Possibly Infinite Sequences in Theorem Provers: A Comparative Study

Marco Devillers[1], David Griffioen*[1,2] and Olaf Müller†[3]

[1] Computing Science Institute, University of Nijmegen, The Netherlands.
{marcod|davidg}@cs.kun.nl
[2] CWI, Amsterdam, The Netherlands, griffioe@cwi.nl
[3] Computer Science Department, Technical University Munich, Germany.
mueller@informatik.tu-muenchen.de

Abstract. We compare four different formalizations of possibly infinite sequences in theorem provers based on higher-order logic. The formalizations have been carried out in different proof tools, namely in Gordon's HOL, in Isabelle and in PVS. The comparison considers different logics and proof infrastructures, but emphasizes on the proof principles that are available for each approach. The different formalizations discussed have been used not only to mechanize proofs of different properties of possibly infinite sequences, but also for the verification of some non-trivial theorems of concurrency theory.

1 Introduction

Sequences occur frequently in all areas of computer science and mathematics. In particular, formal models of distributed systems often employ (possibly infinite) sequences to describe system behavior over time, *e.g.* TLA [Lam94] or I/O automata [LT89]. Recently, there is a growing interest in using theorem provers not only to verify properties of systems described in such a model, but also to formalize (parts of) the model itself in a theorem prover. For this reason, formalizations of possibly infinite sequences in proof tools are needed.

In this paper, we compare a number of such formalizations, which were carried out in theorem provers based on higher-order logic. We compare to what extent the formalizations have been worked out, and draw conclusions on general applicability. In the comparison we consider the following representative requirements on the datatype of possibly infinite sequences: A predicate *finite* characterizes finite sequences, operations on sequences include *hd, tl, map, length, concat* (also known as *append*), *filter* (removal of elements) and *flatten* (concatenation of possibly infinitely many finite sequences).

In particular, *filter* and *flatten* are chosen, because defining them and reasoning about them turned out to be rather complicated in various formalizations,

* Research supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-316-125
† Research supported by BMBF, *KorSys*

especially because of their result depend on infinite calculations. These functions are especially motivated by concurrency theory: for abstraction and modularity purposes, internal messages are often hidden in behaviors using the *filter* function. The *flatten* function is required for proofs about system refinement, where infinitely many steps of a system may be simulated by finite behaviors.
The following four approaches are evaluated and compared:

- HOL-Fun: Sequences are defined as functions by $(\alpha)\mathsf{seq} = \mathbb{N} \to (\alpha)\mathsf{option}$, where the datatype $(\alpha)\mathsf{option} \equiv \mathsf{None} \mid \mathsf{Some}(\alpha)$ is used to incorporate finite sequences into the model: None denotes a "non-existing" element. This approach has been taken by Nipkow, Slind and Müller [NS95, MN97], where it has been used to formalize parts of I/O automata meta-theory. It has been carried out in Isabelle/HOL [Pau94].
- HOL-Sum: Sequences are defined as the disjoint sum of finite and infinite sequences: $(\alpha)\mathsf{seq} \equiv \mathsf{FinSeq}((\alpha)\mathsf{list}) \mid \mathsf{InfSeq}(\mathbb{N} \to \alpha)$. Here $(\alpha)\mathsf{list}$ stands for ordinary finite lists. This approach has been taken by Chou and Peled [CP96] in the verification of a partial-order reduction technique for model checking and by Agerholm [Age94] as an example of his formalization of domain theory. Both versions have been carried out independently from each other in Gordon's HOL [GM93].
- PVS-Fun: Sequences are defined as functions from a downward closed subset of $\mathbb{N}$, where the cardinality of the subset corresponds to the length of the sequence. This is achieved by the dependent product $(S \in \mathbb{I} \times (S \to \alpha))$, where $\mathbb{I} \subseteq \wp(\mathbb{N})$ denotes the set of all downward closed subsets of $\mathbb{N}$. This approach has been taken by Devillers and Griffioen [DG97], who also formalized I/O automata meta-theory. It has been carried out in PVS [ORSH95].
- HOL-LCF: In domain theory, sequences can be defined by the simple recursive domain equation $(\alpha)\mathsf{seq} \equiv \mathsf{nil} \mid (\alpha) : (\alpha)\mathsf{seq}$, where the "cons"-operator : is strict in the first and lazy in its second argument. This approach has been taken by Müller and Nipkow [MN97] as a continuation of the first approach, as that one caused some difficulties that will be sketched later on. It has been carried out in Isabelle/HOLCF [Reg95].

The aim of every formalization is a rich enough collection of theorems, such that independence on the specific model is reached. As this, up to our experience, will not completely be possible, we focus our comparison on the proof principles that are offered by the respective approaches. Their usability, applicability and degree of automation are especially essential for the user of the sequence package and influence proof length considerably. In addition, specific features of the tools and of the respective logics are taken into account.

2 Theorem Provers and Logics

In this section we summarize the distinguishing aspects of the different tools used, as far as they are relevant to the sequence formalizations.

2.1 The different Logics

Isabelle/HOL and Gordon's HOL. Gordon's HOL [GM93] is a theorem prover for higher-order logic developed according to the LCF approach [Pau87]. Isabelle [Pau94] is a *generic* theorem prover that supports a number of logics, among them first-order logic (FOL), Zermelo-Fränkel set theory (ZF), constructive type theory (CTT), higher-order logic (HOL), and others. As Isabelle/HOL and Gordon's HOL are similar, we will in general not distinguish between them and refer to both of them as HOL. Both logics are based on Church's formulation of simple type theory [Chu40], which has been augmented by a ML-style polymorphism and extension mechanisms for defining new constants and types. The following section gives a quick overview, mainly of the notation we use.

Types. The syntax of types is given by $\sigma \quad ::= \quad v \,|\, (\sigma_1, ..., \sigma_n)op$ where $\sigma, \sigma_1, ..., \sigma_n$ range over types, v ranges over type *variables*, and op ranges over n-ary type *operators* ($n \geq 0$). Greek letters (e.g. α, β) are generally used for type variables, and sans serif identifiers (e.g. list, option) are used for type operators. In this paper, we use the type constants $\mathbb{N}$ and $\mathbb{B}$, denoting natural numbers and booleans, and the type operators $\to$ for the function space and $\times$ for the cartesian product.

Terms. The syntax of terms is given by $M \quad ::= \quad c \,|\, v \,|\, (MN) \,|\, \lambda v.M$ where c ranges over constants, v ranges over variables, and M and N range over terms. Sans serif identifiers (e.g. a,b,c) and non-alphabetical symbols (e.g. $\Rightarrow$, $=$, $\forall$) are generally used for constants, and italic identifiers (e.g. x,y,z) are used for variables. Every term in HOL denotes a *total* function and has to be *well-typed*. HOL incorporates Hilbert's choice operator ε as a primitive constant.

HOLCF. HOLCF [Reg95] conservatively extends Isabelle/HOL with concepts of domain theory such as complete partial orders, continuous functions and a fixed point operator. As a consequence, the logic of the original LCF tool [Pau87] constitutes a proper sublanguage of HOLCF.

HOLCF uses Isabelle's *type classes*, similar to Haskell, to distinguish between HOL and LCF types. A type class is a constraint on a polymorphic variable restricting it to the class of types fulfilling certain requirements.

For example, there is a type class $\alpha :: \mathsf{po}$ (partial order) that restricts the class of all types α of the universal type class term of HOL to those for which the constant $\sqsubseteq\colon \alpha \times \alpha \to \mathbb{B}$ is reflexive, transitive and antisymmetric. Showing that a particular type is an *instance* of this type class, requires to prove the properties above for this particular definition of the symbol $\sqsubseteq$. Once this proof has been done, Isabelle can use this semantic information during static type checking.

The default type class of HOLCF is pcpo (pointed complete partial order), which is a subclass of po, equipped with a least element $\bot$ and demanding completeness for $\sqsubseteq$. There is a special type for continuous functions between pcpos. Elements of this type are called *operations*, the type constructor is denoted by $\to_c$, in contrast to the standard HOL constructor $\to$. Abstraction and application of continuous functions is denoted by Λ (instead of λ) and $f\text{`}t$ (instead of

$f\ t$). The fixed point operator $\mathit{fix} : (\alpha :: \mathsf{pcpo} \to_c \alpha) \to_c \alpha$ enjoys the fixed point property $\mathit{fix}f = f(\mathit{fix}f)$. Note that the requirement of continuity is incorporated in the type of *fix* ($\to_c$ instead of $\to$). This illuminates the fact, that checking continuity in HOLCF is only a matter of automatic type checking, as far as terms belong to the proper LCF sublanguage (Λ abstractions and ‘ applications). HOLCF includes a datatype package that allows the definition of domains by recursive equations.

PVS Logic. Similar to HOL, the PVS logic [ORSH95] is based on higher order logic, but type expressions are more expressive, featuring set theoretic semantics. Whereas HOL only allows simple types, PVS offers mechanisms for *subtyping* and *dependent types*. Again, we only give a quick overview, mainly clarifying syntax.

Subtyping is expressed with the usual set notation, e.g., $\{n \in \mathbb{N}.\, even(n)\}$ is the set of all even natural numbers. The dependent sum $(x : A \times B_x)$ – in which the second component B_x depends on a member x of the first set A – denotes the set of all pairs (a, b) where $a \in A$ and $b \in B_a$. For example, if S^i denotes a sequence of length i then members of $(i : \mathbb{N} \times \{a, b, c\}^i)$ would be $(2, ab)$ and $(3, bac)$. A dependent product $(x : A \to B_x)$ denotes all functions f where if $a \in A$ then $f(a) \in B_a$. For example, if f is a member of the dependent product $(i : \mathbb{N} \to \{a, b, c\}^i)$, then $f(2) = ab$ and $f(3) = acb$ would be type-correct. Furthermore, we use π_0 and π_1 for the left- and right-hand projection in a tuple, e.g., $\pi_0((a, b)) = a$.

Whereas the general type checking problem in HOL is decidable, in PVS it is not. The PVS system solves this problem by generating type correctness conditions (TCCs) for those checks it cannot resolve automatically.

Similar to HOL, the specification language of PVS is organized into theories and datatypes, which, in contrast to HOL, can be parameterized by types and constants. This enables an easy handling of generic theories. HOL's type variables and Isabelle's type classes offer a similar mechanism.

2.2 Design Philosophies and Tool Specifics

Both Gordon's HOL and Isabelle/HOL, were developed according to the LCF-system approach [Pau87], which ensures soundness of extensions to the logic. The main idea of the LCF approach is to use abstract data types to derive proofs. Predefined values of a data type corresponded to instances of axioms, and the operations correspond to inference rules. By using a strictly typed language, wherefore ML was developed, theorem security is assured.

PVS, however, is a closed tool. There is no document that describes the exact syntax and semantics of the PVS logic, which is hardwired in the tool. On the other hand, PVS features a tight integration of rewriting and various decision procedures (*e.g.* for arithmetic and propositional logic based on BDDs), which results in a high degree of automation. This is in particular an advantage in comparison to Isabelle/HOL, which in the present version does not offer effective support for arithmetic.

3 HOL-Fun: Functions in Isabelle/HOL

Definition 1 (Type of Sequences). Sequences are defined by the type

$$(\alpha)\mathsf{seq} = \mathbb{N} \to (\alpha)\mathsf{option}$$

using the option datatype defined as: $(\alpha)\mathsf{option} = \mathsf{None} \mid \mathsf{Some}(\alpha)$. None denotes "nonexisting" elements and is used to model finite sequences. To avoid the case in which None appears within a sequence – otherwise the representation would not be unique – the predicate

$$\mathit{is\_sequence}(s) = (\forall i.s(i) = \mathsf{None} \Rightarrow s(i+1) = \mathsf{None})$$

is introduced, which has to hold for every sequence. Sequences therefore can be regarded as a quotient structure, where *is_sequence* characterizes the normal form of each equivalence class. Of course, every operation has to yield a term in normal form. This is the main disadvantage of this approach, as it is not straightforward to construct the normal form for *e.g.* the *filter* function, which will be discussed below.

Definition 2 (Basic Operations). Functions on sequences are defined pointwise. This is especially simple if the output length is equal to the input length (as for *map*) or if it can easily be computed from it (as for $\oplus$).

$$\begin{array}{llll} \mathsf{nil} & = \lambda i.\mathsf{None} & \mathit{hd}(s) & = s(0) \\ \mathit{tl}(s) & = \lambda i.s(i+1) & \mathit{len}(s) & = \sharp\{i\,.\,s(i) \neq \mathsf{None}\} \\ \mathit{map}\,f\,s & = f \circ s & s \oplus t & = \lambda i.\ \mathsf{if}\ i < \mathit{len}(s)\ \mathsf{then}\ s(i)\ \mathsf{else}\ t(i - \mathit{len}(s)) \end{array}$$

where the codomain for *len* and $\sharp$ (cardinality) are the natural numbers, extended by an infinity element: $\mathbb{N}^\infty = \mathsf{Fin}(\mathbb{N})|\mathsf{Inf}$. Arithmetic operations and relations (as *e.g.* $-, <$ have been extended accordingly.

Definition 3 (Filter). Filtering is divided into two steps: first, $\mathit{proj} : (\alpha)\mathsf{seq} \to (\alpha)\mathsf{seq}$ replaces every element not satisfying P by None, then the resulting sequence is brought into normal form. Normalization is achieved by an index transformation $\mathit{it} : \mathbb{N} \to \mathbb{N}$, that has to meet three requirements: first, normalization has to maintain the ordering of the elements, second, every $\mathsf{Some}(a)$ has to appear in the normal form, and third, if there is a None in the normal form, then there will be no Some afterwards. These requirements can directly serve as the definition for *it* using Hilbert's description operator ε.

$$\begin{array}{lll} \mathit{proj}\,P\,s & = \lambda i.\,\mathsf{case}\ s(i)\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{None} & \\ & \qquad\qquad \mid \mathsf{Some}(a) \Rightarrow\ \mathsf{if}\ P(a)\ \mathsf{then}\ \mathsf{Some}(a)\ \mathsf{else}\ \mathsf{None} & \\ \mathit{it}(s) & = \varepsilon\,\mathit{it}\,.\ \mathit{monotone}(\mathit{it}) \wedge & \\ & \qquad \forall i\,.\,s(i) \neq \mathsf{None} \Rightarrow i \in \mathit{range}(\mathit{it}) \wedge & \\ & \qquad \mathit{is\_sequence}(s \circ \mathit{it}) & \\ \mathit{NF}(s) & = s \circ \mathit{it}(s) & \\ \mathit{filter}\,P\,s & = \mathit{NF} \circ (\mathit{proj}\,P\,s) & \end{array}$$

The definition for *it* is a nice requirement specification, but it is not simple to work with it, as for every $\varepsilon x.P(x)$ the existence of an x satisfying P has to be shown. Theoretically, this can be done using proof by contradiction, as we are in a classical logic, but it was not obvious how to do this in this case. In practice, an explicit construction seemed to be unavoidable.

One reason why Müller and Nipkow stopped this sequence formalization at this point [MN97] and changed to a formalization in HOLCF was the complexity of this construction. A second reason was the unsufficient support for arithmetic, provided by Isabelle/HOL up to now, as reasoning about normal forms heavily involves index calculations. However, a version without normal forms has been successfully used to model parts of the meta-theory of I/O-automata [NS95].

Anyway, it will turn out, that the PVS approach is very close to the one presented here, so that an impression of the practicability can be gained from the experiences that have been made there. In particular, *it* reappears in the PVS approach in a very similar fashion, and an explicit construction of it will be presented in that context.

4 HOL-Sum: Lists and Functions in Gordon's HOL

Chou and Peled [CP96] use a disjoint union type of a list for finite sequences, and a function from the natural numbers for infinite sequences.

Definition 4 (Type of Sequences).

$$(\alpha)\mathsf{seq} = \mathsf{FinSeq}((\alpha)\mathsf{list}) \mid \mathsf{InfSeq}(\mathbb{N} \to \alpha)$$

An advantage of this approach is that no normalization of elements in this type is needed. A disadvantage is that a number of the operators on sequences are implemented twice, once in case the argument is a finite sequence, and once in the infinite case.

Definition 5 (Basic Operations). For instance, consider the length *len* and *tl* functions shown below.

$$\begin{aligned} &\mathit{len}(\mathsf{FinSeq}\ l) = \mathsf{Fin}(\mathit{len}\ l) \\ &\mathit{len}(\mathsf{InfSeq}\ f) = \mathsf{Inf} \\ \\ &\mathit{tl}(\mathsf{FinSeq}\ l) \quad = \mathit{TL}\ l \\ &\mathit{tl}(\mathsf{InfSeq}\ f) \quad = \mathsf{InfSeq}(\lambda i\ .\ f(i+1)) \end{aligned}$$

In the above definitions, the length function returns an element in $\mathbb{N}^\infty$. The *tl* function is defined twice, for finite sequences the usual *TL* operator on lists is used, and for infinite sequences it uses a transposition function.

Whenever it is not easy to define a sequence in such a way, Chou and Peled make use of under-specified functions from the natural number to the data set. Such functions are not specified for all arguments greater than the length of a sequence. A conversion function *seq*, which takes a number $n : \mathbb{N}^\infty$ and such a function f as arguments, constructs the corresponding sequence to f of length n. In the definition below, $\mathit{genlist}\, f\, n$ is the finite list of the first n values $f(1), \ldots, f(n)$.

$$\begin{aligned} seq(\mathsf{Fin}\ n)(f) &= \mathsf{FinSeq}(genlist\ f\ n) \\ seq(\mathsf{Inf})(f) &= \mathsf{InfSeq}(f) \end{aligned}$$

For instance, the concatenation function, which takes two sequence arguments, is defined by means of this function. If this function were defined using normal case distinctions on the arguments, one would need four cases.

$$s \oplus t = seq\ (len(s) + len(t))\ (\lambda\, i\ .\ \mathsf{if}\ i < len(s)\ \mathsf{then}\ nth\ s\ i\ \mathsf{else}\ nth\ t\ (i - len(s)))$$

Definition 6 (Filter). Chou and Peled define the *filter* function as the limit of an ascending chain of finite sequences according to the prefix ordering $\sqsubseteq$ on sequences. Below the definitions of chains and limits are given. The argument of both functions is a variable c of type $\mathbb{N} \to (\alpha)\mathsf{list}$.

$$\begin{aligned} chain(c) &= (\forall j\ .\ (c\ j) \sqsubseteq (c\ (j+1))) \\ limit(c) &= seq\ (lub\ (\lambda n\ .\ \exists j\ .\ n = len(c\ j)))\ (\lambda i\ .\ nth\ (c\ (least(\lambda j\ .\ i < len(c\ j))))\ i) \end{aligned}$$

The chain function is a predicate which states that c is a chain iff all the elements satisfy the prefix ordering. The limit function returns the sequence *seq* where the length is the least upper bound *lub* of all lengths in the chain, and the i-th element in a sequence (if any) is the i-element of the first sequence in the chain which holds at least i elements.

The filter function then is defined as the limit of all projections on initial segments of a given argument.

$$\begin{aligned} FilterChain(p)(s)(j) &= \mathsf{FinSeq}(FILTER(p)(list(take\ s\ j))) \\ filter(p)(s) &= limit(FilterChain(p)(s)) \end{aligned}$$

The function *FilterChain* produces a chain of lists where the j-th element in such a list is the projection of p on the first j elements of s. For instance, when filtering all even numbers out of the sequence $(1, 4, 9, 16, 25, \ldots)$ the resulting chain will be $\mathsf{nil} \sqsubseteq (4) \sqsubseteq (4) \sqsubseteq (4, 16) \sqsubseteq \ldots$. The limit of this chain is, of course, the infinite sequence of squares of even numbers.

Properties proven about these limits include that every sequence is the limit of the chain of all of its finite prefixes, and that concatenation is continuous in its right argument, in the sense of Scott's topology. Theorems proven about the *filter* function include that *filter* distributes over concatenation when the first argument is a finite sequence. The *flatten* function has not been defined in this setting; however, a construction similar to *filter* would be necessary.

Definition 7 (Proof Principles). The basic proof principles are structural induction on finite lists and extensionality for infinite sequences. Using *seq*, proofs have to be split up as follows:

$$\frac{(\forall n, f\ .\ P(seq\ (\mathsf{Fin}\ n)\ f)) \qquad (\forall g\ .\ P(seq\ \mathsf{Inf}\ g))}{\forall y\ .\ P(y)}$$

The following more general extensionality proof principle is also available:

$$\frac{len(x) = len(y) \wedge (\forall i < len(x) \,.\, nth\ x\ i = nth\ y\ i)}{x = y}$$

For particular functions as *filter* and $\oplus$, the notions of chains, limits and sometimes continuity are used to prove equality of sequences only by proving their equality for all finite sequences.

After writing the paper we became aware of [Age94], where Agerholm takes the same approach as Chou and Peled, but in a more domain theoretic style and to a much greater extent.

5 PVS-Fun: Functions in PVS

The specification of possibly infinite sequences in PVS by Devillers and Griffioen made use of dependent types. In this manner, sequences are defined as functions from downward closed subsets of the natural numbers to a data set. Below, the definition of the set of all downward closed sets, called index sets, $\mathbb{I}$ is given.

$$\mathbb{I} = \{S \in \wp(\mathbb{N}) \,.\, (\forall i \in S, j \in \mathbb{N} \,.\, j < i \Rightarrow j \in S)\}$$

In the case of finite sequences, the domain of such a function will be an initial segment of the natural numbers which can be constructed with the *below* function (for any $n \in \mathbb{N}$, $below(n)$ is the set of the first n natural numbers $\{0, \ldots, n-1\}$). In case of infinite sequences, the domain of the sequence is the set of natural numbers $\mathbb{N}$. Note, that $\mathbb{I}$ is isomorphic to $\mathbb{N}^\infty$. In the following, $\lfloor S \rfloor$ denotes the smallest element of the set S.

The definition of possibly infinite sequences is given as a dependent product of an index set, and a mapping from that index set to the data set. The sets of finite and infinite sequences are defined with the use of predicate subtyping.

Definition 8 (Type of Sequences).

$$\begin{aligned} A^\infty &= (S \in \mathbb{I} \times (S \to A)) \\ A^\star &= \{x \in A^\infty \,.\, \mathit{finite}(\pi_0(x))\} \\ A^\omega &= \{x \in A^\infty \,.\, \neg \mathit{finite}(\pi_0(x))\} \end{aligned}$$

Note that a tuple of a set and a function is used in this implementation because there does not exists a domain operator in PVS (an operator returning the domain of a given function). In the rest of the paper, we will write $dom(x)$ for the domain of a sequence x, and $x(i)$ for the i-th element in such a sequence.

Simple operators are defined in a straightforward fashion. What is practical about these definitions is that no distinction is made between finite or infinite sequences in the mappings used. As a result, during some proofs no explicit split in reasoning is needed between finite and strictly infinite sequences.

However, sometimes it is needed to make that distinction to derive the appropriate domain for a function. Please consider, for instance, the concatenation operator $\oplus$ defined in the list below.

Definition 9 (Basic Operations).

$$
\begin{array}{ll}
\mathsf{nil} & : \ A^\star \\
\mathsf{nil} & = (\emptyset, f) \ \mathsf{where} \ f \in \emptyset \to A \\
\\
\mathit{len} & : \ A^\star \to \mathbb{N} \\
\mathit{len}(x) & = \sharp \mathit{dom}(x) \\
\\
\mathit{map} & : \ (A \to B) \to A^\infty \to B^\infty \\
\mathit{map}(f)(x) & = (\mathit{dom}(x), (\lambda i : \mathit{dom}(x).f(x(i)))) \\
\\
\oplus & : \ A^\star \times A^\infty \to A^\infty \\
x \oplus y & = (S, (\lambda i : S. \ \mathsf{if} \ i < l \ \mathsf{then} \ x(i) \ \mathsf{else} \ y(i-l) \ \mathsf{fi} \ )) \\
\mathsf{where} & \quad l = \mathit{len}(x), S = \ \mathsf{if} \ \mathit{finite}(y) \ \mathsf{then} \ \mathit{below}(l + \mathit{len}(y)) \ \mathsf{else} \ \mathbb{N} \ \mathsf{fi}
\end{array}
$$

The filter function is basically defined with the use of an enumeration function on ordered sets. Let $W(S,x)$ be the witness set of all indexes i which satisfy $x(i) \in S$, and let $it_{S'}$ be the enumerated sequence of elements of the ordered countable set S'. Then $x \circ it_{W(S,x)}$ is a filtered sequence. For example, suppose one wants to filter all symbols a in the sequence $x = (b,a,a,b,a,\ldots)$. Then $W(\{a\},x) = \{1,2,4,\ldots\}$, and $it_{W(\{a\},x)}$ is the sequence $(1,2,4,\ldots)$. Therefore, $x \circ it_{W(\{a\},x)} = (b,a,a,b,a,\ldots) \circ (1,2,4,\ldots) = (a,a,a,\ldots)$.

Definition 10 (Filter).

$$
\begin{array}{ll}
W(S,x) & = \{i \in \mathit{dom}(x) \mid x(i) \in S\} \\
\tilde{S} & = \ \mathsf{if} \ \mathit{finite}(S) \ \mathsf{then} \ \mathit{below}(\sharp(S)) \ \mathsf{else} \ \mathbb{N} \ \mathsf{fi} \\
S^{-0} & = S \\
S^{-(n+1)} & = \begin{cases} \emptyset & , S^{-n} = \emptyset \\ S^{-n} \setminus \lfloor S^{-n} \rfloor & , \ \mathsf{otherwise} \end{cases} \\
\mathit{it}_S(i) & = \lfloor S^{-i} \rfloor \\
\mathit{filter}(S,x) & = (\tilde{W}(S,x), x \circ \mathit{it}_{W(S,x)})
\end{array}
$$

Although most proofs concerning sequence operators are simple in this setting, a proof of even a simple property about *filter* is complicated (which in a similar fashion is expected for *flatten* that has not been formalized yet). Proofs performed about *filter* include proofs that the *it* function is a monotonic bijective function, and of the primitive recursive characterization of *filter*:
$\mathit{filter}_S(a \hat{} x) = \ \mathsf{if} \ a \in S \ \mathsf{then} \ a \hat{} \mathit{filter}_S x \ \mathsf{else} \ \mathit{filter}_S x \ \mathsf{fi}$

Definition 11 (Proof Principles). The most used proof principle in this setting is called extensionality, point-to-point wise equality

$$\frac{\mathit{dom}(x) = \mathit{dom}(y) \wedge (\forall i \in \mathit{dom}(x) \ . \ x(i) = y(i))}{x = y}$$

As a corollary, we would like to mention that properties over down-ward closed subsets of the natural numbers can easily be proven with a generalized induction scheme on these subsets. Let S be a down-ward closed subset of the natural numbers then

$$\frac{((0 \in S \Rightarrow p(0)) \wedge (\forall (n+1) \in S \,.\, p(n) \Rightarrow p(n+1)))}{(\forall n \in S \,.\, p(n))}$$

For finite sequences, also structural inductions rules and induction to the length of sequences are given.

6 HOL-LCF: Domain Theory in Isabelle/HOL

Definition 12 (Type of Sequences). Using the HOLCF datatype package sequences are defined by the simple recursive domain equation

$$\textbf{domain}\ (\alpha)\mathsf{Seq} = \mathsf{nil} \mid (\alpha) \star (\textbf{lazy}\ (\alpha)\mathsf{Seq})$$

where nil and the "cons"-operator $\star$ are the constructors of the datatype. By default domain constructors are strict, therefore $\star$ is strict in its first argument and lazy in the second. This means, that elements of the type $(\alpha)\mathsf{Seq}$ come in three flavors:

- Finite total sequences: $a_1 \star \ldots \star a_n \star \mathsf{nil}$
- Finite partial sequences: $a_1 \star \ldots \star a_n \star \bot$
- Infinite sequences: $a_1 \star a_2 \star a_3 \ldots$

The domain package automatically proves a number of user-relevant theorems, *e.g.* concerning the constructors, discriminators, and selectors of the datatype.

Sequence Elements in HOL. Domain definitions, like $(\alpha)\mathsf{Seq}$, require the argument type α to be in type class pcpo. However, in Müller's case, domains are appropriate for recursively defining sequences, but *elements* in sequences are often easier to handle in a total fashion, as types of class term. Therefore types of class term are lifted to *flat domains* using the type constructor lift:

$$(\alpha)\mathsf{lift} = \mathsf{Undef} \mid \mathsf{Def}(\alpha)$$

Here, both α and $(\alpha)\mathsf{lift}$ are elements of term, but by adding the two definitions

$$\begin{aligned} \bot &= \mathsf{Undef} \\ x \sqsubseteq y &= (x = y) \mid x = \mathsf{Undef} \end{aligned}$$

and proving the properties of a complete partial order with a least element, $(\alpha)\mathsf{lift}$ becomes an instance of pcpo. Note that $\bot$ and $\sqsubseteq$ are overloaded and this definition only fixes their meaning at type $(\alpha)\mathsf{lift}$. In the sequel, $\bot$ is written instead of Undef.

Sequences are now defined as $(\alpha)\mathsf{seq} = ((\alpha)\mathsf{lift})\mathsf{Seq}$ and a new "cons"-operator for elements of type class term is introduced: $x\hat{}xs = (\mathsf{Def}\, x) \star xs$. Using the lift constructor has several advantages:

- If sequence elements do not need support for infinity or undefinedness, we are not forced to press the overhead of domain theory into them, but lift them as late as possible to a domain, just when it is really needed.
- Many datatypes are well supported in HOL, *e.g.* lists or natural numbers. We can make reuse of these theories, theorem libraries, and tailored proof procedures.
- Within the new "cons"-operator $x\hat{}xs$ the Def constructor serves as an implicit tag showing definedness of an element. As we will show later with an example, this simplifies or even eliminates reasoning about the $\bot$ case.

Besides lifting basic types it is necessary to lift also domains and codomains of functions, built by the type constructor $\rightarrow$. Furthermore the automatic proof support for continuity has to be extended. Details can be found in [MN97, MS96].

Definition 13 (Basic Operations). Operations are defined as fixed points, from which recursive equations are derived automatically. For example, *map* has type

$$map \ : \ (\alpha \rightarrow \beta) \rightarrow (\alpha)\mathsf{seq} \rightarrow_c (\beta)\mathsf{seq}$$

and the following rewrite rules

$$\begin{array}{l} map\ f\text{`}\bot = \bot \\ map\ f\text{`}\mathsf{nil} = \mathsf{nil} \\ map\ f\text{`}(x\hat{}xs) = f(x)\hat{}map\ f\text{`}xs \end{array}$$

are automatically derived from the definition

$$\begin{array}{l} map\ f = fix\text{`}(\Lambda h.\Lambda s\,.\,\mathsf{case}\ s\ \mathsf{of}\ \mathsf{nil} \Rightarrow \mathsf{nil} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid (x\hat{}xs) \Rightarrow f(x)\hat{}(h\text{`}xs)) \end{array}$$

According to domain theory, the argument of *fix* in this definition has to be a continuous function in order to guarantee the existence of the least fixed point. This continuity requirement is handled automatically by type checking, as every occuring function is constructed using the continuous function type $\rightarrow_c$.

Note, that the derived recursive equations are just the algebraic definitions of the corresponding functions for finite lists, extended for the $\bot$ case. Therefore, informally speaking, defining operations on finite lists smoothly carries over to infinite lists.

Definition 14 (Filter and Flatten). All other operations are defined likewise easily. This is especially remarkable for the *filter* and *flatten* operations that would cause some trouble especially in the functional formalizations:

$$\begin{array}{l} filter \ : \ (\alpha \rightarrow \mathbb{B}) \rightarrow (\alpha)\mathsf{seq} \rightarrow_c (\alpha)\mathsf{seq} \\ filter\ P\text{`}\bot = \bot \\ filter\ P\text{`}\mathsf{nil} = \mathsf{nil} \\ filter\ P\text{`}(x\hat{}xs) = \ \mathsf{if}\ P(x)\ \mathsf{then}\ x\hat{}filter\ P\text{`}xs\ \mathsf{else}\ filter\ P\text{`}xs \end{array}$$

Flatten is defined in a similar simple fashion. Note, that these fixed point definitions incorporate the intuition of computability. Therefore, lemmas like

filter $P\,(x \oplus y) =$ *filter* $P\,x \oplus$ *filter* $P\,y$ do not only hold for finite x. Consider the example $P = (\lambda x.x = a)$, $x = (a, b, b, b, ...)$ and $y = (a, a, a, a....)$. Whereas in HOL-LCF the mentioned lemma would hold (because $(a, \bot) = (a, \bot)$), in other formalizations this lemma would not $((a, \mathsf{nil}) = (a, a, a...))$.

Definition 15 (Proof Principles). The proof principles that are discussed in the following are all automatically proved by the HOLCF datatype package.

A very strong proof principle is **structural induction**, as it allows one to reason about infinite sequences, as if they were finite, modulo an admissibility requirement:

$$\frac{adm\,(P) \quad P(\bot) \quad P(\mathsf{nil}) \quad (\forall x, xs\,.\,P(xs) \Rightarrow P(x\hat{}xs))}{\forall y.P(y)} \tag{1}$$

Note, how Def serves here as an implicit tag for definedness: In the equivalent rule for $(\alpha)\mathsf{Seq}$ the last assumption of this rule would be $(\forall x, xs\,.\,x \neq \bot \wedge P(xs) \Rightarrow P(x \star xs))$. The nasty case distinction $x \neq \bot$ can be omitted, as $(\mathsf{Def}\,x) \neq \bot$ and $x\hat{}xs = (\mathsf{Def}\,x) \star xs$.

A predicate P is defined to be admissible, denoted by $adm(P)$, if it holds for the least upper bound of every chain satisfying P. However, in practice, one rather uses a syntactic criterion (see *e.g.* [Pau87]): Roughly, it states that if P, reduced to conjunctive normal form, contains no existential quantifier or negation, admissibility of P boils down to continuity of all functions occuring in P. Therefore, if one stays within the LCF sublanguage, admissibility in theses cases can be proven automatically, *i.e.* we get the proof of the infinite case for free. The following exceptions and extensions of the rough guideline above are especially useful when trying to satisfy the syntactic criterion: Firstly, $t(x) \not\sqsubseteq c$ and $t(x) \neq \bot$ are admissible in x, if t is continuous. Secondly, predicates over chain-finite domains are admissible, and finally, substitution maintains admissibility.

Besides (1) and conventional fixed point induction, there are also weaker structural induction rules, that do not need admissibility, namely for the finite case

$$\frac{P(\mathsf{nil}) \quad (\forall x, xs\,.\,P(xs) \wedge \mathit{finite}(xs) \Rightarrow P(x\hat{}xs))}{\forall y\,.\,\mathit{finite}(y) \Rightarrow P(y)}$$

and an analogous rule for the partial case. Furthermore, the **take lemma**

$$\frac{\forall n.take\,n`x = take\,n`y}{x = y}$$

and the **bisimulation** rule, that follows easily from the take lemma, are available:

$$\frac{bisim R \quad R(x, y)}{x = y}$$

where $bisim\ R = \forall x, y.R(x, y) \Rightarrow$
$(x = \bot \Rightarrow y = \bot) \wedge$
$(x = \mathsf{nil} \Rightarrow y = \mathsf{nil}) \wedge$
$(\exists a, x'.x = a\hat{}x' \Rightarrow \exists b, y'.y = b\hat{}y' \wedge R(x', y') \wedge a = b)$

7 Comparison

Comparing the functional Approaches. As mentioned earlier, HOL-Fun and PVS-Fun are similar to the extent that they both use functions to define sequences. To achieve this common goal, two complementary ways are chosen: Whereas HOL-Fun *extends* the *codomain* of the function by the element None modeling partiality, PVS-Fun *restricts* the *domain* of the function. Therefore, the main proof principle within such a setting is extensionality. Since the approaches are very similar and HOL-Fun (at least the version including normal forms) has not been extensively studied, we will concentrate on the experiences made with PVS-Fun.

Experiences with PVS-Fun**.** It turned out that the extensionality principle works very well for the standard operators, since these operators often only perform but simple index transformations on their arguments. As an example, consider the concatenation operator $\oplus$ in the PVS-Fun section. Because these index transformations often involve simple linear expressions, and the PVS prover has considerable support for linear arithmetic, most proofs are done with a minimum of human guidance, typically by just expanding the definitions involved. However, the definition of *filter* was tedious, and proofs of basic properties about it were very hard, since this involved more than just reasoning over basic index transformations. In conclusion, the definition given seems to be too ad-hoc. An approach where in a more general fashion definitions can be given, together with matching proof principles, should be the focus of future research for such a formalization.

Experiences with HOL-Sum**.** The HOL-Sum approach is a pragmatic mixture of algebraic lists for finite sequences and functions for infinite sequences. Equality of sequences therefore can be proven with structural induction for the finite case, and function equality in the infinite case. Therefore proofs show a twofold character. The *filter* function, however, still is a problem, as there the two representations have to be related, since *filter* may produce either finite or infinite sequences from an infinite one. For this reason, notions of chains, limits and continuity were introduced, which in [CP96], however, are only used for proofs about specific functions, whereas general proof principles involving continuity are not developed. Agerholm [Age94] takes this step and carries over the whole world of domain theory to this setting. Agerholm concludes that his development of sequences was long and tedious (50 pages of 70 lines each) and in his opinion rather an "ad-hoc approach". The main difficulties arise from a threefold definition of $\sqsubseteq$ — both sequences finite, both infinite, and finite/infinite — which results in several versions of every single fact throughout the whole development.

Experiences with HOL-LCF**.** HOL-LCF employs domain theory to extend algebraic lists to infinity, so that a uniform approach is obtained. Of the formalizations discussed, it is the most powerful formalization incorporating a number of proof principles, and the largest body of proven lemmata. However, at first sight it seems that domain theory has two drawbacks: all types must denote domains; all functions must be continuous. But the first requirement can effec-

tively be relaxed by the (α)lift type constructor. And the latter rather offers advantages than disadvantages. Firstly, arbitrary recursion can be defined by fixed points. Unfortunately, this means also that definitions of non-continuous functions are delicate, *e.g.* the fair merge function cannot be defined without leaving the LCF sublanguage. Secondly, continuity extends the familiar structural induction rule to infinite objects for free, at least for equations about lazy lists. For general formulae a rather liberal syntactic criterion exists. Here, Müller's experience in formalizing I/O automata was quite encouraging: in almost all cases it was possible (sometimes by reformulating the goal) to satisfy this criterion or to get by with the finite induction rule. In the remaining cases, it seems not to be advisable to prove admissibility via its definition, as this then often becomes the hardest part of the entire proof. Instead, one should switch to other proof principles, that do not require admissibility. These are the take-lemma (which is similar to extensionality), or bisimulation. For these principles a corecursive characterization of the operators would be useful in order to automate coinductive proofs, that usually – compare the experiences by Paulson [Pau97] – involve more case distinctions than the inductive proofs.

Overall Evaluation. In conclusion, we may distinguish three basic proof schemata for sequences: extensionality (point-wise equality), rules using admissibility or at least continuity, and bisimulation. The first of the three principles turned out to be inconvenient in practice to prove equalities of arbitrary functions. The second principle is strong, but builds on top of an extensive theory: In HOLCF this theory is provided, for HOL-Sum it would be a lot of work to incorporate these notions in a more general fashion. Experience with the application of the bisimulation principle to sequences seems to be rather preliminary (see also next section). Of course, it should be possible to derive all three proof principles in every setting. However, proof principles can only easily be applied when corresponding definitions or characterizations of the occuring functions exist. It is not known, for instance, whether it is easy to derive coalgebraic lemmata from definitions given in a functional manner.

Related Work. Concerning coalgebraic approaches, there is, up to our knowledge, no published work on a coalgebraic formalization with an equally large body of lemmata as the formalizations discussed. Paulson [Pau97] provided a mechanization of coinduction and corecursion in Isabelle/HOL – independent of domain theory –, which he applied also to the formalization of lazy lists. Unfortunately, the *filter* function – which indeed turned out to be very crucial – has not been mechanized there. Leclerc and Paulin-Mohring use Coq to formalize possibly infinite sequences coalgebraically as well [LPM93]. A problem is that they cannot express the *filter* function, as it does not fit into their *constructive* framework. Hensel and Jacobs [HJ97] showed how to obtain inductive and coinductive proof principles for datatypes with iterated recursion from a categorical analysis. They formalized a number of these datatypes in PVS and have also some promising recent results in formalizing coalgebraic possibly-infinite sequences. Recently, Feferman [Fef96] developed a recursion theory and applied it to the formalization of sequences. Similar to HOL-LCF, his solution

incorporates finite, partial and infinite sequences. However, it does not require continuity. His approach has not been mechanized in a proof tool yet.

8 Conclusion and Future Research

We compared four formalizations of possibly infinite sequences in different higher-order logics and proof tools. Two of them – the Isabelle/HOLCF and the PVS solution – have been extensively used by the authors to model the meta-theory of I/O automata. The sequence theories include more than 100 theorems and required between 3 and 6 man months.

In general we have the following view on the formalizations; with respect to automation and usability the HOL-LCF package is developed the furthest. It offers a strong definitional scheme, and multiple proof principles for proofs by induction, extensionality or bisimulation.

Although domain theory gets simpler to use and to automate by integrating as much as possible from HOL, some users might be reluctant to take the significant step to switch to domain theory. These users probably will have to develop further one of the other approaches. The HOL-Fun and PVS-Fun approaches were not worked out completely. Within the PVS-Fun approach it became clear that ad-hoc definitions like the *filter* function result in too large proof obligations. The extensionality principle also seems to be not adequate for reasoning about infinite sequences.

The approaches taken by Chou and Peled HOL-Sum and Agerholm [Age94] are pragmatic and more "ad-hoc" ways to deal with sequences. For specific purposes such a theory is built up quickly and may be satisfactory, but in general the twofold or even threefold character of proofs is inconvenient. Basically, the approach suffers from the fact, that domain theory is (partly) used to define recursive functions, but not to define recursive domains, which, however, is the crucial point of domain theory.

Coinductive types are being implemented in different proof tools in the moment. As the packages also offer definitional principles, and coinduction (or bisimulation) seems much stronger than the extensionality principle, they are an interesting candidate for possibly infinite sequences as well. However, to our knowledge, at the moment there is not much experience with coinductive types used in sophisticated verifications.

Acknowledgement. We thank Ching-Tsun Chou for intensive and fruitful discussions on his formalization of sequences.

References

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

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[CP96] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. In T. Margaria and B. Steffen, editors, *Proc. 2nd Workshop Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[DG97] Marco Devillers and David Griffioen. A formalization of finite and infinite sequences in PVS. Technical Report CSI-R9702, Computing Science Institute, University of Nijmegen, 1997.

[Fef96] Solomom Feferman. Computation on abstract data types. the extensional approach, with an application to streams. *Annals of Pure and Applied Logic*, 81:75–113, 1996.

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

[HJ97] U. Hensel and B. Jacobs. Proof principles for datatypes with iterated recursion. Technical Report CSI-R9703, Computing Science Institute, University of Nijmegen, 1997.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LPM93] 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, *Proc. Types for Proofs and Programs (TYPES'93)*, volume 806 of *Lecture Notes in Computer Science*, 1993.

[LT89] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[MN97] Olaf Müller and Tobias Nipkow. Traces of I/O-Automata in Isabelle/HOLCF. In *Proc. 7th Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[MS96] Olaf Müller and Konrad Slind. Isabelle/HOL as a platform for partiality. In *CADE-13 Workshop: Mechanization of Partial Functions, New Brunswick*, pages 85–96, 1996.

[NS95] Tobias Nipkow and Konrad Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 101–119. Springer-Verlag, 1995.

[ORSH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pau87] Lawrence C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Pau97] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Automated Reasoning*, 7, 1997.

[Reg95] Franz Regensburger. HOLCF: Higher Order Logic of Computable Functions. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer-Verlag, 1995.

This article was typeset using the LaTeX macro package with the LLNCS2E class.