewed (via predicate subtyping) as a state set, carries a machine structure[11]

[11] The proof of the resulting tcc needs the fact that `R` is a bisimulation.

```
bisim_struct(struct1,struct2)(R:(bisimulation?(struct1,struct2))) :
  [(R) -> Lift[[A,(R)]]] = ...
```

composed of `struct1` and `struct2`. The proof proceeds by showing

```
bisim_struct_map : LEMMA
  FORALL(R:PRED[[X1,X2]]) : bisimulation?(struct1,struct2)(R) IMPLIES
    struct_map?(bisim_struct(struct1,struct2)(R))(LAMBDA(z:(R)) :
      coreduce(struct1)(proj_1(incl(struct1,struct2)(R)(z))))
    AND
    struct_map?(bisim_struct(struct1,struct2)(R))(LAMBDA(z:(R)) :
      coreduce(struct2)(proj_2(incl(struct1,struct2)(R)(z))))
```

stating that both `coreduce(struct1)` precomposed with the first projection function `[(R)->X1]` and `coreduce(struct2)` precomposed with the second projection `[(R)->X2]` satisfy the predicate `struct_map?` and must therefore be equal using lemma `struct_map_unique[A,(R)]`.

The proof of equalities of sequences will, in what follows, often employ the `bisim_finality`–Lemma. In comparison to the usage of the lemma `struct_map_unique`, bisimulation proofs only need the construction of a suitable bisimulation relation. The structures `struct1` and `struct2` are often straightforward: in many proofs `struct1=struct2=next` is sufficient. This variant of the bisimulation proof method is supplied in a separate lemma, because it is used most of the time. The bisimulation relation itself is often constructed from the equality to be proven. In general, finding a suitable bisimulation is the heart of an equality proof.

In some cases we face simple equations where the left or the right hand side are of the form `coreduce(struct)(x)`. Then the lemma `coreduce_unique` can be more appropriate because the necessary instantiations are done (almost) automatically and fewer subgoals have to be handled.

So far, the combinator `coreduce` is the only way of constructing sequences. We will proceed by deriving a variety of operations on sequences from it[12].

The finality of sequences implies the fact that `next` is an isomorphism. A candidate for the inverse of `next` is defined coinductively:

```
next_inv_struct : [Lift[[A,Seq[A]]] -> Lift[[A,Lift[[A,Seq[A]]]]]] =
   LAMBDA(z:Lift[[A,Seq[A]]]) :
     IF bot?(z)
     THEN bot
     ELSE up(proj_1(down(z)), next(proj_2(down(z))))
     ENDIF

 next_inv : [Lift[[A,Seq[A]]] -> Seq[A]] = coreduce(next_inv_struct)
```

The actual behavior of `next_inv` becomes clear via

```
 empty_seq : Seq[A] = next_inv(bot[[A,Seq[A]]])

 cons_seq : [[A,Seq[A]] -> Seq[A]] =
   LAMBDA(a:A, x: Seq[A]) : next_inv(up(a,x))
```

These are the (derived !) constructors for sequences; `next_inv` is just a cotupling of them. The proof[13] of

```
 next_iso : LEMMA
   (FORALL(x:Lift[[A,Seq[A]]]) : next(next_inv(x)) = x)
     AND
   (FORALL(y:Seq[A]) : next_inv(next(y)) = y)
```

[12] In the theories `Seq_prop`, `Seq_comp`, and `Seq_functoriality`, see [HJ97www].
[13] Following the outline given in Section 3.

implies the expected rewrite rules for the constructors :

```
next_empty : LEMMA
  next(empty_seq) = bot

next_cons : LEMMA
  FORALL(a:A, x: Seq[A]) : next(cons_seq(a,x)) = up(a,x)
```

Moreover, the lemma `next_iso` entails that `next` is an injective function and, therefore, serves as an additional (non–recursive) proof method.

It is often more convenient to have abbreviations for the output value `at(x,n)` and the remaining sequence `tail(x,n)` after a number `n` of destructions than just the one step destruction by `next`. Obviously `at` is a partial function because accessing a sequence at a position beyond its length must fail. We encode the partiality using (again) the `Lift` datatype:

```
at(x:Seq[A], n:nat) : RECURSIVE Lift[A] =
  IF n=0
  THEN IF bot?(next(x)) THEN bot
       ELSE up(proj_1(down(next(x))))
       ENDIF
  ELSE IF bot?(next(x)) THEN bot
       ELSE at(proj_2(down(next(x))),n-1)
       ENDIF
  ENDIF
  MEASURE (LAMBDA(x:Seq[A], n:nat) : n)
```

Finite observations on possibly infinite sequences are typically defined using induction on natural numbers. PVS provides for this purpose a definition scheme for bounded recursion which ensures that the recursion will terminate. The operation

```
tail(x:Seq[A], n:nat) : RECURSIVE Seq[A] = ...
```

is defined similarly but tail returns the empty sequence instead of `bot` in the case of failure. These operations behave as intended—we prove (by induction) a variety of rewrite rules for instance:

```
at_empty : LEMMA
  FORALL(n:nat) : at(empty_seq,n) = bot

tail_cons : LEMMA
  FORALL(x:Seq[A], a:A, n:nat) : tail(x,n) = tail(cons_seq(a,x),n+1)

at_tail : LEMMA
  FORALL(x:Seq[A], n,m:nat) : at(tail(x,n),m) = at(x,n+m)
```

In Section 2 we suggested

```
at_eqn : LEMMA
  FORALL(x,y:Seq[A]) : (FORALL(n:nat) : at(x,n) = at(y,n)) IMPLIES x = y
```

as (yet another) rule for proving the equality of two sequences. This rule formalises the intuition that two sequences are equal if they cannot be distinguished by finite observations. In fact, this property is proved via the bisimulation relation

```
at_eqn_rel : PRED[[Seq[A],Seq[A]]] = {z: [Seq[A], Seq[A]] |
    FORALL (n: nat): at(PROJ_1(z), n) = at(PROJ_2(z), n)}

at_eqn_rel_bisim : LEMMA
  bisimulation?(next[A],next[A])(at_eqn_rel)
```

For the proof of `at_eqn_rel_bisim` we have to handle the following goal which is obtained by expanding the definitions and simplification.

```
{-1}    FORALL (n: nat): at(x1!1, n) = at(x2!1, n)
  |-------
{1}    IF bot?(next[A](x1!1))
        THEN IF bot?(next[A](x2!1)) THEN TRUE
        ELSE FALSE
        ENDIF
       ELSE IF bot?(next[A](x2!1)) THEN FALSE
        ELSE
          (PROJ_1(down(next[A](x1!1))) = PROJ_1(down(next[A](x2!1)))
           AND FORALL (n: nat):
               at(PROJ_2(down(next[A](x1!1))), n)
                   = at(PROJ_2(down(next[A](x2!1))), n))
        ENDIF
       ENDIF
```

Here the constants of the form `x1!1` are skolem constants generated by PVS while skolemising universally quantified terms. The proof proceeds by a case analysis. If `bot?(next(x1!1))` we can conclude from the assumption `-1` and the lemma `at_empty` that `bot?(next(x2!1))` holds and vice versa. In the case that both `x1!1` and `x2!1` are not empty we obtain the following goals

```
at_eqn_rel_bisim.2.2 :

[-1]    FORALL (n: nat): at(x1!1, n) = at(x2!1, n)
  |-------
{1}    PROJ_1(down(next[A](x1!1))) = PROJ_1(down(next[A](x2!1)))
{2}    bot?(next[A](x2!1))
[3]    bot?(next[A](x1!1))

at_eqn_rel_bisim.2.3

[-1]    FORALL (n: nat): at(x1!1, n) = at(x2!1, n)
  |-------
{1}    at(PROJ_2(down(next[A](x1!1))), n!1)
          = at(PROJ_2(down(next[A](x2!1))), n!1)
{2}    bot?(next[A](x2!1))
[3]    bot?(next[A](x1!1))
```

The first one disappears by applying the assumption `-1` instantiated with `0`. The second one uses the assumption instantiated with `n!1+1`.

The functions `at` and `tail` find many applications in our formalisation because they resemble notions familiar from (finite) lists. Similar operations for more sophisticated coalgebraic datatypes such as possibly infinitely branching trees with possible infinite depth [HJ97] might be more difficult. In these cases the general notions of shape and position [Jay96] will occur. Here the shape[14] (length) of a sequence is either a natural number if it is finite or `bot` for infinity and a position is a natural number.

Another useful basic operation is composition or concatenation of two sequences explained in Section 3.

```
  comp_struct : [[Seq[A],Seq[A]] -> Lift[[A,[Seq[A],Seq[A]]]]] =
    LAMBDA(x,y:Seq[A]) :
      IF bot?(next(x))
```

[14] The theories `Seq_shape` and `Seq_zip` contain the definition of shape for sequences and some basic properties.

```
    THEN IF bot?(next(y)) THEN bot
         ELSE up(proj_1(down(next(y))),(empty_seq,proj_2(down(next(y)))))
         ENDIF
    ELSE up(proj_1(down(next(x))),(proj_2(down(next(x))),y))
    ENDIF

comp : [[Seq[A],Seq[A]] -> Seq[A]] = coreduce(comp_struct)
```

A state of the machine `comp_struct` consists of pairs of sequences `x,y:Seq[A]`. If the first sequence `x` is empty then the output of the machine depends on the output of the second sequence `y` and the successor state consists of `x` and the successor state of `y`. If `x` is not empty, we observe its first element; the successor state of the machine consists of the successor of `x` and `y`. Such a machine can be considered as the sequential composition of the machines generating `x` and `y`. The observed behavior is therefore the concatenation of the given behaviors.

The composition operation (as `append` for finite lists) has the empty sequence as its left and right unit and is associative:

```
comp_assoc : LEMMA
  FORALL(x,y,z:Seq[A]) : comp(x,comp(y,z)) = comp(comp(x,y),z)
```

The proof of this lemma is a simple benchmark for a sequence formalisation (see *e.g.* [Pau97a]). The standard coinductive proof uses the bisimulation

```
R = {u,v: Seq[A] |
       EXISTS(x,y,z:Seq[A]) :
        u = comp(x, comp(y, z)) AND v = comp(comp(x, y), z)}
```

which is the default bisimulation to be extracted from an equality. The proof follows in principle the same structure as the one for the lemma `at_eqn_rel`. The only complication arises because of the existential quantification inside `R`: we have to check for nonempty `u` and `v` that `u=comp(x, comp(y, z)) AND v= comp(comp(x, y), z)` implies

```
EXISTS(x,y,z:Seq[A]) :
  proj_2(down(next(u))) = comp(x, comp(y, z))
  AND
  proj_2(down(next(v))) = comp(comp(x, y), z)
```

The instantiations for the existential quantifier depend on the actual value of `x`, `y`, and `z`. For instance, in the case that `x` is empty but `y` is not, the destruction of the composition is determined by the destruction of `y`. Therefore we instantiate with the triple `(x,proj_2(down(next(y))),z)`. By a case distinction on what component of the composition is "active" (*i.e.* providing the next observation) and a choice of the suitable instantiation we complete the proof.

Similar existentially quantified formulas arise during all proofs using a default bisimulation such as `R`. As a rule of thumb the user may first determine the active variable (in the above sense) and then instantiate its position with the successor state. This heuristic works well for a great number of proofs. In Subsection 4.3 we will sketch a proof with a more sophisticated instantiation in this proof step.

An interesting aspect of composition is the way how it deals with infinite values. For instance, we prove

```
infinite_comp1 : LEMMA
  FORALL(x,y:Seq[A]) : infinite?(x) IMPLIES comp(x,y) = x
```

which meets our intuition that sequences are characterised by the finite observations one can perform. In Subsection 4.2 composition will be used for the definition of the prefix order on sequences.

We defined the sequences type in a parameterised theory. Therefore `Seq` can be considered as a construction which given a type `A` returns the type of sequences of elements of `A`, namely

`Seq[A]`. The function `seq_map(f):[Seq[A]->Seq[B]]` is a natural extension of this construction to functions `f:[A ->B]` applying `f` to each element of a sequence. This function takes values in `Seq[B]` and is, therefore, defined coinductively:

```
seq_map_struct(f:[A->B]) : [Seq[A] -> Lift[[B,Seq[A]]]] =
  LAMBDA(x:Seq[A]) :
    IF bot?(next(x)) THEN bot
    ELSE up(f(proj_1(down(next(x)))), proj_2(down(next(x))))
    ENDIF

seq_map(f:[A->B]) : [Seq[A] -> Seq[B]] =  coreduce(seq_map_struct(f))
```

The extension of `Seq` to a construction on functions satisfies various (standard) properties[15]: it preserves identities

```
seq_map_id : LEMMA
  FORALL(x:Seq[A]) : seq_map(id[A])(x) = x
```

and composition of functions

```
seq_map_comp : LEMMA
  FORALL(f:[A->B],g:[B->C],x:Seq[A]) :
    seq_map(g o f)(x) = seq_map(g)(seq_map(f)(x))
```

The map function furthermore preserves injectivity of functions

```
seq_map_pres_inj : LEMMA
  FORALL(f:[A->B]) : injective?(f) IMPLIES injective?(seq_map(f))
```

and therefore the `Seq`–construction permits the lifting of predicates.

We conclude this subsection with the formalisation of invariants on sequences and the corresponding proof principles. Following Section 3, invariants are predicates on sequences which are stable under destruction by `next`

```
invariant? : PRED[PRED[Seq[A]]] = LAMBDA(P:PRED[Seq[A]]) :
  FORALL(x:Seq[A]) : P(x) IMPLIES P(tail(x))
```

where `tail(x)` is an abbreviation for `tail(x,1)`. For an arbitrary predicate `P` the predicate `gi(P)` defined by

```
gi : [PRED[Seq[A]] -> PRED[Seq[A]]] =
  LAMBDA(P:PRED[Seq[A]]) : {x:Seq[A] | FORALL(n:nat) : P(tail(x,n))}
```

is the greatest invariant contained in `P`:

```
gi_invariant : LEMMA
  FORALL(P:PRED[Seq[A]]) : invariant?(gi(P))

gi_greatest : LEMMA
  FORALL(P,Q:PRED[Seq[A]]) :
    invariant?(Q) AND (FORALL(x:Seq[A]) : Q(x) IMPLIES P(x))
      IMPLIES (FORALL(x:Seq[A]) : Q(x) IMPLIES gi(P)(x))
```

Least invariants are defined in a dual fashion. Invariants correspond to safety properties; a sequence satisfies an invariant if the invariant holds at all of its tails. The greatest invariant is therefore the most general safety property entailing a given predicate. We can generalise invariants to arbitrary machines `struct`:

[15] Turning `Seq` into an endofunctor on the category of sets and total functions.

```
invariant?(struct) : PRED[PRED[X]] =
  LAMBDA(P:PRED[X]) : FORALL(x:X) :
    P(x) AND up?(struct(x)) IMPLIES P(proj_2(down(struct(x))))
```

Thus, an invariant is closed under application of `struct`, implying that if `x` is in `P` then `P` must contain all states that are reachable from `x` (via `struct`).

In particular, we can now specify a "local" predicate[16] `P` on a sequence by observation on a finite number of single `next` steps. The greatest invariant extends `P` to all reachable states in the weakest way—it globalises `P`. For instance, the predicate of those sequences all of whose elements satisfy a predicate `Q` on the parameter set `A` can be defined as the greatest invariant

```
gi({x: Seq[A] | up?(next(x)) IMPLIES Q(proj_1(down(next(x))))})
```

of a predicate which only involves single steps. This predicate gives a characterisation[17] of the type `Seq[(Q)]`. The greatest invariant definition has the advantage that it comes equipped with a proof principle, see the next lemma below. Another example is the predicate `ordered?` used in Subsection 5.2.

Least and greatest invariant definitions are a special instance of (co)inductive definitions using least and greatest fixedpoints of monotone operators on bounded sets (*i.e.* predicates) as in [Pau97b]. The `coreduce` combinator can only define functions taking values in `Seq[A]`. Thus, least and greatest invariants extend the expressiveness of our approach. The combination of the finality of sequences, the maximality of the greatest invariants, and preservation of invariants by homomorphisms of machines provides a coinductive proof principle for greatest invariants:[18]

```
struct_gi_greatest : LEMMA
   FORALL(P:PRED[X],Q:PRED[Seq[A]]) :
     (invariant?(struct)(P) AND FORALL(x:X) : P(x)
                                 IMPLIES Q(coreduce(struct)(x)))
       IMPLIES
     (FORALL(x:X) : P(x) IMPLIES gi(Q)(coreduce(struct)(x)))
```

If there is an invariant `P` on the generating automaton and it implies (translated by `coreduce`) the local predicate `Q`, then `P` implies the greatest invariant of `Q` as well. We will use this method in Subsection 5.2 to prove that a certain insert operation on sequences preserves ordering.

4.2 Prefix ordering: simulations and a proof principle for admissible predicates

Thus far we have discussed coinductive proof methods for checking equalities of sequences. This subsection is concerned with the prefix ordering on sequences mentioned in Section 3 and discusses related proof techniques and results in our formalisation.

The prefix ordering on sequences relies on the composition operation

```
prefix : PRED[[Seq[A],Seq[A]]] =
  LAMBDA(x,y:Seq[A]) : EXISTS(z:Seq[A]) : comp(x,z) = y
```

We prove that `prefix` is a partial order and introduce, for convenience, various rewrites, for instance

```
prefix_tail : LEMMA
  FORALL(x,y:Seq[A]) : prefix(x,y) IMPLIES
    FORALL(n:nat) : prefix(tail(x,n),tail(y,n))

prefix_infinite : LEMMA
  FORALL(x,y:Seq[A]) : (infinite?(x) AND prefix(x,y)) IMPLIES x = y
```

[16] In general, `P` can be any predicate on sequences.
[17] Similar to the `every` predicate, which is generated for ADT definitions in PVS.
[18] This is a special case of an invariant proof principle in [Jac97].

The latter emphasises again that equality of sequences is determined by finite observation: if `x` is an infinite prefix of `y` then it is indistinguishable from `y`. Note, moreover, that the empty sequence is a prefix of every sequence.

In the same way as bisimulations serve as a local (single step) proof method for equalities, simulations provide means for proving the prefix ordering. A simulation[19] is a relation on sequences satisfying

```
simulation? : PRED[PRED[[Seq[A],Seq[A]]]] =
  LAMBDA(R:PRED[[Seq[A],Seq[A]]]) :
    FORALL(x,y:Seq[A]) : R(x,y) IMPLIES
      (up?(next(x)) IMPLIES
         (up?(next(y)) AND proj_1(down(next(x))) = proj_1(down(next(y)))
                       AND R(proj_2(down(next(x))),proj_2(down(next(y))))))
```

A simulation ensures that every step which can be taken by the first sequence can be matched by a similar step of the related sequence. The simulation proof principle

```
prefix_simulation : LEMMA
  FORALL(x,y:Seq[A]) : prefix(x,y) IFF
    EXISTS(R:PRED[[Seq[A],Seq[A]]]) : simulation?(R) AND R(x,y)
```

follows from an application of the bisimulation proof principle. For the (only if) direction we show that the `prefix` relation is a simulation. Vice versa, we perform a case distinction on whether `x` is finite or infinite. In the finite case, the sequence `y` can be cut into the finite start with the same length as `x` and a remainder which witnesses, as desired, the prefix order. If `x` is infinite the relation

```
{x,y:Seq[A] | R(x,y) AND infinite?(x)}
```

is a bisimulation—following directly from the simulation properties of `R`. The empty sequence (as any other sequence would do as well) then witnesses the prefix order.

Using the prefix order one can describe ascending chains `AChains?[Seq[A],prefix[A]]` of sequences as functions `f:[nat -> Seq[A]]` such that `prefix(f(n),f(n+1))` holds[20] for each `n`. Interestingly, we can now define the least upper bound of an ascending chain by coinduction

```
lub_struct : [(AChains?[Seq[A],prefix[A]]) ->
                Lift[[A,(AChains?[Seq[A],prefix[A]])]]] =
  LAMBDA(f:(AChains?[Seq[A],prefix[A]])) :
    IF FORALL(n:nat) : bot?(next(f(n))) THEN bot
    ELSE up(proj_1(down(next(f(least({n:nat|NOT bot?(next(f(n)))}))))),
            LAMBDA(n:nat) : tail(f(n),1))
    ENDIF

lub : [(AChains?[Seq[A],prefix[A]]) -> Seq[A]] = coreduce(lub_struct)
```

The state space of the automaton `lub_struct` is the set of ascending chains. The destruction of the chain of empty sequences just yields `bot`. Otherwise we can observe the head of any nonempty chain element (for convenience we take the least). The new state is computed by the pointwise application of `tail`. Now we can prove

```
seq_lub : LEMMA
  FORALL(f:(AChains?[Seq[A],prefix[A]])) :
    least_upperbound?[Seq[A],prefix[A]](f)(lub(f))
```

[19] The definitions of bisimulation and of invariant are determined by the functor T of the coalgebras that one is considering. The notion of simulation is determined in a similar manner if there is an order on the objects $T(X)$. It is described as an "ordered bisimulation" in [Fio96]. Our notion of simulation for sequences is an instance of an ordered bisimulation, using the flat order on the functor $T(X) = 1 + (A \times X)$.

[20] We could have used as well infinite sequences of sequences defining the ascending chain property as a greatest invariant similarly to `ordered?` in Subsection 5.2.

The simulation proof principle shows that `lub(f)` is an upper bound using the simulation (for all `n`)

```
{ x,y:Seq[A] |  EXISTS(m:nat) : x = tail(f(n),m) AND
                                 y = tail(lub(f),m)}
```

The same proof principle provides minimality: suppose there is another upper bound `z` of the chain `f` then the relation

```
{x,y:Seq[A] |
  EXISTS(m:nat) :
   x = lub(LAMBDA(n:nat) : tail(f(n),m)) AND
   y = tail(z,m)}
```

is a simulation yielding `prefix(lub(f),z)`. Therefore the prefix relation yields a complete partial order. Moreover, we prove that every sequence is the least upper bound of the chain of its finite prefixes. Together with the result that states that each sequence is finite if and only if it is a compact element (see [DP96]) we conclude that the prefix ordering gives rise to an algebraic cpo.

This result implies another proof principle for sequences for admissible predicates. These are predicates which are closed under least upper bounds:

```
admissible? : PRED[PRED[Seq[A]]] =
  LAMBDA(p:PRED[Seq[A]]) : FORALL(f:(AChains?[Seq[A],prefix[A]])) :
    (FORALL(n:nat) : p(f(n))) IMPLIES p(lub(f))
```

We prove the induction principle

```
admissible_holds : LEMMA
  FORALL(p:PRED[Seq[A]]) :
    (admissible?(p) AND FORALL(l:list[A]) : p(list_incl(l)))
       IMPLIES FORALL(x:Seq[A]) : p(x)
```

If an admissible predicate holds for all finite sequences (built up from lists) it therefore holds for arbitrary sequences being a least upper bound of a chain of finite sequences. Thus, in order to verify that an admissible predicate holds, induction on (finite) lists is sufficient. This (in our approach derived) proof rule is fundamental in the domain theoretic formalisation of sequences as presented in [MN97]. Note, however, that the application of this induction principle involves the proof of admissibility of the predicate in question. As long as we cannot guarantee this property by extra simple (syntactic) criteria (as in [Reg95, MN97]) the induction principle is of little use in our framework.

4.3 Filtering and flattening

Filtering and flattening of (finite) lists are frequently used standard operations. Filtering removes all elements of a list which do not satisfy a given predicate. A list of lists is flattened by removing the inner brackets. For finite lists both inductively defined operations do (naturally) terminate.

This picture changes for possibly infinite sequences. We can not expect that an algorithm for filtering does terminate for all sequences, because for a given infinite sequence we might not be able to determine in finitely many steps the first element for which the predicate in question holds. The same is, less obviously, valid for flattening. Suppose we want to calculate the first element of a flattened infinite sequence of lists. This requires the calculation of the first nonempty element of the sequence (and then the first element of this list) which reduces to the filter problem. In the following we will focus on filtering.

Filtering (as well as flattening) is defined coinductively (because their codomain is `Seq[A]`). Recall that this requires us to determine a local one step structure map on the domain of the operation. The structure map has to incorporate the search for the first element for which a predicate holds. Instead of "programming" a search algorithm we use (declaratively) an oracle which tells us the desired position if it exists and otherwise returns `bot`.

```
holds_first_at : [[PRED[A],Seq[A]] -> Lift[nat]] =
  LAMBDA(p:PRED[A],x:Seq[A]) :
    IF (FORALL(n:nat) : up?(at(x,n)) IMPLIES NOT p(down(at(x,n))))
    THEN bot
    ELSE up[nat](least({ n : nat | up?(at(x,n)) AND p(down(at(x,n)))}))
    ENDIF
```

This oracle is of course not operational, instead the clause `(FORALL(n:nat) : ...` is descriptive. Contrary to Paulson [Pau97a] we think that such a desciptive filter is relevant to computer science application. We indicate its usefuleness for the development of meta theory in Subsection 5.1. Note moreover that the negation of the descriptive clause in above definition characterises a subset of sequences which are fair *w.r.t.* the given predicate as indicated in [Rei96]. Restricted to such fair sequences the oracle has computational meaning.

The oracle is the essential part of the structure map for filtering:

```
filter_struct : [[PRED[A],Seq[A]] -> Lift[[A,[PRED[A],Seq[A]]]]] =
  LAMBDA(p:PRED[A], x:Seq[A]) :
    IF bot?(holds_first_at(p,x)) THEN bot
    ELSE up(down(at(x,down(holds_first_at(p,x)))),
            (p,tail(x,1+down(holds_first_at(p,x)))))
    ENDIF

filter : [[PRED[A],Seq[A]] -> Seq[A]] = coreduce(filter_struct)
```

In words, `holds_first_at` returns a natural number (if it exists) which is appropriately fed into `at` and `tail` to calculate the next valid output and the successor state. Again for efficiency we provide various rewrite rules for `filter`. For instance:

```
filter_cons : LEMMA
  FORALL(p:PRED[A], x:Seq[A], a:A) : filter(p,cons_seq(a,x)) =
    IF p(a)
    THEN cons_seq(a,filter(p,x))
    ELSE filter(p,x)
    ENDIF
```

The filter operation does not commute with composition (concatenation) of arbitrary sequences, *i.e.* the equation

```
filter(p,comp(x,y)) = comp(filter(p,x),filter(p,y))
```

does not hold in general (as discussed in Section 7). Nevertheless, if `x` is finite[21] the above equation holds:

```
filter_comp : LEMMA
  FORALL(l:list[A],x:Seq[A],p:PRED[A]) :
    filter(p,comp(list_incl(l),x)) =
      comp(filter(p,list_incl(l)),filter(p,x))
```

The proof uses a simple induction on lists and the filter rewrites we supplied.

The lemma

```
filter_and : LEMMA
  FORALL(x:Seq[A],p,q:PRED[A]) : filter(p,filter(q,x)) =
    filter((LAMBDA(a:A) : p(a) AND q(a)),x)
```

has a more challenging coinductive proof. We show that the standard candidate

[21] More generally `x` should be fair *w.r.t.* the filter predicate—here we prove just the finite case.

```
{u,v:Seq[A] | EXISTS(x:Seq[A]) :
              u = filter(p, filter(q, x)) AND
              v = filter((LAMBDA (a: A): p(a) AND q(a)), x)}
```

is a bisimulation. Firstly, the left hand side of the equality is empty if and only if the right hand side is empty by expanding the definitions of `filter` and using properties of `holds_first_at`. Secondly if both sides are not empty we have to check that the observable output (given by `next`) is the same and the next states are again related. This involves a somewhat complicated reasoning about those positions in `x` and `filter(q,x)` where the predicates hold the first time. We describe just one (simple) case. Suppose the first position n_q where `q` holds in `x` is strictly less than $n_{p \wedge q}$ (the first position for `p` and `q`). Further suppose between n_q and $n_{p \wedge q}$ exist m elements for which `q` does not hold. Then $n_{p \wedge q} = n_q + m + k_p$ where k_p is the first position in `filter(q,x)` for which `p` holds as well. This equation can now be used to calculate the outputs and instantiations.

Our (simply typed) filter function can be typed more accurately by using dependent types

```
filter_pred(p:PRED[A]) : [Seq[A] -> Seq[(p)]] =
  coreduce(filter_pred_struct(p))
```

where `filter_pred_struct` is defined similarly (but with tighter typing) to `filter_struct`. The original filter function can be obtained from `filter_pred` but not vice versa:

```
pred_incl(p:PRED[A]) : [(p) -> A] = LAMBDA(x:(p)) : x

seq_pred_incl(p:PRED[A]) : [Seq[(p)] -> Seq[A]] =
  seq_map(pred_incl(p))

filter_pred_filter : LEMMA
  FORALL(p:PRED[A],x:Seq[A]) :
    seq_pred_incl(p)(filter_pred(p)(x)) = filter(p,x)
```

However, most of the properties of `filter` are reflected by `seq_pred_incl` using the fact from Subsection 4.1 that `seq_map` preserves injectivity.

Filtering sequences does commute with `seq_map` in the following way

```
filter_map : LEMMA
  FORALL (f: [A -> B], p: PRED[B], x: Seq[A]):
    filter(p,seq_map(f)(x)) = seq_map(f)(filter((p o f),x))
```

Note that a predicate `p: PRED[B]` gives rise to a predicate `p o f: PRED[A]` by using the "translation" `f`. This lemma looks more complicated for `filter_pred`

```
filter_pred_map : LEMMA
  FORALL (f: [A -> B], p: PRED[B], q: PRED[A], x: Seq[A], g:[(q)->(p)]):
     q = p o f AND f o pred_incl(q) = pred_incl(p) o g
     IMPLIES
     filter_pred(p)(seq_map(f)(x)) = seq_map(g)(filter_pred(q)(x))
```

because the dependent types are taken into account. The proof of the refinement lemma in Subsection 5.1 crucially relies on it.

Finally, the flatten function reads

```
flatten_struct : [Seq[list[A]] -> Lift[[A,Seq[list[A]]]]] =
  LAMBDA(x:Seq[list[A]]) :
    LET y = holds_first_at((LAMBDA(l:list[A]) : NOT l=null),x) IN
      IF bot?(y) THEN bot
      ELSE up(car(down(at(x,down(y)))),
              cons_seq(cdr(down(at(x,down(y)))),
```

```
                        tail(x,1+down(y))))
      ENDIF

flatten : [Seq[list[A]] -> Seq[A]] = coreduce(flatten_struct)
```

The heart of `flatten_struct` is again the oracle `holds_first_at` for a predicate which checks if a list is nonempty. The main `flatten`–lemma is

```
filter_flatten : LEMMA
  FORALL(x:Seq[list[A]],p:PRED[A]) :
    filter(p,flatten(x)) =
      flatten(seq_map(LAMBDA(l:list[A]) : list_filter(p,l))(x))
```

which shows that filtering and flattening commute. The proof is akin to the lemma `filter_and` because again we have to handle the filtering of a conjunction of two predicates.

This concludes our brief tour through our sequence formalisation. We present applications of these notions and techniques in the next section.

5 Two challenges

In this section we apply the coalgebraic sequence formalisation to two areas: automata theory and functional programming. Sequences occur in automata theory as executions and traces. Executions are alternating sequences of states and actions which record the internal state changes of an automaton. A trace is obtained from an execution by filtering out the information which determines the externally visible behavior of an execution. In our case, where we distinguish external and internal actions[22] a trace consist solely of external actions. Using a trace semantics, an automaton A implements an automaton B if every trace of A can be performed by B, that is the set of traces of B is a superset of the set of traces of A. A refinement is a function between the state sets of two automata which commutes with the transition relation of the automata in a suitable way. The refinement lemma then states that the existence of a refinement implies trace inclusion. This is an important proof method because local reasoning is sufficient for a refinement proof. The refinement lemma is a (nontrivial) illustration for the use of filtering and mapping of sequences.

CHARITY [CF92, CS95] is a programming language which is functional in style and is solely based on induction for (initial) datatypes and coinduction for (final) codatatypes. A codatatype for sequences supports definitions in the style of `coreduce`. The second challenge we briefly discuss in this section involves a coinductive definition of an insert operation for sequences *w.r.t.* an order predicate on the elements. This insert operation can be programmed in CHARITY (using the `coreduce` equivalent). In PVS we prove its correctness: inserting an element preserves ordering. Interestingly the ordering predicate is a greatest invariant, and the invariant proof principle yields the correctness proof. This example appropriately combines coinductive techniques for programming and reasoning.

5.1 Refinement lemma from automata theory

We consider automata with state space `S` as transition systems with two label sorts `Ext` and `Int` for external and internal actions:

```
IEaut : TYPE = [# trans? : PRED[[S, Coprod[Int,Ext], S]], start? : PRED[S] #]
```

Here `Coprod` is the disjoint union defined as a PVS datatype with constructors (injections) `in1` and `in2`. The type `IEaut` is a record type. The record projection `trans?` accesses the transition relation, `start?` the set of start states. An automaton `a` of type `IEaut` can perform internal

[22] In I/O–automata theory [LV95] the external actions are further partitioned into input and output actions.

transitions `(s,in1(i),t)` in `trans?(a)` or external transitions `(s,in2(e),t)` in `trans?(a)` which can be distinguished by the tags `in1` and `in2` of the coproduct.

For the purpose of the formalisation of the refinement lemma we alter the standard definition of an execution: instead of considering alternating sequences of states and actions starting with a state in `start?`, an ("extended") execution has the type

```
Seq[[S, Coprod[list[Int], [list[Int],Ext,list[Int]]], S]]
```

Its elements are triples `(s,x,t)` where `s` and `t` are states in `S` connected via the expanded transition `x`. The expanded transition consists either of a list of silent (internal) actions of the considered automaton or of an external action wrapped in two lists of internals. Intuitively, the notion of an expanded transition resembles the $\stackrel{\sigma}{\Rightarrow}$ notation from automata theory where σ is, in our case, either the empty or singleton sequence of visible actions.

The predicate

```
expand(trans?) : PRED[[S, Coprod[list[Int], [list[Int],Ext,list[Int]]], S]]]
```

ensures that the triples are indeed (expanded) transition steps of a given transition relation `trans?`. Executions `exec?(aut)` are then defined as the greatest invariant of the (local) predicate

```
exec0?(trans?(aut)) : PRED[Seq[(expand?(trans?(aut)))]]
```

which guarantees that the third component of a (triple) element matches the first of the next sequence element (if it exists). Finally, executions whose first state is a start state of `aut` are collected in `execof?(aut)`. To obtain a trace from an execution we first forget about the state information, then filter out all internal steps, and finally project to a sequence of externals:

```
is_ext?(aut) : PRED[(expand(trans?(aut)))] =
  LAMBDA(z:(expand(trans?(aut)))) :
     is2?[list[Int], [list[Int],Ext,list[Int]]](proj_2(z))

remove_ints(aut) : [Seq[(is_ext?(aut))] -> Seq[Ext]] =
  seq_map(LAMBDA(w:(is_ext?(aut))) : proj_2(out2(proj_2(w))))

exec2trace(aut) : [(exec?(aut)) -> Seq[Ext]] =
  remove_ints(aut) o filter_pred(is_ext?(aut))
```

Observe that we use `filter_pred`, which produces a sequence of the subtype of expanded external steps, instead of `filter`, which would not allow the appropriate composition of functions. The predicate `traces?(aut)` consists then of those sequences which are filtered from an execution of `aut`.

The structure of the extended executions reflects the notion of refinement between two automata:

```
refinement?(aut1,aut2) : PRED[[[S1->S2],[Ext1->Ext2]]] =
  LAMBDA(r:[S1->S2],f:[Ext1->Ext2]) :
    (FORALL(s:S1) : start?(aut1)(s) IMPLIES start?(aut2)(r(s))) AND
    (FORALL(s,t:S1) :
      (FORALL(a:Int1) : trans?(aut1)(s,in1(a),t) IMPLIES
         silent?(trans?(aut2))(r(s),r(t)))
         AND
      (FORALL(a:Ext1) : trans?(aut1)(s,in2(a),t) IMPLIES
         EXISTS(u,v:S2) : silent?(trans?(aut2))(r(s),u) AND
                          trans?(aut2)(u,in2(f(a)),v) AND
                          silent?(trans?(aut2))(v,r(t))))
```

A refinement following this definition, is a pair of a mapping `r` between the state spaces and a translation[23] `f` between the externally visible actions preserving the start states and satisfying

[23] This generalises refinements to automata with possibly different label sets.

1. If the automaton `aut1` can perform a silent action `in1(a)` then the automaton `aut2` matches this action with a sequence of silent actions.

2. If `aut1` performs an external action `in2(a)` then `aut2` matches this action by an expanded transition containing the translated action `in2(f(a))`.

The refinement lemma now states that the existence of an refinement implies trace inclusion *w.r.t.* the translation `f` of the refinement:

```
FORALL (x:(traces(aut1))) : traces(aut2)(seq_map(f)(x))
```

In the sequel we collect two automata related by a refinement in the structure

```
auts_ref : VAR [# aut1 : IEaut[S1,Int1,Ext1],
                  aut2 : IEaut[S2,Int2,Ext2],
                  ref : (refinement?(aut1,aut2)) #]
```

The proof proceeds in three steps

1. Using the refinement properties we define a function

   ```
   translate_steps(auts_ref) :
     [(expand(trans?(aut1(auts_ref)))) -> (expand(trans?(aut2(auts_ref))))]
   ```

   which translates an expanded transition step of `aut1` to an expanded transition step of `aut2` and preserves the distinction between internal and external (translated by `f`) steps.

2. This translation is extended to sequences via `seq_map`

   ```
   seq_refine2exec(auts_ref) : [Seq[(expand(trans?(aut1(auts_ref))))] ->
                                  Seq[(expand(trans?(aut2(auts_ref))))]] =
     seq_map(translate_steps(auts_ref))
   ```

   which preserves the greatest invariant

   ```
   seq_refine2exec_in_exec : LEMMA
     FORALL(x:(exec?(aut1(auts_ref)))) :
       exec?(aut2(auts_ref))(seq_refine2exec(auts_ref)(x))
   ```

3. The proof succeeds by pasting together the definitions of `seq_refine2exec` and `exec2trace` and applying the lemmas `seq_map_comp` and `filter_pred_map` from Subsection 4.1.

5.2 Sorted sequences

A sequence of elements of type `A` is ordered *w.r.t.* a relation `<=` on `A` if

```
ordered?(<=) : [Seq[A] -> bool] =
 LAMBDA (s:Seq[A]) : FORALL (n:nat, a1,a2:A) :
    at(s,n) = up(a1) AND at(s,n+1) = up(a2)
    IMPLIES a1<=a2
```

holds for it. This ad hoc definition is, in fact, a greatest invariant of the local predicate

```
ord_local(<=) : [Seq[A] -> bool] =
 LAMBDA (s:Seq[A]): FORALL (a1,a2:A) :
   at(s,0) = up(a1) AND at(s,1) = up(a2)
   IMPLIES a1 <= a2
```

This (simple) observation supplies us with the greatest invariant proof principle for checking if a sequence is ordered.

We consider an operation which inserts an element in a given sequence without disturbing the ordering given by <=. This insert operation is defined coinductively using the following machine (resembling the definition from [Coc96]).

```
pd_struct(<=) : [[Lift[A],Seq[A]] -> Lift[[A,[[Lift[A],Seq[A]]]]]]=
  LAMBDA (x:Lift[A], s:Seq[A]) :
   IF bot?(x) THEN
      IF bot?(next(s)) THEN bot
      ELSE up(proj_1(down(next(s))), (bot,proj_2(down(next(s)))))
      ENDIF
   ELSE LET v = down(x) IN
    IF bot?(next(s)) THEN up(v, (bot,empty_seq))
    ELSE
      LET a = proj_1(down(next(s))), t = proj_2(down(next(s))) IN
        IF a <= v THEN up(a, (up(v), t))
        ELSE up(v, (bot, cons_seq(a,t)))
        ENDIF
    ENDIF
   ENDIF
```

The first component of the state space is either `bot` signaling that no further insert is needed, or `up(a)` where `a` is an element which still should be inserted (at some later stage). In the case of `bot` the machine reproduces the behavior of the given sequence. Otherwise the machine tries the insertion at the current state. If the given sequence is empty or its head element is not related to `a` then the insertion can take place: we produce `a` as the output and proceed with the original sequence and `bot` in the first component. If the insertion cannot be performed we postpone it to the next step.

We now prove that the resulting insert operation

```
pd(<=) : [[Lift[A],Seq[A]] -> Seq[A]] = coreduce(pd_struct(<=))

push_down(<=) : [[A,Seq[A]] -> Seq[A]] =
  LAMBDA (a:A, s:Seq[A]) : pd(<=)(up(a),s)
```

preserves ordering of sequences (which we consider as a correctness criterion).

```
push_down_ordered : LEMMA
  FORALL (x:Seq[A]) :
   ordered?(<=)(x) IMPLIES
   FORALL (a:A) : ordered?(<=)(push_down(<=)(a,x))
```

The proof of lemma `push_down_ordered` relies on the proof principle `struct_gi_greatest` for greatest invariants from Subsection 4.1. This proof principle supports (as the (bi)simulation proof principles do as well) local reasoning for global properties. Lemma `push_down_ordered` represents a typical goal for this proof method: we wish to check if a greatest invariant (`ordered?(<=)`) holds for images of the `coreduce` combinator (here `pd(<=)`) under the assumption that some other predicate holds on the domain of `coreduce`. In our case the predicate

```
P = {z:[Lift[A],Seq[A]] | ordered?(<=)(proj_2(z))}
```

on the domain `[Lift[A],Seq[A]]` of `pd(<=)` is obtained from the assumption in the lemma by substitution[24]. Now the proof proceeds in two steps

[24] We apply the substitution functor for the second projection.

- We check that `P` is an invariant for the structure map `pd_struct`. This basically involves a case distinction following the structure of `pd_struct`. We observe that one step of `pd_struct` preserves the extended predicate `P` and, thus, implicitly the ordering.
- If `P` holds on an element `x` of `[Lift[A],Seq[A]]` then `pd(<=)(x)` satisfies the local predicate `ord_local(<=)`. Here this proof step reduces to various simple rewrite steps.

In general, finding a suitable invariant `P` on the domain of the `coreduce` combinator is the heart of a greatest invariant proof.

Interestingly, the formal proof of this correctness criterion revealed an inaccuracy in earlier (CHARITY) versions. We had to strengthen `<=` from a preorder (as suggested in [Coc96]) to a dichotomous preorder[25]. All (standard) tests performed with finite observations in CHARITY had (accidently) involved dichotomous orders only.

6 Coalgebraic datatypes, more generally

So far we have described various (PVS-)theories of sequences based on the finality of the "next" destructor operation on sequences. This may seem *ad hoc* to those who are not familiar with coalgebraic datatypes. In order to put our approach in a wider context, we indicate in this section how it forms part of a more general theory of coalgebraic datatypes. We will sketch a possible syntax for such coalgebraic datatypes—based on the approach suggested first in [Hag87] and implemented in CHARITY [CF92, CS95], but put in a form adapted to PVS. Additionally, we sketch how to formulate associated proof principles with invariants and bisimulations for such general coalgebraic datatypes, following [HJ95, HJ98, HJ97]. What we will say will be formulated for logical languages, in contrast to type theoretic languages, where one may have coalgebraic datatypes as well (like in the COQ system [BBC+97, Gim95, Gim96]). We do so because these logical languages are more familiar and therefore more suitable for conveying the main ideas.

We illustrate the syntax for (algebraic) datatypes in PVS via the following example of finite lists (of elements of some parameter type `A`).

```
List[A : TYPE] : DATATYPE
  BEGIN
    nil : nil?
    cons(hd : A, tl : list) : cons?
  END List
```

In this notation, `nil` and `cons` are the constructors for building elements of the datatype `List[A]`, `hd` and `tl` are the accessors, and `nil?` and `cons?` are the recognisers. When the PVS system type-checks a file with this datatype definition, it internally generates theories (represented externally in the separate file `List_adt.pvs`) which contain various associated axioms (essentially describing these lists as initial algebras). For example, these theories comprise an induction principle and a "map" definition for lists. See [ORSvH95, ORR+96, RSC96] for more information.

Given this syntax for algebraic datatypes, one can envisage a syntax for coalgebraic datatypes in which we could write our type of sequences as:

```
Seq[A : TYPE] : CODATATYPE
  BEGIN
    next : Lift[[A, Seq]]
  END Seq
```

The proof assistant could then automatically generate associated definition- and proof-principles, involving `coreduce`, `map`, `record`, bisimulations and invariants, and thus exploit the finality of `Seq[A]`—according to the intended model of this type definition.

Following the CHARITY approach, the general form for such coalgebraic datatype definitions could be as follows.

[25] All elements are comparable.

```
NewType[A1, ..., An : TYPE] : CODATATYPE
  BEGIN
    destr1 : [P1[As] -> T1[As,NewType]]
    ...
    destrm : [Pm[As] -> Tm[As,NewType]]
  END
```

In this definition, the types `A1`, ..., `An` are type parameters. We have written `As` for the list $\langle$`A1`, ..., `An`$\rangle$ of these parameters. The `destr1`, ..., `destrm` are destructors, which tell us what we can observe about the type `NewType` that we are defining. In explicit form they have types:

```
destri : [NewType[As] -> [Pi[As] -> Ti[As, NewType[As]]]]
```

Or equivalently, using Currying:

```
destri : [NewType[As], Pi[As] -> Ti[As, NewType[As]]]
```

The `Pi` are types of inputs, in which the type `NewType` that we are defining is not allowed to occur. The `Ti` are the output types, in which `NewType` can occur. We may assume that the `Pi` and `Ti` are built up from constant types, using finite products (written as $A \times B$ or $[A, B]$), coproducts (or disjoint unions, written as $A + B$) and previously defined datatypes and codatatypes.

We briefly describe associated definition principles using `coreduce` and `record`, like in CHARITY. For an arbitrary type `X` we have

```
coreduce : [[X, P1[As] -> T1[As, X]],..., [X, Pm[As] -> Tm[As, X]]
               -> [X -> NewType[As]]]
```

satisfying the equations (for $1 \leq i \leq m$):

```
destri(coreduce(fs)(x), ai) = map_Ti(ids, coreduce(fs))(fi(x, ai))
```

where the `fi` are functions `[X, Pi[As] -> Ti[As, X]]`, and `ai` is an element of the type `Pi[As]`.

The `record` combinator is easier: it has type

```
record : [[P1[As] -> T1[As, NewType[As]]],..., [Pm[As] -> Tm[As,
NewType[As]]]
               -> NewType[As]]
```

and satisfies the equations

```
destri(record(fs), a) = fi(a)
```

This `record` combinator gives an inverse to the m-tuple $\langle$`destr1`, ..., `destrm`$\rangle$ of destructors, like `next_inv` is inverse to `next` towards the end of Section 3. Note, that the `record` combinator can be used to derive constructors for the datatype. However, these constructors may become clumsy once exponentials appear in the codomain of destructors as in `[A -> Lift[[NewType[A,B],B]]]` where `record` remains the only "constructor".

What we have described so far is essentially as in the (recent higher order version of the) CHARITY language[26]. We continue to sketch how one can formulate proof principles for bisimulations and invariants with respect to general codatatype definitions as for `NewType` above, following [HJ95, HJ98, HJ97]. This requires "liftings" of substitution in types from types to predicates and to relations. For a type `T[A]` containing a parameter `A`, one can define by induction on the structure of `T` the liftings:

```
pred_lift_T : [PRED[A] -> PRED[T[A]]]
rel_lift_T : [REL[A] -> REL[T[A]]]
```

[26] In fact, our approach has to handle variance problems as well: `NewType` is required to occur only positively inside the output type `Ti`, and (possible) positive, negative or mixed occurrences of type parameters `Ai` lead to a slightly more complicated typing of map. These issues (as well as strength which we dropped for simplicity) will be dealt with in a future more detailed proposal following the ideas manifested in CHARITY.

where `REL[A]` is the type `PRED[[A,A]]` of binary relations on `A`. The precise form of this lifting is not so important at this stage, and may be found in [HJ95, HJ98, HJ97].

Returning to our general codatatype `NewType`, we can now define that an arbitrary predicate `Q : PRED[NewType[As]]` is an *invariant* (for this codatatype `NewType`) if it satisfies (for all $1 \leq i \leq m$):

```
FORALL(x : NewType[As]) : Q(x) IMPLIES
  FORALL(a : Pi[As]) : pred_lift_Ti(As, Q)(destri(x, a))
```

Similarly, a binary relation `R : REL[NewType[As]]` is a *bisimulation* if (for all $1 \leq i \leq m$):

```
FORALL(x, y : NewType[As]) : R(x,y) IMPLIES
  FORALL(a : Pi[As]) : rel_lift_Ti(As, R)(destri(x, a), destri(y, a))
```

The proof principles for bisimulations can now be formulated as:

```
FORALL(R : REL[NewType[As]]) : bisimulation?(R) IMPLIES
  FORALL(x, y : NewType[As]) : R(x,y) IMPLIES x = y
```

Invariants play an important rôle in refinements (or implementations), see *e.g.* [LV95, LG86], (and [Jac97] for a coalgebraic account). One obtains a proof principle for invariants as soon as one has a notion of greatest invariant `gi(Q)` contained in an arbitrary predicate `Q` (see [Jac97] for a definition of `gi(Q)` as countable meet, with the earlier definition for sequences as a special case).

Hopefully, this sketch will give the reader an idea of a general approach to coalgebraic datatypes, of which our formalisation of sequences is a special case. What we have described has not been fully implemented in any of the currently available proof assistants. Several proof assistants offer features which would allow for an implementation of such datatypes. Paulsons (co)inductive definitions and datatype package in ISABELLE/ZF [Pau97b] encodes datatypes as least and greatest fixed points of monotone operators. However these datatypes are merely recursive sets and are not automatically equipped with the combinators. Nevertheless, these could be added by coinductive definitions. Following the approach of [LP94] (weak) final coalgebras could be implemented by existential types. The easiest "implementation" in any logic is, of course, the assertion of an extra axiom stating the finality. However, depending on the logical framework such an axiom could may lead to inconsistencies.

7 Concluding remarks

We briefly discuss some alternative approaches to the formalisation of sequences and infinite objects in general in relation to our coalgebraic work.

Coquand [Coq94] proposes to encode possibly infinite objects or expressions by systems of guarded recursive equations. These are defining equations in the flavour of inductive definitions with the extra requirement that recursive occurences of functions are guarded by some constructors of the lazy (or final) datatype in question. In fact, this restriction can be reformulated in terms of destruction: an expression is guarded if it results from applications of destructors of the datatype. The coreduce combinator in our approach defines unique solutions (f, for given h) to flat systems of equations[27]

$$\mathsf{next}(f(x_1, \ldots, x_n)) = \mathsf{lift}(id \times f)(h(x_1, \ldots, x_n))$$

however, our presentation in terms of defining machines or coalgebras does not impose the extra restriction to guarded terms. Coquand is able to express general systems of equations where the lookahead by destruction is possibly deeper than one. These can be unfolded to flat systems along the lines of [BM96]. His proof principle appears to be a blend of our coinductive principles for coinductive definition given by coreduce and greatest invariants. In fact, it states in our terms that a proposition can be destructed and occur recursively in the proof, thus, expressing an invariant

[27] For multiple equations one can use a coproduct of state spaces.

property. Hence, his inductive proof principle is truely coinductive in spirit (besides it is not well–founded).

Leclerc and Paulin-Mohring [LP94, Pau96] suggest an impredicative encoding of streams (only infinite sequences) using existential types in COQ. Their encoding involves the definition of a coreduce equivalent called "built", and is based on [Wra89]. They develop stream specific proof principles using indices or positions and also the notion of invariant. However, their encoding lacks coinductive proof principles because existential types yield weakly final coalgebras and therefore only the existence part of the definition of coreduce. Uniqueness of such encodings can be established under additional parametricity assumptions, see *e.g.* [Has91, PA93].

Closest to our approach is Paulsons account on (co)datatypes in ISABELLE/HOL [Pau97a]. He focuses on the definitional encoding of codatatypes in higher order logics and derives basic examples for coinductive definitions and proofs. We take off at exactly this point and demonstrate that regardless of the implementation of the final coalgebra powerful tools can be derived. Our theory development abstracts from the particular implementation because we desired a theory which is, to a certain degree, independent from the tool or logic that is being used.

Various formalisations of sequences in proof assistants —not including the current coalgebraic approach—are discussed in [DGM97]. One can distinguish three "direct" formalisations of sequences (of elements of some type A) in terms of suitable existing types. (1) Infinite sequences of elements of $\mathsf{Lift}(A)$, *i.e.* of elements of A augmented with an extra element for undefined. This approach occurs in [NS95]. (2) Disjoint union of finite and infinite sequences, see [CP96, Age94]. (3) Functions from downclosed subsets of the natural numbers to A, in [DG97]. One way or another, all three approaches lead to unpleasantly or even unmanageably complicated details when proving elementary properties about operations like filtering and flattening, see [DGM97].

When it comes to manageability, the domain theoretic approach of [MN97] (based on the formalization of domain theory in [Reg95]) often leads to easier proofs (than in the present coalgebraic approach), since induction can be used for admissible predicates (see the end of Subsection 4.2). Typically in domain theory initial algebras and final coalgebras coincide as a solution of a domain equation (see *e.g.* [AJ94, Fio96]). Thus, their central induction proof principle is restricted to such frameworks. Of course, the domain theoretic approach is perfect if the problem (involving sequences) fits into the context of the Logic of Computable Functions. However, the domain theoretic setting requires that all functions have to be continuous, and all predicates (subject to inductive proofs) have to be admissible. This restricts the application domain, and generates extra proof obligations. There is an obvious inclusion from the ordinary set/type theoretic world into the domain theoretic world (via lifting with sets/types as flat domains), but once one has entered the world of domains, there is no way out: all subsequent work that uses sequences has to be done with domains.

Our formalisation in the general higher order logic of PVS also allows the verification of properties of expressions which have (only) descriptive definitions. This is adequate if specifications are not required to have computational meaning. Characteristic for our formalisation is that filtering does *not* commute with composition, in the sense that the equation

$$\mathsf{filter}(p, \mathsf{comp}(x, y)) = \mathsf{comp}(\mathsf{filter}(p, x), \mathsf{filter}(p, y)) \qquad (*)$$

does not hold for *all* sequences $x, y{:}\,\mathsf{Seq}(A)$. For example, if $A = \{0, 1\}$ and the predicate $p{:}\,A \to \mathsf{bool}$ is only true on 1, then by taking x to be the infinite sequence of 0s, and y to be the infinite sequence of 1s, we obtain

$$\begin{array}{rcll} \mathsf{filter}(p, \mathsf{comp}(x, y)) & = & \mathsf{filter}(p, x) & \text{because } x \text{ is infinite} \\ & = & \mathsf{empty\_seq}. & \\ \mathsf{comp}(\mathsf{filter}(p, x), \mathsf{filter}(p, y)) & = & \mathsf{comp}(\mathsf{empty\_seq}, y) & \\ & = & y. & \end{array}$$

In the domain theoretic formalisation of sequences these outcomes are the same [DGM97], because the (continous) filter function returns "divergence" on x which is preserved by composition. This contradicts our intuition about sequences as formalised in this paper. However, the equation $(*)$

makes perfect sense in a domain theoretic context, where sequences may terminate normally or abnormally. In a coalgebraic setting this behaviour $(*)$ can also be realised by using an alternative signature (or functor) for sequences: instead of $X \mapsto \mathsf{lift}(A \times X)$ as used above, one can take $X \mapsto \mathsf{lift}(\mathsf{lift}(A \times X))$. In the latter case, two bottom elements are adjoined, one for normal and one for abnormal termination. The associated final coalgebra then takes the form:

$$\mathsf{LCFSeq} \xrightarrow{\quad\mathsf{next}\quad} \mathsf{lift}(\mathsf{lift}(A \times \mathsf{LCFSeq}(A))$$

An outcome $\mathsf{next}(\sigma)$ either yields $\bot = \mathsf{bot}$ for abnormal termination (divergence), $\downarrow = \mathsf{up}(\mathsf{bot})$ for normal termination, or (a, σ') where a is an element of A, and σ' is a (tail) sequence. Composition for such alternative sequences can be defined via the machine structure

$$\mathsf{comp\_struct}(\sigma,\tau) = \begin{cases} \bot & \text{if } \mathsf{next}(\sigma) = \bot \text{ or } \mathsf{next}(\tau) = \bot \\ \downarrow & \text{if } \mathsf{next}(\sigma) = \downarrow \text{ and } \mathsf{next}(\tau) = \downarrow \\ (a, (\sigma, \tau')) & \text{if } \mathsf{next}(\sigma) = \downarrow \text{ and } \mathsf{next}(\tau) = (a, \tau') \\ (a, (\sigma', \tau)) & \text{if } \mathsf{next}(\sigma) = (a, \sigma'). \end{cases}$$

and an appropriate filter would produce divergence on unfair (*w.r.t.* p) sequences. For such a sequence formalisation commutation of filter and composition can be established, in accordance with our intuition.

Thus, coalgebraic datatypes in higher order logic may provide sequence formalisation with either computational or descriptive semantics. A key advantage of the coalgebraic approach is that refined observations manifest themselves in a refined type of the destructor codomain signature.

A disadvantage of the coalgebraic approach is that it is relatively unknown, and, as yet, not fully supported by existing proof assistants. We hope that the present paper contributes to the familiarity, acceptance and further development of coalgebraic notions and techniques.

Acknowledgements

We thank Marco Devillers and David Griffioen for stimulating discussions, and for pushing us to substantiate our claims about the suitability of the coalgebraic approach. Larry Paulson and the referees gave valuable comments on the paper.

References

[Age94] S. Agerholm. *A HOL Basis for Reasoning about Functional Programs.* PhD thesis, Univ. of Aarhus, Denmark, 1994.

[AJ94] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, Dov M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford Univ. Press, 1994.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-Chr. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant User's Guide Version 6.1. Technical Report 203, INRIA Rocquencourt, France, May 1997.

[BM96] J. Barwise and L. Moss *Vicious Circles.* Number 60 in CSLI Lecture Notes. CSLI Publications, Stanford University, 1996.

[CF92] J.R.B. Cockett and T. Fukushima. About charity. Technical Report 92/480/18, Dep. Comp. Sci., Univ. Calgary, 1992.

[Coc96] J.R.B. Cockett. Charitable thoughts, 1996. (draft lecture notes, available under `http://www.cpsc.ucalgary.ca/projects/charity/home.html`).

[Coq94] Thierry Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Selected Papers 1st Intl. Workshop on Types for Proofs and Programs, TYPES'93, Nijmegen, The Netherlands, 24–28 May 1993*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, Berlin, 1994.

[CP96] C.T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lect. Notes Comp. Sci., pages 241–257. Springer, Berlin, 1996.

[CS95] J.R.B. Cockett and D. Spencer. Strong categorical datatypes II: A term logic for categorical programming. *Theor. Comp. Sci.*, 139:69–113, 1995.

[DG97] M. Devillers and D. Griffioen. A formalization of finite and infinite sequences in PVS. Techn. Rep. CSI-R9702, Comput. Sci. Inst., Univ. of Nijmegen, 1997.

[DGM97] M. Devillers, D. Griffioen, and O. Müller. Possibly infinite sequences in theorem provers: A comparative study. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104, Murray Hill, NJ, August 1997. Springer-Verlag.

[DP96] B.A. Davey and H.A. Priestley *Introduction to Lattices and Order*. Cambridge Univ. Press, 1990.

[Fio96] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Cambridge Univ. Press, 1996.

[Gim95] E. Giménez. Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, number 996 in Lect. Notes Comp. Sci., pages 39–59. Springer, Berlin, 1995.

[Gim96] E. Giménez. Implementation of co-inductive types in Coq: an experiment with the Alternating Bit Protocol. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, number 1158 in Lect. Notes Comp. Sci., pages 135–152. Springer, Berlin, 1996.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge Univ. Press, 1993.

[Hag87] T. Hagino. A typed lambda calculus with categorical type constructors. In D.H. Pitt, A Poigné, and D.E. Rydeheard, editors, *Category and Computer Science*, number 283 in Lect. Notes Comp. Sci., pages 140–157. Springer, Berlin, 1987.

[Has91] R. Hasegawa. Categorical data types in parametric polymorphism *Math. Struct. Comp. Sci.*, 4:71–109, 1991.

[HJ95] C. Hermida and B. Jacobs. An algebraic view of structural induction. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic 1994*, number 933 in Lect. Notes Comp. Sci., pages 412–426. Springer, Berlin, 1995.

[HJ98] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. & Comp.*, to appear, 1998.

[HJ97] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. In Eugenio Moggi and Giuseppe Rosolini, editors, *Category Theory and Computer Science*, volume 1290 of *Lecture Notes in Computer Science*, pages 220–241, Santa Margherita Ligure, Italy, September 1997. Springer-Verlag.

[HJ97www] U. Hensel and B. Jacobs. A coalgebraic theory of sequences in PVS. available under `http://www.cs.kun.nl/~bart/sequences.html` , 1997. PVS Theories and Proofs.

[Jac97] B. Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 276–291, Sydney, Australia, December 1997. Springer-Verlag.

[Jay96] C.B. Jay. Data categories. In M.E. Houle and P. Eades, editors, *Computing: The Australasian Theory Symposium Proceedings, Melbourne, Australia, 29–30 January, 1996*, volume 18, pages 21–28. Australian Computer Science Communications, 1996. ISSN 0157–3055.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, 62:222–259, 1996.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, MA, 1986.

[LP94] Francois Leclerc and Christine Paulin-Mohring. Programming with streams in Coq: A case study: The sieve of Eratosthenes. In H. Barendregt and T. Nipkow, editors, *Selected Papers 1st Intl. Workshop on Types for Proofs and Programs, TYPES'93, Nijmegen, The Netherlands, 24–28 May 1993*, volume 806 of *Lecture Notes in Computer Science*, pages 191–212. Springer-Verlag, Berlin, 1994.

[LP92] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Techn. rep. ecs-lfcs-92-211, LFCS, Edinburgh, 1992.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations. I. Untimed systems. *Inf. & Comp.*, 121(2):214–233, 1995.

[MJ96] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit: A case study exploring the boundary between formal reasoning systems. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Bastad, Sweden, September 1996. Springer-Verlag Electronic Workshops in Computing.

[MN97] O. Müller and T. Nipkow. Traces of I/O-automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, number 1214 in Lect. Notes Comp. Sci., pages 580–594. Springer, Berlin, 1997.

[NS95] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, number 996 in Lect. Notes Comp. Sci., pages 101–119. Springer, Berlin, 1995.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996.

[ORS93] S. Owre, J.M. Rushby, and N. Shankar. *The* PVS *Specification Language*, 1993. available from `ftp.csl.sri.com/pub/pvs`.

[ORSvH95] S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.

[Pau96] C. Paulin-Mohring. Circuits as streams in COQ: Verification of a sequential multiplier. In S. Berardi and M. Coppo, editors, *Workshop on Types for Proofs and Programs*, number 1158 in Lect. Notes Comp. Sci., pages 216–230. Springer, Berlin, 1996.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lect. Notes Comp. Sci. Springer, Berlin, 1994.

[Pau97a] L.C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journ. of Logic and Computation*, 7:175–204, 1997.

[Pau97b] L. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Essays in Honour of Robin Milner*. Press, 1997. in press.

[PA93] G. Plotkin and M. Abadi. A Logic for Parametric Polymorphism In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lect. Notes Comp. Sci., pages 361–375. Springer, Berlin, 1993.

[Reg95] F. Regensburger. HOLCF: Higher order logic of computable functions. In E.Th. Schubert, Ph.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, number 971 in Lect. Notes Comp. Sci., pages 293–307. Springer, Berlin, 1995.

[Rei96] Horst Reichel. Unifying ADT – and evolving algebra specifications. *EATCS Bulletin*, 59, 1996.

[RSC96] J.M. Rushby and D. W. J. Stringer-Calvert. A Less elementary tutorial for the PVS specification and verification system. Technical Report CSL-95-10, CSL, SRI International, 1996.

[Rut96] J.J.M.M. Rutten. Universal coalgebra: a theory of systems. CWI Report CS-R9652, 1996.

[Wra89] G.C. Wraith. A note on categorical datatypes. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Science*, number 389, pages 118–127. Springer, Berlin, 1989.