Recursive Functions on Lazy Lists via Domains and Topologies

Andreas Lochbihler[1] and Johannes Hölzl[2]

[1] Institute of Information Security, ETH Zurich, andreas.lochbihler@inf.ethz.ch
[2] Institut für Informatik, TU München, hoelzl@in.tum.de

Abstract. The usual definition facilities in theorem provers cannot handle all recursive functions on lazy lists; the filter function is a prime counterexample. We present two new ways of directly defining functions like filter by exploiting their dual nature as producers and consumers. Borrowing from domain theory and topology, we define them as a least fixpoint (producer view) and as a continuous extension (consumer view). Both constructions yield proof principles that allow elegant proofs. We expect that the approach extends to codatatypes with finite truncations.

1 Introduction

Coinductive datatypes (codatatypes for short) are popular in theorem provers [4,5,8,16,18,20], especially to formalise different forms of computation. Possibly infinite (lazy) lists, the most prominent example, are used to e.g. model traces of finite and infinite executions [17]. Today, Isabelle/HOL has a definitional package to construct codatatypes and define primitively corecursive functions [5].

codatatype α llist $=$ [] | $\alpha \cdot \alpha$ llist

Yet, not all functions of interest are primitively corecursive; and the definition facilities based on well-founded recursion [13,21] cannot handle them either, when they produce infinite codatatype values by infinite corecursion. Hence, such functions have to be defined by other means. In this paper, we consider recursive functions that are notoriously hard to define [8], because their recursive specification does not uniquely determine them. In particular, we focus on the best-known example lfilter given by the specification (SPEC).[1]

$$\begin{aligned} &\mathsf{lfilter}\ P\ [] &&= [] \\ &\mathsf{lfilter}\ P\ (x \cdot \overline{xs}) &&= (\mathsf{if}\ P\ x\ \mathsf{then}\ x \cdot \mathsf{lfilter}\ P\ \overline{xs}\ \mathsf{else}\ \mathsf{lfilter}\ P\ \overline{xs}) \end{aligned} \tag{SPEC}$$

Eq. (SPEC) is not primitively corecursive, as no constructor guards the second recursive call. Neither can well-founded recursion handle it, as an infinite list $\overline{xs}$ causes infinite recursion. Nor does (SPEC) fully specify lfilter: for $P = (\lambda\_.\ \mathsf{False})$ and $\overline{xs} = x \cdot x \cdot x \cdot \ldots$ the infinite repetition of some x e.g. (SPEC) collapses to the vacuous condition $\mathsf{lfilter}\ P\ \overline{xs} = \mathsf{lfilter}\ P\ \overline{xs}$, i.e. $\mathsf{lfilter}\ P\ \overline{xs}$ could be any lazy list.

As HOL functions are total anyway, HOL users often "totalise" a function with a partial specification such that proving becomes easier. Following this tra-

[1] We prefix functions on lazy lists with l to distinguish them from their counterpart on finite lists; variables for lazy lists carry overbars $\overline{xs}$, for finite lists underbars $\underline{xs}$.

$$\mathsf{lfilter}\ P\ (\mathsf{lfilter}\ Q\ \overline{xs}) = \mathsf{lfilter}\ (\lambda x.\ P\ x \wedge Q\ x)\ \overline{xs} \qquad (\textsc{conj})$$

$$\mathsf{lset}\ (\mathsf{lfilter}\ P\ \overline{xs}) = \mathsf{lset}\ \overline{xs} \cap \{\, x \mid P\ x \,\} \qquad (\textsc{lset})$$

$$\mathsf{lfilter}\ P\ \overline{xs} = [] \longleftrightarrow \forall x \in \mathsf{lset}\ \overline{xs}.\ \neg P\ x \qquad (\textsc{nil})$$

$$\mathsf{ldistinct}\ \overline{xs} \longrightarrow \mathsf{ldistinct}\ (\mathsf{lfilter}\ P\ \overline{xs}) \qquad (\textsc{ldistinct})$$

$$\mathsf{lrel}\ R\ \overline{xs}\ \overline{ys} \wedge (\forall x\ y.\ R\ x\ y \longrightarrow (P_1\ x \longleftrightarrow P_2\ y)) \longrightarrow \mathsf{lrel}\ R\ (\mathsf{lfilter}\ P_1\ \overline{xs})\ (\mathsf{lfilter}\ P_2\ \overline{ys}) \qquad (\textsc{lrel})$$

Fig. 1: Proven properties of lfilter

dition, we want to define lfilter such that $\mathsf{lfilter}\ P\ \overline{xs} = []$ whenever $\overline{xs}$ contains no elements satisfying P. This way, equations like (CONJ) and (LSET) in Fig. 1 hold unconditionally, even if $\overline{xs}$ is infinite and none of its elements satisfies P (and Q).

Of course, lfilter can be defined in an ad hoc fashion (see §5), but this has two drawbacks. First, one must come up with another construction for each new function. Second, such constructions typically lack a proof principle. Thus, proofs get cluttered with construction details as the definition must be unfolded.

In this work, we present two approaches to defining functions such as lfilter. They are inspired by two views on specifications like (SPEC). First, we can think of lfilter as corecursively producing a list lazily, i.e. a function of type $\beta \Rightarrow \alpha$ llist for some state type β; when another element is requested, it calls itself with an updated state. Borrowing ideas from domain theory, we turn α llist into a complete partial order, lift it point-wise to $\beta \Rightarrow \alpha$ llist, and take the least fixpoint of the functional associated with (SPEC) for lfilter (§2). Alternatively, we can also view lfilter as recursively consuming a lazy list, i.e. a function of type α llist$\Rightarrow\beta$ for some result type β. In §3, we therefore define lfilter on finite lists by (primitive) recursion and continuously extend it to infinite lists via topological limits.

Clearly, the two approaches require more machinery than ad hoc constructions. But more importantly, both approaches yield proof principles: either structural induction on lazy lists and fixpoint induction (§2.3) or uniqueness of limits and convergence on a closed set (§3.2). They allow elegant proofs with a high degree of automation. To show them in action, we prove the five exemplary properties listed in Fig. 1. Since lfilter both produces and consumes a lazy list, it is a good example to compare the two approaches. We do so in §4.

In this paper, we focus on lfilter, but we have defined more functions on lazy lists this way. Our approach simplifies their (formerly ad hoc) definitions and the proofs in an existing codatatype library [16]. We expect that our approach generalises to a large class of codatatypes (§6) and can be ported to other systems.

2 The Producer View: Least Fixpoints

In this section, we formalise lfilter as the least fixpoint solution to (SPEC). This construction views lfilter as a function that produces a lazy list. First, we define lfilter as a least fixpoint (§2.2) borrowing ideas from domain theory (introduced in §2.1). Next, we set up the infrastructure for the induction proofs (§2.3). Finally, we show how to prove the five properties of lfilter (§2.4) listed in Fig. 1.

2.1 Background on orders and fixpoints

In this section, we review some domain theory formalised in plain HOL [14].

An *order* $\leq$ for a given type is a binary relation that is reflexive, transitive, and antisymmetric. Given an order $\leq$, a *chain* Y is a set whose elements are all related in $\leq$ (predicated by $\mathsf{chain}\ (\leq)\ Y$). An order $\leq$ on some type α and a function $\bigvee :: \alpha\ \mathsf{set} \Rightarrow \alpha$ form a *chain-complete partial order* (ccpo) iff $\bigvee Y$ denotes the least upper bound (lub) of every chain Y w.r.t. $\leq$, i.e. for all Y with $\mathsf{chain}\ (\leq)\ Y$, if $x \in Y$, then $x \leq \bigvee Y$, and whenever $x \leq z$ for all $x \in Y$, then $\bigvee Y \leq z$. As the empty set is a chain, every ccpo has a least element bottom $\bot = \bigvee \emptyset$. For example, the type of sets $\alpha\ \mathsf{set}$ ordered by inclusion $\subseteq$ forms a ccpo with lub $\bigcup Y$ and bottom $\emptyset$. An order $\leq$ is lifted pointwise to functions: $f \mathbin{\uparrow\leq} g$ denotes $\forall x.\ f\ x \leq g\ x$. Analogously, the lub $\uparrow\bigvee Y$ on a chain Y of functions is determined pointwise: $\uparrow\bigvee Y\ x = \bigvee\{f\ x.\ f \in Y\}$. If $(\bigvee, \leq)$ is a ccpo, so is $(\uparrow\bigvee, \uparrow\leq)$. Similarly, $\leq\times\leq'$ orders pairs component-wise according to $\leq$ and $\leq'$, resp.; and $(\bigvee\times\bigvee')Y = (\bigvee(\pi_1\text{`}Y), \bigvee'(\pi_2\text{`}Y))$ computes the lub component-wise. Here, π_1 and π_2 are the projections, and $f\text{`}A$ denotes the image of the set A under the function f. If $(\bigvee, \leq)$ and $(\bigvee', \leq')$ are ccpos, so is $(\bigvee\times\bigvee', \leq\times\leq')$.

A function f is *monotone* w.r.t. $\leq$ and $\leq'$ (written $\mathsf{mono}\ (\leq)\ (\leq')\ f$) iff $f\ x \leq' f\ y$ for all x, y with $x \leq y$. A monotone function f is *(order) continuous* w.r.t. $(\bigvee, \leq)$ and $(\bigvee', \leq')$ iff it preserves lubs of non-empty chains (written $\mathsf{mcont}\ (\bigvee, \leq)\ (\bigvee', \leq')\ f$). Formally, $f\ (\bigvee Y) = \bigvee'(f\text{`}Y)$ for all Y with $\mathsf{chain}\ (\leq)\ Y$ and $Y \neq \emptyset$. A continuous function f is *strict* iff it propagates $\bot$, i.e. $f\ (\bigvee\emptyset) = \bigvee'\emptyset$. A predicate P is *admissible* (written $\mathsf{adm}\ (\bigvee, \leq)\ P$) iff $P\ (\bigvee Y)$ for all non-empty chains Y such that $P\ x$ for all $x \in Y$. Admissibility is closed under composition with continuous functions, i.e. if $\mathsf{adm}\ (\bigvee, \leq)\ (\lambda x.\ P\ x)$ and $\mathsf{mcont}\ (\bigvee', \leq')\ (\bigvee, \leq)\ f$, then $\mathsf{adm}\ (\bigvee', \leq')\ (\lambda x.\ P\ (f\ x))$.

Let $F :: \alpha \Rightarrow \alpha$ be a monotone function on a ccpo $(\bigvee, \leq)$. Then, by the Knaster-Tarski fixpoint theorem, F has a *least fixpoint* $\mathsf{fixp}\ (\bigvee, \leq)\ F$, which is given by the lub of the transfinite iteration of F starting at $\bot$.

2.2 Definition

As mentioned in §1, we define $\mathsf{lfilter}\ P$ as the least fixpoint of the functional associated to the specification (SPEC); since $\mathsf{lfilter}$ passes the predicate P unchanged to the recursive calls, we treat it as a fixed parameter. Thus, we obtain the functional $\mathsf{F}_P :: (\alpha\ \mathsf{llist} \Rightarrow \alpha\ \mathsf{llist}) \Rightarrow (\alpha\ \mathsf{llist} \Rightarrow \alpha\ \mathsf{llist})$ given by

$$\mathsf{F}_P\ f\ \overline{xs} = (\mathsf{case}\ \overline{xs}\ \mathsf{of}\ [] \Rightarrow [] \mid x \cdot \overline{xs'} \Rightarrow \mathsf{if}\ P\ x\ \mathsf{then}\ x \cdot f\ \overline{xs'}\ \mathsf{else}\ f\ \overline{xs'}) \quad (1)$$

For the Knaster-Tarski fixpoint theorem, we need a ccpo on $\alpha\ \mathsf{llist} \Rightarrow \alpha\ \mathsf{llist}$ for which F_P is monotone. It suffices to provide one for $\alpha\ \mathsf{llist}$ and lift it point-wise to functions with codomain $\alpha\ \mathsf{llist}$. We choose the prefix order $\sqsubseteq$, which (2) defines coinductively. The least upper bound $\bigsqcup :: \alpha\ \mathsf{llist}\ \mathsf{set} \Rightarrow \alpha\ \mathsf{llist}$ is given by primitive corecursion (3). Here, lhd and ltl return the head and tail of a lazy list, resp.; and the definite descriptor $\iota x.\ P\ x$ denotes the unique x such that $P\ x$ if it exists and is unspecified otherwise. We show the ccpo properties for $(\bigsqcup, \sqsubseteq)$ by (rule or structural) coinduction.

$$\frac{}{[] \sqsubseteq \overline{ys}} \qquad \frac{\overline{xs} \sqsubseteq \overline{ys}}{x \cdot \overline{xs} \sqsubseteq x \cdot \overline{ys}} \tag{2}$$

$$\begin{aligned} \bigsqcup Y = (&\mathsf{if}\ Y \subseteq \{[]\}\ \mathsf{then}\ [] \\ &\mathsf{else\ let}\ Y' = \{\overline{xs} \in Y.\, \overline{xs} \neq []\}\ \mathsf{in}\ (\iota x.\, x \in \mathsf{lhd}\ `\ Y') \cdot \bigsqcup(\mathsf{ltl}\ `\ Y')) \end{aligned} \tag{3}$$

The prefix order is a natural choice, as it makes the constructor $\_ \cdot \_$ monotone in the recursive argument. Monotonicity is crucial for the existence of the fixpoint (see below). Moreover, the least element $[]$ carries the least information possible. In fact, $\sqsubseteq$ corresponds to the approximation order on the domain of infinite streams, when we interpret $[]$ as "undefined", the additional value that each domain contains. In this view, a finite lazy list represents the set of all its extensions at the end, and this set shrinks when we extend it.

Now, we define lfilter as the least fixpoint of F_P in the ccpo $(\uparrow\bigsqcup, \uparrow\sqsubseteq)$ using the **partial-function** package by Krauss [14]. Given the specification (SPEC) as input, it constructs the functional F_P, proves monotonicity, defines lfilter as the least fixpoint, and derives (SPEC) and a fixpoint induction rule (4) from the definition. The monotonicity proof decomposes the functional syntactically into primitive operations and uses their monotonicity properties. For F_P, we provide the monotonicity theorem for $\cdot$ as a hint, which follows directly from (2).

This completes our first definition of lfilter. After some preparations (§2.3), we prove in §2.4 that lfilter is in fact the desired solution for (SPEC).

2.3 Preparations for Proofs by Induction

Least fixpoints and the ccpo structure on lazy lists provide two induction proof principles, which we review now. Every least fixpoint definition generates an induction rule; the one for lfilter is shown in (4). The second premise requires that the statement Q to be proved holds for the least function $\lambda\_.\ []$ where the fixpoint iteration starts; and in the inductive step (third premise), some underapproximation f replaces the function lfilter P. Admissibility (first premise) ensures that Q is preserved when taking the lub of the iteration for the fixpoint.

$$\frac{\mathsf{adm}\ (\uparrow\bigsqcup, \uparrow\sqsubseteq)\ Q \qquad Q\ (\lambda\_.\ []) \qquad \forall f.\, Q\ f \wedge f \uparrow\sqsubseteq \mathsf{lfilter}\ P \longrightarrow Q\ (\mathsf{F}_P\ f)}{Q\ (\mathsf{lfilter}\ P)} \tag{4}$$

Fixpoint induction corresponds to the producer view, as it assumes nothing about the parameter; rather, f in the inductive step returns a prefix of lfilter P.

Alternatively, structural induction over a lazy list (5) is available. The inductive cases (second and third premise) yield that the property P holds for all finite lists (predicate lfinite). Admissibility (first premise) ensures that P also holds for the whole list $\overline{xs}$, as all finite prefixes of $\overline{xs}$ form a chain with lub $\overline{xs}$. Clearly, structural induction takes the consumer point of view, because in typical use cases, it acts on a variable $\overline{xs}$ that a function takes as argument.

$$\frac{\mathsf{adm}\ (\bigsqcup, \sqsubseteq)\ P \qquad P\ [] \qquad \forall x\ \overline{xs}.\, \mathsf{lfinite}\ \overline{xs} \wedge P\ \overline{xs} \longrightarrow P\ (x \cdot \overline{xs})}{P\ \overline{xs}} \tag{5}$$

Both induction principles require that the inductive statement is admissible. Müller et al. [19] have already noted in the context of Isabelle's LCF formalisation HOLCF that admissibility is often harder to prove than the inductive steps. Huffman [12] describes the syntax-directed approach to automate these proofs. Proof rules such as (6) first decompose the statement into atoms along the logical connectives. Others then separate each atom into a predicate and its arguments (interpreted as a function of the induction variable). If the arguments are continuous, it suffices to show admissibility of the predicate; HOLCF includes admissibility rules for comparisons and (in)equalities. This approach works well in practice, because all HOLCF functions are continuous by construction.

$$\frac{\mathsf{adm}\ (\bigvee, \leq)\ (\lambda x.\, \neg\, P\ x) \qquad \mathsf{adm}\ (\bigvee, \leq)\ (\lambda x.\, Q\ x)}{\mathsf{adm}\ (\bigvee, \leq)\ (\lambda x.\, P\ x \longrightarrow Q\ x)} \tag{6}$$

We have proved a similar set of syntax-directed proof rules. They achieve a comparable degree of automation for discharging admissibility conditions. However, they require some manual setup, in particular continuity proofs. We will discuss this now at three examples, namely lfilter, lmap and lset.

First, we prove that lfilter P is continuous. This will allow us later to switch from the producer to the consumer view, i.e. from (4) to (5). As we have defined lfilter P as a least fixpoint, we can leverage the general result that least fixpoints preserve monotonicity and continuity (Thm. 1).

Theorem 1 (Least fixpoints preserve monotonicity and continuity). *Let $(\bigvee, \leq)$ be a ccpo, let $F :: (\beta \Rightarrow \alpha) \Rightarrow (\beta \Rightarrow \alpha)$ satisfy mono $(\uparrow\leq)$ $(\uparrow\leq)$ F. If F preserves monotonicity (continuity), then F's least fixpoint is monotone (continuous). Formally, let g abbreviate fixp $(\uparrow\bigvee, \uparrow\leq)$ F. If mono $(\leq')$ $(\leq)$ $(F\ f)$ for all f with mono $(\leq')$ $(\leq)$ f, then mono $(\leq')$ $(\leq)$ g. If mcont $(\bigvee', \leq')$ $(\bigvee, \leq)$ $(F\ f)$ for all f with mcont $(\bigvee', \leq')$ $(\bigvee, \leq)$ f, then mcont $(\bigvee', \leq')$ $(\bigvee, \leq)$ g.*

Hence, it suffices to show that F_P in (1) is monotone and continuous in $\overline{xs}$ provided that f is so, too. Like for admissibility, we follow a syntax-directed decomposition approach, as continuity is preserved under function composition. We prove rules that decompose the expression into individual functions and then show that they themselves are continuous. Unfortunately, control constructs like case and if are in general neither monotone nor continuous if the branching term depends on the argument. In (1), this is the case for the case combinator.

As we frequently prove functions on α llist strict, we derive a specialised continuity rule (7) (and an analogous monotonicty rule) for an arbitrary ccpo $(\bigvee, \leq)$ with bottom $\bot$. (It cannot handle non-strict functions like $+\!+$ defined in (27).)

$$\frac{\forall x.\, \mathsf{mcont}\ (\bigsqcup, \sqsubseteq)\ (\bigvee, \leq)\ (\lambda \overline{ys}.\, f\ x\ \overline{ys}\ (x \cdot \overline{ys}))}{\mathsf{mcont}\ (\bigsqcup, \sqsubseteq)\ (\bigvee, \leq)\ (\lambda \overline{xs}.\, \mathsf{case}\ \overline{xs}\ \mathsf{of}\ [] \Rightarrow \bot \mid x \cdot \overline{ys} \Rightarrow f\ x\ \overline{ys}\ \overline{xs})} \tag{7}$$

By (7), it suffices to show for all x that $\lambda \overline{ys}.\, \mathsf{if}\ P\ x\ \mathsf{then}\ x \cdot f\ \overline{ys}\ \mathsf{else}\ f\ \overline{ys}$ is monotone and continuous if f already is. Note that the branching condition $P\ x$ no longer depends on the bound variable $\overline{ys}$, so it suffices to prove that the individual branches are monotone and continuous; rule (8) formalises this.

$$\frac{\mathsf{mcont}\ (\bigvee,\leq)\ (\bigvee',\leq')\ f \qquad \mathsf{mcont}\ (\bigvee,\leq)\ (\bigvee',\leq')\ g}{\mathsf{mcont}\ (\bigvee,\leq)\ (\bigvee',\leq')\ (\lambda x.\,\mathsf{if}\ c\ \mathsf{then}\ f\ x\ \mathsf{else}\ g\ x)} \tag{8}$$

Finally, we are left with proving that $\lambda\overline{ys}.\, x \cdot f\ \overline{ys}$ and $\lambda\overline{ys}.\, f\ \overline{ys}$ are monotone and continuous, which follows immediately from $\cdot$ and f being so. Although this proof seems lengthy on paper, it is a one-liner in Isabelle, as its rewriting engine performs the decomposition automatically thanks to the setup outlined above.

The above illustrates how to prove continuity of functions defined in terms of fixp. Other functions are defined by other means, but we want to prove them continuous, too. For example, the codatatype packages defines lmap f, which applies f to all elements of a lazy list (9), and lset, which converts a lazy list to the set of its elements, but not in terms of fixp. The easiest way to prove continuity is to show that they are the least fixpoint of the functionals in (10) and (11), resp. Then, we reuse Thm. 1 and our machinery from above.

$$\mathsf{lmap}\ f\ [] = [] \qquad \mathsf{lmap}\ f\ (x \cdot \overline{xs}) = f\ x \cdot \mathsf{lmap}\ f\ \overline{xs} \tag{9}$$

$$\mathsf{M}_f\ g\ \overline{xs} = (\mathsf{case}\ \overline{xs}\ \mathsf{of}\ [] \Rightarrow [] \mid x \cdot \overline{ys} \Rightarrow f\ x \cdot g\ \overline{ys}) \tag{10}$$

$$\mathsf{S}\ f\ \overline{xs} = (\mathsf{case}\ \overline{xs}\ \mathsf{of}\ [] \Rightarrow \emptyset \mid x \cdot \overline{ys} \Rightarrow \{x\} \cup f\ \overline{ys}) \tag{11}$$

The proofs for $\mathsf{lmap}\ f = \mathsf{fixp}\ (\uparrow\bigsqcup, \uparrow\sqsubseteq)\ \mathsf{M}_f$ and $\mathsf{lset} = \mathsf{fixp}\ (\uparrow\bigcup, \uparrow\subseteq)\ \mathsf{S}$ fall into two parts: (i) monotonicity of M_f and S is shown by **partial-function**'s monotonicity prover and (ii) the actual fixpoint equation by the proof principle associated with the definition (structural coinduction for lmap; lset requires two separate directions with induction on lset and fixpoint induction, resp.). Monotonicity is needed to unfold the fixpoint property in the (co)inductive steps.

2.4 Proving the Properties

With all these preparations in place, we now show how they yield concise proofs for the properties of interest. We start with (NIL), i.e. that the least fixpoint indeed picks the desired solution for (SPEC). First, we illustrate the obvious approach of proving the two directions separately. From right to left, given $\neg P\ x$ for all $x \in \mathsf{lset}\ \overline{xs}$, we must show $\mathsf{lfilter}\ P\ \overline{xs} = []$, or, equivalently, $\mathsf{lfilter}\ P\ \overline{xs} \sqsubseteq []$. Structural coinduction does not work here, as (SPEC) may recurse forever, but fixpoint induction is good at proving upper bounds, [] in our case. Admissibility of $\lambda f.\, \forall\overline{xs}.\, (\forall x \in \mathsf{lset}\ \overline{xs}.\, \neg P\ x) \longrightarrow f\ \overline{xs} \sqsubseteq []$ follows directly from decomposition and admissibility of comparisons. In contrast, from left to right by contraposition, we have to prove the non-trivial lower bound $[] \sqsubset \mathsf{lfilter}\ P\ \overline{xs}$ under the assumption $P\ x$ for some $x \in \mathsf{lset}\ \overline{xs}$. Fixpoint induction cannot do this, so we resort to other proof principles. Fortunately, induction on $x \in \mathsf{lset}\ \overline{xs}$ is available, and the cases are solved automatically by rewriting.

Alternatively, we can switch to the consumer view and prove (NIL) directly by induction on $\overline{xs}$ using (5). Rewriting solves the inductive cases. Regarding admissibility of $\lambda\overline{xs}.\, \mathsf{lfilter}\ P\ \overline{xs} = [] \longleftrightarrow \forall x \in \mathsf{lset}\ \overline{xs}.\, \neg P\ x$, the rules decompose it into four atoms: $\lambda\overline{xs}.\, \mathsf{lfilter}\ P\ \overline{xs} \neq []$ and $\lambda\overline{xs}.\, \mathsf{lfilter}\ P\ \overline{xs} = []$ and $\lambda\overline{xs}.\, \forall x \in \mathsf{lset}\ \overline{xs}.\, \neg P\ x$ and $\lambda\overline{xs}.\, \exists x \in \mathsf{lset}\ \overline{xs}.\, P\ x$. For the (in)equalities and

bounded quantifiers, we have admissibility rules, and their arguments $\mathsf{lfilter}\ P$ and lset are continuous by §2.3. Therefore, this proof of (NIL) is automatic.

lemma $\mathsf{lfilter}\ P\ \overline{xs} = [] \longleftrightarrow \forall x \in \mathsf{lset}\ \overline{xs}.\ \neg\, P\ x$ **by**(induction $\overline{xs}$) simp-all

Next, we prove property (CONJ) from the introduction. Taking the consumer view, the proof is a one-liner by induction on $\overline{xs}$ plus rewriting, because we have already shown that $\mathsf{lfilter}\ P$ is continuous. Fixpoint induction can also prove (CONJ), but the two directions "$\sqsubseteq$" and "$\sqsupseteq$" must be shown separately. Moreover, we still need continuity of $\mathsf{lfilter}\ P$ for admissibility, because when going from left to right, we have to replace $\mathsf{lfilter}\ Q$ in the context $\forall \overline{xs}.\ \mathsf{lfilter}\ P\ (\bullet\ \overline{xs}) \sqsubseteq \ldots$.

Property (LSET) is similar to (CONJ). We show it by induction on $\overline{xs}$; admissibility requires continuity of lset, $\mathsf{lfilter}$, and $\cap$. Fixpoint induction is also possible.

In the remainer of this section, we prove two more properties with user-defined predicates. The predicate $\mathsf{ldistinct}$ denotes that all elements of a lazy list are distinct, and the relator $\mathsf{lrel}\ R\ \overline{xs}\ \overline{ys}$ lifts a binary relation R point-wise to the lazy lists $\overline{xs}$ and $\overline{ys}$. The rules below define them coinductively.

$$\frac{}{\mathsf{ldistinct}\ []} \qquad \frac{x \notin \mathsf{lset}\ \overline{xs} \qquad \mathsf{ldistinct}\ \overline{xs}}{\mathsf{ldistinct}\ (x \cdot \overline{xs})} \tag{12}$$

$$\frac{}{\mathsf{lrel}\ R\ []\ []} \qquad \frac{R\ x\ y \qquad \mathsf{lrel}\ R\ \overline{xs}\ \overline{ys}}{\mathsf{lrel}\ R\ (x \cdot \overline{xs})\ (y \cdot \overline{ys})} \tag{13}$$

Proofs by induction require admissibility of the statement. As $\mathsf{ldistinct}$ is a new predicate, we prove admissibility directly by unfolding the definition and by coinduction on $\mathsf{ldistinct}$. The proof for lrel is similar. Moreover, we also show that non-distinctness is admissible; this follows from prefixes of distinct lists being distinct. Now, we are ready to show properties (LDISTINCT) and (LREL) from Fig. 1.

Taking the consumer view, we show (LDISTINCT) by induction on $\overline{xs}$; as $\overline{xs}$ occurs in the assumptions, rule (6) requires that the negated assumption, i.e. non-distinctness, be admissible, too (there is no rule for negation). The inductive steps are solved automatically, as we can rewrite $\mathsf{lset}\ (\mathsf{lfilter}\ P\ \overline{xs})$ with (LSET).

Alternatively, we can also take the producer view, i.e. fixpoint induction on $\mathsf{lfilter}$. This demonstrates another limitation of fixpoint induction: recall that fixpoint induction replaces $\mathsf{lfilter}\ P$ by some underapproximation f, i.e. we cannot use (LSET) for rewriting $\mathsf{lset}\ (f\ \overline{xs})$. Fortunately, we get $f \uparrow\sqsubseteq \mathsf{lfilter}\ P$ in the inductive step and derive $\mathsf{lset}\ (f\ \overline{xs}) \subseteq \mathsf{lset}\ \overline{xs}$ by monotonicity of lset. Otherwise, we would have had to re-prove (LSET) simultaneously in the inductive step. This modularity problem frequently arises with fixpoint induction: all required properties of a function have to be threaded through one big induction, which incurs losses in proof automation and processing speed.

Finally, consider (LREL). Note that the property of not being related in lrel is not admissible. This means that the decomposition rules do not work if the induction variable under lrel in an assumption. Thus, we cannot induct over $\overline{xs}$ (unless we prove admissibility manually, but we would rather not). We use fixpoint induction instead. Yet, the two occurrences of $\mathsf{lfilter}$ in (LREL) have different

types. As (4) replaces only occurrences of the same type, we resort to parallel fixpoint induction. The general parallel fixpoint induction rule for two ccpos $(\bigvee, \leq)$ and $(\bigvee', \leq')$ with least elements $\bot$ and $\bot'$ and two monotone functionals F and G is shown below. Since the projections π_1 and π_2 are monotone and continuous, the parallel fixpoint induction proof becomes fully automatic again.

$$\frac{\mathsf{adm}\ (\bigvee\times\bigvee', \leq\times\leq')\ (\lambda x.\, P\,(\pi_1\ x)\,(\pi_2\ x)) \quad P\ \bot\ \bot' \quad \forall x\ y.\, P\ x\ y \longrightarrow P\,(F\ x)\,(G\ y)}{P\ (\mathsf{fixp}\ (\bigvee, \leq)\ F)\ (\mathsf{fixp}\ (\bigvee', \leq')\ G)}$$

3 The Consumer View: Continuous Extensions

Some proofs about lfilter in §2.4 already took the consumer point of view. Now, we do so also for defining lfilter. In general, we first define a function on finite lists α list and then extend it to lazy lists. For the running example, we first define filter :: $(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha\ \mathsf{list} \Rightarrow \alpha\ \mathsf{list}$ on finite lists using primitive recursion (14). Then, we define lfilter as the *continuous extension* of filter (15).

$$\begin{aligned} \mathsf{filter}\ P\ [] &= [] \\ \mathsf{filter}\ P\ (x \cdot \underline{xs}) &= (\mathsf{if}\ P\ x\ \mathsf{then}\ x \cdot \mathsf{filter}\ P\ \underline{xs}\ \mathsf{else}\ \mathsf{filter}\ P\ \underline{xs}) \end{aligned} \tag{14}$$

$$\mathsf{lfilter}\ P\ \overline{xs} = \bigsqcup\{\lceil \mathsf{filter}\ P\ \lfloor \overline{ys} \rfloor \rceil \mid \overline{ys} \in {\downarrow}\overline{xs}\} \tag{15}$$

where ${\downarrow}\overline{xs} = \{\, \overline{ys} \mid \mathsf{lfinite}\ \overline{ys} \wedge \overline{ys} \sqsubseteq \overline{xs} \,\}$ denotes the set of finite prefixes of $\overline{xs}$, $\lceil\_\rceil$ embeds finite lists in lazy lists, and $\lfloor\_\rfloor$ is its inverse. This construction yields the same function as the least fixed point in §2.2—see §3.4 for the proofs.

Why do we call this a continuous extension? To generalise this construction method, we introduce a topology on lazy lists with two properties (§§3.2, 3.3). First, every chain of finite lists "converges" towards a lazy (possibly finite) list. Second, every lazy list can be "approximated" by a set of finite lists. Hence, continuous extensions are unique if they exist. So, we extend a function f :: $\alpha\ \mathsf{list} \Rightarrow \beta$ to a function lf :: $\alpha\ \mathsf{llist} \Rightarrow \beta$ by picking the continuous one. This also explains why this is the consumer view: the codatatype is an argument to the function, and the codomain is an arbitrary topology. For unique extensions, the codomain must be a T2 topology.

3.1 Topology in Isabelle/HOL

This section summarises the formalisation of topologies in Isabelle/HOL [11].[2]

A *topology* is specified by the *open sets* (predicate open). In a topology, the whole space must be open (its elements are called *points*), and binary intersection and arbitrary union must preserve openness. A predicate P is a *neighbourhood* of a point x if it holds on an open set which contains x. A *punctured neighbourhood* P of x (written $P\ \mathsf{at}\ x$) is a neighbourhood of x which not necessary holds on x.

[2] As the topology formalisation relies on type classes, we now switch to type classes for ccpos, too. Hence, we no longer write the ccpo $(\bigvee, \leq)$ as a parameter for constants like adm and mcont. Instead, it is taken from the type class.

A point x is *discrete* iff $\{x\}$ is open. A topology is called a *T2 space*, if for every two points $x \neq y$ there exists two disjoint neighbourhoods P_x at x and P_y at y.

A function f *converges* on a point x (written $f \xrightarrow{x} y$) iff for all open sets Y around y the predicate $\lambda x.\ f\ x \in Y$ is a punctured neighbourhood of x. The function f is *continuous* at x iff $f \xrightarrow{x} f\ x$. Clearly, convergence is meaningless for discrete points x, as $\{x\}$ is open. Also, each f is then continuous at x.

A set is *closed* iff its complement is open, a predicate P is closed iff $\{x \mid P\ x\}$ is closed. Closedness of predicates is preserved under composition with continuous functions. *Convergence on a closed set* (16) is our main proof principle. If the predicate $P \circ f$ is a punctured neighbourhood of x and P is closed ($\mathsf{closed}\ \{x \mid P\ x\}$), then P also holds at the point x itself, unless x is discrete.

$$\frac{\neg\,\mathsf{open}\ \{x\} \qquad f \xrightarrow{x} y \qquad \mathsf{closed}\ \{x \mid P\ x\} \qquad P \circ f\ \mathsf{at}\ x}{P\ y} \tag{16}$$

3.2 Topology on a Chain-Complete Partial Order

In this section, we introduce a topology for ccpos. In the ccpo topology, an open set is not accessible from outside, i.e. whenever the least upper bound of a non-empty chain is in the open set, then their intersection is not empty (17).

$$\mathsf{open}\ S \longleftrightarrow (\forall C.\ \mathsf{chain}\ C \longrightarrow C \neq \emptyset \longrightarrow \bigvee C \in S \longrightarrow C \cap S \neq \emptyset) \tag{17}$$

It differs from the usual Scott topology only in that open sets need not be upward closed. We omit this condition for two reasons: (i) we need a T2 space, but the Scott topology is not, and (ii) finite lists should be discrete, i.e. $\mathsf{open}\ \{\overline{xs}\}$ if $\overline{xs}$ is finite. Every ccpo topology is a T2 space, since open as defined in (17) fulfills the topology axioms and separation of points.

As mentioned in §3.1, convergence $f \xrightarrow{x} y$ ignores the value of f at the point x. Thus, if x is discrete, convergence is meaningless at this point. To avoid this issue, we introduce variants of convergence and punctured neighbourhood.[3]

$$f \xrightarrow{x}{}' y \longleftrightarrow \mathsf{if}\ \mathsf{open}\ \{x\}\ \mathsf{then}\ f\ x = y\ \mathsf{else}\ f \xrightarrow{x} y \tag{18}$$

$$P\ \mathsf{at}'\ x \longleftrightarrow \mathsf{if}\ \mathsf{open}\ \{x\}\ \mathsf{then}\ P\ x\ \mathsf{else}\ P\ \mathsf{at}\ x$$

For continuity, both limits are equivalent: $f \xrightarrow{x} f\ x$ iff $f \xrightarrow{x}{}' f\ x$. For at' we get a stronger variant of (16) as proof principle: No matter if x is discrete, the closed predicate P holds on x if P is a punctured neighbourhood of x.

$$\frac{\mathsf{closed}\ \{x \mid P\ x\} \qquad P\ \mathsf{at}'\ x}{P\ x} \tag{19}$$

When the convergence limit exists, we select it with definite description: $\mathsf{Lim}\ f\ x = \iota y.\ f \xrightarrow{x}{}' y$. As a ccpo topology is a T2 space, the limit is unique.

[3] The formalisation of convergence in Isabelle/HOL uses topological filters for the argument, as described in [11]. The punctured neighbourhoods at and at' are topological filters, but for a shorter presentation we avoid their introduction.

3.3 Constructing lfilter

As lazy lists are a ccpo, they also form a ccpo topology as described in §3.2. We first observe that the finite lists are dense in this topology, i.e. every lazy list is the limit of a sequence of finite lists. Moreover, a lazy list is discrete iff it is finite: $\mathsf{open}\ \{\overline{xs}\} \longleftrightarrow \mathsf{lfinite}\ \overline{xs}$. This yields a nice characterization of at' (20), from which we easily derive that at' behaves as expected on the constructor $\_ \cdot \_$ (21).

$$P\ \mathsf{at}'\ \overline{xs} \longleftrightarrow \exists \overline{ys} \in {\downarrow}\overline{xs}.\, \forall \overline{zs} \in {\downarrow}\overline{xs}.\, \overline{ys} \sqsubseteq \overline{zs} \longrightarrow P\ \overline{zs} \tag{20}$$

$$(\lambda \overline{ys}.\, x \cdot \overline{ys}) \xrightarrow{\overline{xs}}{}' \ x \cdot \overline{xs} \qquad \frac{(\lambda \overline{zs}.\, f\ (x \cdot \overline{zs})) \xrightarrow{\overline{xs}}{}'\ y}{f \xrightarrow{x \cdot \overline{xs}}{}'\ y} \tag{21}$$

Hence, at' behaves as expected on finite and infinite lists. Thus, we define $\mathsf{lfilter}\ P\ \overline{xs}$ as the limit of $\mathsf{filter}\ P$:

$$\mathsf{lfilter}\ P\ \overline{xs} = \mathsf{Lim}\ (\lambda \overline{ys}.\, \lceil \mathsf{filter}\ P\ \lfloor \overline{ys} \rfloor \rceil)\ \overline{xs} \tag{22}$$

Before proving lfilter's properties, we must prove that it continuously extends filter. Extension (23) shows that they coincide on finite lists. This follows from (18) and uniqueness of limits by unfolding the definitions of lfilter and Lim.

$$\mathsf{lfinite}\ \overline{xs} \longrightarrow \mathsf{lfilter}\ P\ \overline{xs} = \lceil \mathsf{filter}\ P\ \lfloor \overline{xs} \rfloor \rceil \tag{23}$$

Then, we show that lfilter is continuous everywhere (25). It suffices to show that the limit exists, as uniqueness of limits then ensures continuity. To that end, we prove the theorem (24): if a function f is monotone on all finite lazy lists, then it converges on $\overline{xs}$ to the lub of the image of $\overline{xs}$'s finite prefixes under f. This also completes the proof, as filter is monotone. Our initial definition (15) follows from these rules.

$$\frac{\forall \overline{ys}\ \overline{zs}.\, \overline{ys} \sqsubseteq \overline{zs} \wedge \mathsf{lfinite}\ \overline{zs} \longrightarrow f\ \overline{ys} \leq f\ \overline{xs}}{f \xrightarrow{\overline{xs}}{}' \bigvee (f\ `{\downarrow}\overline{xs})} \tag{24}$$

$$\mathsf{lfilter}\ P \xrightarrow{\overline{xs}}{}' \mathsf{lfilter}\ P\ \overline{xs} \tag{25}$$

3.4 Proving with Topology

In this section, we prove that the definition in (22) satisfies the specification (SPEC) and the properties from Fig. 1. In general, reasoning about lfilter first reduces the property on lazy lists to a property on finite lists. The characterisation of at' on lazy lists (20) yields the following proof principle. It is derived from (19) by taking $\overline{ys} = []$ as witness for the existential quantifier in (20).

$$\frac{\mathsf{closed}\ \{\overline{xs} \mid P\ \overline{xs}\} \qquad \forall \overline{zs} \in {\downarrow}\overline{xs}.\, P\ \overline{zs}}{P\ \overline{xs}} \tag{26}$$

This proof rule splits a goal $P\ \overline{xs}$ into two subgoals: (i) $\mathsf{closed}\ \{\overline{xs} \mid P\ \overline{xs}\}$ and (ii) $\forall \overline{zs} \in {\downarrow}\overline{xs}.\, P\ \overline{zs}$. Closedness is usually proved automatically in two steps. First, $P\ xs$ is decomposed into an atomic predicate and functions. These are

then shown closed and continuous using pre-proven theorems such as closedness of equality in a T2 space (§3.1) and continuity of lfilter (25). In subgoal (ii), $\downarrow\overline{xs}$ consists only of finite lists. Hence, we have indeed reduced the statement from arbitrary lazy lists to their finite subset. This goal is proved either by induction on lfinite $\overline{zs}$, or by rewriting with equations like (23) into functions over finite lists. For proving the specification (SPEC) and the properties (CONJ, LSET), this approach suffices. We also use it to show that our two definitions of lfilter from (§2.2) and (22) are equivalent.

Note that the second goal keeps the prefix relation between $\overline{zs}$ and $\overline{xs}$. Crucially, this maintains the relation of subgoal (ii) to further assumptions that are not part of the predicate P. When we prove (LDISTINCT), we operate only on the conclusion ldistinct (lfilter P $\overline{xs}$). Closedness (subgoal (i)) follows from ldistinct being closed and lfilter being continuous by preservation of closedness under composition with continuous functions. Subgoal (ii) is

$$\mathsf{ldistinct}\ \overline{xs} \longrightarrow \forall \overline{zs} \in \downarrow\overline{xs}.\ \mathsf{ldistinct}\ (\mathsf{lfilter}\ P\ \overline{zs}).$$

As prefixes of distinct lists are distinct, it suffices to show the following for all $\overline{zs}$.

$$\mathsf{ldistinct}\ \overline{zs} \longrightarrow \mathsf{lfinite}\ \overline{zs} \longrightarrow \mathsf{ldistinct}\ (\mathsf{lfilter}\ P\ \overline{zs})$$

Existing lemmas about filter and ldistinct suffice to show this, but induction on lfinite $\overline{zs}$ would work, too.

Property (NIL) is more complicated. The statement is not a closed predicate, so we cannot easily reduce it to finite lists. Instead we prove the direction from left to right using (LSET), and the converse using our approach from above.

4 Comparison

In this section, we compare our approaches least fixpoints (§2) and continuous extensions (§3) in five respects: the requirements on the codatatype and on the type of the function, the role of monotonicity, proof principles, and proof elegance.

Ccpo Structure on the Codatatype. Both approaches require a ccpo structure on the codatatype. As monotonicity is crucial for definitions and proofs (see below), functions of interest (and the constructors in particular) should be monotone. For lazy lists, the prefix order with [] as the least element is a natural choice. The extended naturals enat $= 0 \mid$ eSuc enat are a ccpo under the usual ordering $\leq$. Even terminated lazy lists given by (α, β) tllist $=$ TNil $\beta \mid$ TCons α $((\alpha, \beta)$ tllist$)$ form a useful ccpo under the prefix ordering extended with TNil b as least element for any user-specified, but fixed b. Yet, we have not found useful ccpos for codatatypes without finite values like infinite lists α stream $=$ Stream α (α stream).

Type Restrictions on Function Definitions. For recursive definitions, the two approaches pose different requirements on the function. Least fixpoints need the ccpo on the codomain whereas the domain can be arbitrary. Therefore, this

works for functions that produce a codatatype value such as iterate below. In contrast, they cannot handle functions that only consume a codatatype value such as lsum, which sums over a lazy list. Dually, continuous extensions require a ccpo topology on the domain whereas the codomain can be any T2 space. This works for functions that consume a codatatype value such as lsum, but this approach cannot define producers such as iterate.

$$\mathsf{iterate}\ f\ x = x \cdot \mathsf{iterate}\ f\ (f\ x)$$

$$\mathsf{sum}\ [] = 0 \qquad \mathsf{sum}\ (x \cdot \underline{xs}) = x + \mathsf{sum}\ \underline{xs} \qquad \mathsf{lsum}\ \overline{xs} = \mathsf{Lim}\ \mathsf{sum}\ \overline{xs}$$

Monotonicity. To derive the recursive specification from the definition, we have to show well-definedness for both approaches. For least fixpoints, the associated functional must be monotone, i.e. recursion may only occur in monotone contexts. For example, this approach cannot handle lmirror, because concatenation $+\!+$ is not monotone in its first argument, which contains the recursive call.

$$[] +\!+ \overline{ys} = \overline{ys} \qquad (x \cdot \overline{xs}) +\!+ \overline{ys} = x \cdot (\overline{xs} +\!+ \overline{ys}) \tag{27}$$

$$\mathsf{lmirror}\ [] = [] \qquad \mathsf{lmirror}\ (x \cdot \overline{xs}) = x \cdot (\mathsf{lmirror}\ \overline{xs} +\!+ [x]) \tag{28}$$

This shows how the choice of ccpo determines what functions can be defined. The **partial-function** package [14] automates the monotonicity proof and derives the recursive specification. Note that the defined function need not be monotone itself; we can e.g. define $+\!+$ as a least fixpoint for (27).

Continuous extensions need a different form of monotonicity. To derive the recursive equations of the continuous extension, we must show that the limit exists. By (24), it suffices to show that the function (not the functional) is monotone. Thus, we cannot define lmirror as a continuous extension, either. This time, the problem is not with $+\!+$, but rather lmirror, which is not monotone.

Another difference to least fixpoints is that the function need not be continuous at all points, as the continuous extension is defined pointwise. This is essential for functions like lsum that are well-defined only on a subset of its parameters such as the lists of positive real numbers extended with infinity.

Proof Principles. The main advantage of our approaches over ad hoc constructions like in [16] is that they bring their own proof principles: fixpoint induction (4) and structural induction (5), convergence on a closed set (19). They all require admissibility of the induction statement, since $\mathsf{closed}\ \{x \mid P\ x\}$ in a ccpo topology coincides with $\mathsf{adm}\ (\bigvee, \leq)\ P$—just unfold the definition of open sets (17) to see this. The two notions of continuity are closely related, too. Monotonicity and order continuity imply convergence in the ccpo topology. The converse does not hold; this reflects difference between the point-wise flavour of continuous extensions and the function-as-a-whole style of least fixpoints.

Convergence on a closed set (26) and structural induction on lazy lists (5) take the consumer view, i.e. they only work for functions that consume a codatatype value. Interestingly, the former generalises the latter. Convergence keeps the bound $\overline{zs} \in \downarrow\overline{xs}$. In comparison, structural induction relaxes the bound $\overline{zs} \in \downarrow\overline{xs}$

to lfinite $\overline{zs}$ and inlines the induction on lfinite $\overline{zs}$. More abstractly, (5) reduces the statement directly to an induction on the finite subset of lazy lists. In contrast, the topological approach translates it to a corresponding statement on the type of finite lists by rewriting with identites such as (23)—the latter is then typically shown by induction.

Fixpoint induction has no counterpart in continuous extensions, as it is a proof principle for producers. It is harder to use than induction on lazy lists, see §2.4 for examples. In particular, fixpoint induction cannot show non-trivial lower bounds. However, it allows to prove properties such as (LREL) where the other principles fail. In fact, we have not yet been able to prove (LREL) by topological means, as we are not yet able to handle general predicates over two variables.

Proof Elegance. As a rough measure of proof elegance, we take the size of proofs for the five properties in Fig. 1. In the fixpoint approach, they all consist of just two steps: (i) the induction method generates the admissibility condition and the inductive cases, and (ii) an automatic proof method solves them immediately. Similarly, the topological approach first applies the proof principle (19) and then solves the subgoals. The level of automation is similar, except when we have to show the statement on finite lists by induction, which does not happen automatically. In summary, proving (CONJ, LSET, NIL, LDISTINCT) takes between 2 and 5 steps with an average of 2.75. For comparison, the former ad hoc construction of lfilter in [16] requires for proving the properties in Fig. 1 between 2 and 35 steps each with an average of 13—not even counting any of the auxiliary lemmas such as (30).

5 Related Work

Functions on Codatatypes. Devillers et al. [8] compare different formalisations of lazy lists that were available in 1997. They note the general difficulty of defining lfilter and lconcat—given by (29)—and proving their properties.

$$\mathsf{lconcat}\ [] = [] \qquad \mathsf{lconcat}\ (\overline{xs} \cdot \overline{xss}) = \overline{xs} +\!\!+ \mathsf{lconcat}\ \overline{xss} \tag{29}$$

In [20], Paulson describes the construction of codatatypes in Isabelle and the primitively corecursive definition of the well-known functions lmap and ++ with coinduction as proof principle for equality. He notes that he did not know of a natural formalisation for lconcat in HOL. Later, he defined lfilter using an inductive search predicate (file `LFilter.thy` distributed with Isabelle until 2009). Thus, all proofs about lfilter need corresponding lemmas about the search predicate. For example, his 72-line proof of (CONJ) needs seven auxiliary lemmas. For comparison, ours is one line—our preparations are not negligible, but we reuse monotonicity and continuity in many lemmas. In Coq, Bertot [4] relies on a similar search predicate; he transforms non-local properties like sortedness into local ones to simplify proofs.

Matthews [18] presents a framework to define corecursive functions via contractions for converging equivalence relations (CER) over a well-founded relation, Gianantonio and Miculan generalise CERs to complete ordered families of equiv-

alences (COFE) [10]. CERs and COFEs require uniqueness of the specification and therefore yield a proof principle for equality. To prove contraction for lfilter, Matthews needs an inductive search predicate similar to Paulson's, and a search function that returns the first index of an element satisfying P.

Charguéraud [7] formalised the optimal fixpoint (OFP) combinator in Coq. It allows to define a large class of recursive functions, but it cannot pick any particular solution if the specification is not unique. This is arguably closer to the specification, but it complicates proofs: for the OFP of (SPEC), e.g. (LSET) holds only if $\overline{xs}$ is finite or P holds for infinitely many elements of $\overline{xs}$. For proof principles, he relies on a generalisation of COFEs, as the OFP does not provide any.

The Coinductive library [16], developed by the first author, includes functions on lazy lists and lemmas about them. The approach in this paper simplifies the definitions of and proofs about lfilter and similar functions. Previously, their definition was rather involved; lfilter was defined as the corecursive unfolding of ldropWhile; ldropWhile depended on ltakeWhile, llength, and ldrop; and ldrop on further functions. The auxiliary functions have some value of their own, so the overhead was limited. Yet, the theorems about lfilter (like those in Fig. 1) needed other theorems about the auxiliary functions. Thus, definitions and proofs both lacked elegance. The proof of (CONJ) e.g. required the specialised lemma (30).

$$\mathsf{lhd}\ (\mathsf{ldropWhile}\ P\ (\mathsf{lfilter}\ Q\ \overline{xs})) = \mathsf{lhd}\ (\mathsf{ldropWhile}\ (\lambda x.\ P\ x \vee \neg Q\ x)\ \overline{xs}) \quad (30)$$

Domain-Theoretic Approaches. Formalisations of domain theory and Scott's logic of computable functions (LCF) exist in HOL [1], Coq [3], and Isabelle/HOL [12]. They provide facilities to define domains and (non-terminating) recursive functions as least fixpoints as well as sophisticated proof automation. They support embedding of ordinary functions into LCF, but not the converse.

Although domains and codatatypes both contain infinite values, they are different, as all domains contain the value "undefined". Coinductive lists e.g. either end with [] or are infinite. In contrast, LCF lists can also end with undefined, e.g. filtering an infinite list whose elements all violate the predicate returns "undefined" instead of []. Thus, coinductive lists are almost isomorphic to infinite streams in HOLCF, except that the domain package additionally requires that the element type α forms a ccpo, too.

Undefinedness plays a central role in LCF: it conceptually represents all values, as monotonicity and continuity permit replacing undefined with a more specific value. This is sensible in modelling functional programs, but also complicates the theorem statements and their proofs (see e.g. [6]). Being based on HOL, our approach need not treat [] specially and can therefore deal with non-continuous functions, too. Our choice of topology reflects this, too. In our ccpo topology, finite values x are discrete, i.e., open $\{x\}$. In contrast, all Scott-open sets S are upward closed, i.e. if S contains x, then S contains all elements greater than x, too. Hence, our topology is finer than the Scott topology, so more functions are continuous, e.g. lsum on lists with a finite number of negative elements.

Two works have applied basic domain theory for defining recursive functions in HOL. First, Agerholm [2] suggested to define arbitrary recursive function as

the least fixpoint in a domain by lifting the function's codomain; when termination has been shown, his tool then casts the function back to plain HOL. Hence, our application with infinite recursion is out of scope. Second, Krauss [14] realised that a tail-recursive or monadic function can be defined as a least fixpoint, because its syntactic structure ensures monotonicity. He formalised the relevant concepts in Isabelle and implemented the **partial-function** package. To our knowledge, this has only been used for the option and state-exception monads. We re-use and extend his work to define non-monadic functions on codatatypes.

Topology for domain theory. We do not know of any formalisation that defines recursive functions using topology except for Lester [15]. He formalises the Scott topology of a directed complete partial order in PVS and uses it to prove the existence of the fixpoint operator. Friedrich [9] formalises the Scott topology in Isabelle/HOL to characterise liveness and security properties topologically.

6 Beyond lfilter and lazy lists

We have described how to use domain theory and topology to define recursive functions on codatatypes. The presentation has focused on the function lfilter, as it illustrates the main ideas well and allows us to compare the approaches. But they are not restricted to it. We have used them with the same ccpo to define lconcat (29), ldropWhile (31), and ldrop (32) and to prove numerous lemmas. These functions pose the same challenge of unbounded, unproductive recursion as lfilter. In addition, the definition of lconcat relies on $+\!\!+$ being monotone (and continuous in the topological approach) in the second argument, which contains the recursive call, and ldrop shows that we handle multiple parameters.

$$\begin{aligned} &\mathsf{ldropWhile}\ P\ [] &&= [] \\ &\mathsf{ldropWhile}\ P\ (x \cdot \overline{xs}) &&= (\mathsf{if}\ P\ x\ \mathsf{then}\ \mathsf{ldropWhile}\ P\ \overline{xs}\ \mathsf{else}\ x \cdot \overline{xs}) \end{aligned} \tag{31}$$

$$\mathsf{ldrop}\ 0\ \overline{xs} = \overline{xs} \qquad \mathsf{ldrop}\ n\ [] = [] \qquad \mathsf{ldrop}\ (\mathsf{eSuc}\ n)\ (x \cdot \overline{xs}) = \mathsf{ldrop}\ n\ \overline{xs} \tag{32}$$

In terms of automating the definitions and proofs, we have used only standard Isabelle tools so far. Hence, we have not yet reached the level of sophisticated packages such as HOLCF [12]. Indeed, our approaches offer more flexibility, as they use the full function space and allow non-continuous functions to some extent. Better automation of the function definitions is left as future work.

It is not yet clear which codatatypes can be turned into useful ccpos. Clearly, it should be possible for codatatypes with finite truncations, i.e. whenever there is a non-recursive constructor. Then, this constructor can cut off a possibly infinite subtree and thus serve as bottom element. Possibly-infinite lists (α llist and (α, β) tllist) and binary trees (α tree = Leaf | Node α (α tree) (α tree)) fall in this class. Conversely, if the codatatype contains only infinite values, e.g. infinite lists (α stream), a general approach seems impossible. Codatatypes with nested recursion such as α rtree = Tree α (α rtree llist) will be more challenging. Working out the precise boundaries of the approach is left as future work. We hope that such insights will lead to automated constructions of ccpos for codatatypes.

Acknowledgements. J.C. Blanchette, J. Breitner, O. Maric, D. Traytel, and the anonymous reviewers suggested many textual improvements. A. Popescu helped generalising our topology on lazy lists to ccpos. Hölzl is supported by DFG grant Ni 491/15-1.

References

1. Agerholm, S.: LCF examples in HOL. In: Higher Order Logic Theorem Proving and Its Applications. LNCS, vol. 859, pp. 1–16. Springer (1994)
2. Agerholm, S.: Non-primitive recursive function definitions. In: Higher Order Logic Theorem Proving and Its Applications. LNCS, vol. 971, pp. 17–31. Springer (1995)
3. Benton, N., Kennedy, A., Varming, C.: Some domain theory and denotational semantics in Coq. In: TPHOLs'09. LNCS, vol. 5674, pp. 115–130. Springer (2009)
4. Bertot, Y.: Filters on coinductive streams, an application to Eratosthenes' sieve. In: TLCA 2005. LNCS, vol. 3461, pp. 102–115. Springer (2005)
5. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: ITP 2014. LNCS. Springer (2014)
6. Breitner, J., Huffman, B., Mitchell, N., Sternagel, C.: Certified HLints with Isabelle/HOLCF-Prelude. In: Haskell and Rewriting Techniques (HART). (2013)
7. Charguéraud, A.: The optimal fixed point combinator. In: ITP 2010. LNCS, vol. 6172, pp. 195–210. Springer (2010)
8. Devillers, M., Griffioen, D., Müller, O.: Possibly infinite sequences in theorem provers: A comparative study. In: TPHOLs 1997. LNCS, vol. 1275, pp. 89–104 (1997)
9. Friedrich, S.: Topology. Archive of Formal Proofs (2004) `http://afp.sf.net/entries/Topology.shtml`, Formal proof development.
10. Gianantonio, P., Miculan, M.: A unifying approach to recursive and co-recursive definitions. In: TYPES 2002. LNCS, vol. 2646, pp. 148–161. Springer (2003)
11. Hölzl, J., Immler, F., Huffman, B.: Type classes and filters for mathematical analysis in Isabelle/HOL. In: ITP'13. LNCS, vol. 7998, pp. 279–294. Springer (2013)
12. Huffman, B.C.: HOLCF'11: A Definitional Domain Theory for Verifying Functional Programs. PhD thesis, Portland State University (2012)
13. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning 44(4), 303–336 (2010)
14. Krauss, A.: Recursive definitions of monadic functions. In: PAR 2010. EPTCS, vol. 43, pp. 1–13 (2010)
15. Lester, D.R.: Topology in PVS: continuous mathematics with applications. In: AFM 2007, pp. 11–20. ACM (2007)
16. Lochbihler, A.: Coinductive. Archive of Formal Proofs (2010) `http://afp.sf.net/entries/Coinductive.shtml`, Formal proof development.
17. Lochbihler, A.: Making the Java memory model safe. ACM Trans. Program. Lang. Syst. 35(4), 12:1–65 (2014)
18. Matthews, J.: Recursive function definition over coinductive types. In: TPHOLs 1999. LNCS, vol. 1690, pp. 73–90. Springer (1999)
19. Müller, O., Nipkow, T., Oheimb, D.v., Slotosch, O.: HOLCF = HOL + LCF. J. Funct. Program. 9, 191–223 (1999)
20. Paulson, L.C.: Mechanizing coinduction and corecursion in higher-order logic. J. Logic Comput. 7(2), 175–204 (1997)
21. Slind, K.: Function definition in higher-order logic. In: TPHOLs 1996. LNCS, vol. 1125, pp. 381–397. Springer (1996)