Fundamenta Informaticae XX (2005) 1–14
IOS Press

Proof Methods for Corecursive Programs

Jeremy Gibbons
Oxford University Computing Laboratory, UK

Graham Hutton
School of Computer Science and IT, University of Nottingham, UK

Abstract. Recursion is a well-known and powerful programming technique, with a wide variety of applications. The dual technique of corecursion is less well-known, but is increasingly proving to be just as useful. This article is a tutorial on the four main methods for proving properties of corecursive programs: fixpoint induction, the approximation (or take) lemma, coinduction, and fusion.

1. Introduction

Recursion is a central concept in computing, with applications ranging from the theoretical foundations of computation [34] to practical programming techniques [8]. In recent years, it has become increasingly clear that the dual but less well-known concept of *corecursion* is just as useful [1, 4, 20, 27].

Following the work of Moss and Danner [28] on the foundations of corecursion, we use the term *corecursive program* for a function whose range is a type defined recursively as the greatest solution of some equation. Dually, we use the term *recursive program* for a function whose domain is type defined recursively as the least solution of some equation. These definitions are rather general — in particular, they require neither self-reference, nor patterns of definition that ensure properties such as productivity or termination — but they will suffice for our expository purposes here.

As an example, if types are modelled as sets, then a type of infinite lists of integers can be defined as the greatest set X for which there is a bijection $X \;\cong\; \mathbb{Z}\times X$, and hence any function that produces such an infinite list is (according to our definition) a corecursive program. Similarly, a type of finite lists can be defined as the least set X for which $X \;\cong\; 1+(\mathbb{Z}\times X)$, where 1 is a singleton set and $+$ is disjoint union of sets, and hence any function that consumes such a finite list is a recursive program.

For programming examples we use Haskell [32], a pure functional language with non-strict semantics. Purity (or referential transparency) and non-strictness together permit substitution of equals for equals, and hence proofs by simple equational reasoning. In addition, a semantic basis in terms of complete partial orders has the convenient property that there is no distinction between least and greatest solutions to type equations, as these notions coincide [12, 39]. For example, in this setting the equation $X \;\cong\; 1+(\mathbb{Z}\times X)$ has a unique solution for X, given by the type of finite, partial (undefined after a certain point) and infinite lists of integers. Hence, in this article recursive programs and corecursive programs are simply functional programs that have recursively-defined types as their domain and range, respectively.

Historically, the basic method for proving properties of corecursive programs is *fixpoint induction* [2], which is derived from the domain-theoretic approach to programming language semantics. Applying fixpoint induction is rather tedious, but for many applications we can use the higher-level *approximation lemma* [5], a recent improvement of the well-known *take lemma* [7]. Alternatively, we can reason directly in terms of the structure of programs themselves and use *coinduction* [18]. However, the use of inductive or coinductive methods can often be avoided altogether by using *fusion* [25], an algebraic law derived from the basic pattern of corecursive definition.

This article is a tutorial on the above methods for proving properties of corecursive programs. Each method is presented, proved to be correct, and illustrated with an example. We conclude with a comparison of the four methods and references to further reading. For simplicity, we restrict our attention to corecursive programs that produce lists, but none of the four methods are specific to this type. The reader is assumed to be familiar with the basics of recursive programming and proof (for example, see [5]), but no prior knowledge of corecursive programming and proof is assumed.

2. The map-iterate property

Consider a recursive type of lists in which the empty list is denoted by $[\,]$, and non-empty lists are constructed using an infix operator $(:)$ that prepends a value to a list. For example, $0 : 1 : 2 : [\,]$ is a finite list, while the equation $\mathit{ones} = 1 : \mathit{ones}$ defines the infinite list $\mathit{ones} = 1 : 1 : 1 : \cdots$.

A standard corecursive function for lists is *iterate* f, which produces an infinite list by successively applying a function f to a seed value, and is defined by the following equation:

$$\mathit{iterate}\ f\ x \quad = \quad x : \mathit{iterate}\ f\ (f\ x)$$

(For simplicity, we avoid explicitly specifying types in this article, but they can easily be inferred from the definitions if required.) Unwinding this definition a few steps, we see that:

$$\mathit{iterate}\ f\ x \quad = \quad x : f\ x : f\ (f\ x) : f\ (f\ (f\ x)) : \cdots$$

For example, if *inc* is the increment function on natural numbers, then $\mathit{nats} = \mathit{iterate}\ \mathit{inc}\ 0$ defines the infinite list $\mathit{nats} = 0 : 1 : 2 : \cdots$. Another standard corecursive (and also recursive) function is *map* f, which produces a list by applying a function f to each value in a list, and is defined as follows:

$$\begin{array}{lcl} \mathit{map}\ f\ [\,] & = & [\,] \\ \mathit{map}\ f\ (x : \mathit{xs}) & = & f\ x : \mathit{map}\ f\ \mathit{xs} \end{array}$$

Unwinding this definition a few steps shows that:

$$\mathit{map}\ f\ (x_0 : x_1 : x_2 : x_3 : \cdots) \quad = \quad f\ x_0 : f\ x_1 : f\ x_2 : f\ x_3 : \cdots$$

For example, $\mathit{map}\ \mathit{inc}\ \mathit{nats}$ produces the infinite list $1 : 2 : 3 : \cdots$. This same list can also be produced by the expression $\mathit{iterate}\ \mathit{inc}\ (\mathit{inc}\ 0)$. Whereas the former expression increments each number in the infinite list of naturals, the latter successively applies the increment function starting with the number one. Generalising from this example yields the *map-iterate property* [7]:

$$\mathit{map}\ f\ (\mathit{iterate}\ f\ x) \ = \ \mathit{iterate}\ f\ (f\ x)$$

This equation states that iterating a function and then mapping it gives the same result as applying the function and then iterating it, namely an infinite list of the form:

$$f\ x : f\ (f\ x) : f\ (f\ (f\ x)) : f\ (f\ (f\ (f\ x))) : \cdots$$

But how can the *map-iterate* property be proved? Note that the standard method of structural induction on lists is not applicable, because there is no list argument over which induction can be performed. In the remainder of the article we review and compare the four main methods that *can* be used to prove properties of corecursive functions, using the *map-iterate* property as our running example.

3. Fixpoint induction

Fixpoint induction is derived from the domain-theoretic approach to programming language semantics [38]. The basic idea in this approach is that types are *complete partial orders* (cpos), that is, sets with a partial-ordering $\sqsubseteq$, a least element $\perp$, and limits of all non-empty chains. In turn, programs are *continuous functions*, that is, functions between cpos that preserve the partial-order and limit structure.

Now consider an equation $x = f\,x$ that defines a value x in terms of itself and some continuous function f. A well-known fixpoint theorem [38] states that this equation has a least solution for x, denoted by $\mathit{fix}\ f$ and called the *least fixpoint* of f, which is adopted as the semantics of the definition. Moreover, $\mathit{fix}\ f$ is constructed as the limit of the following infinite chain:

$$\perp\ \sqsubseteq\ f\perp\ \sqsubseteq\ f\,(f\perp)\ \sqsubseteq\ f\,(f\,(f\perp))\ \sqsubseteq\ \cdots$$

As a simple example of this approach, consider again the equation $\mathit{ones} = 1 : \mathit{ones}$ that defines the infinite list $1:1:1:\cdots$. This definition can be rewritten as $\mathit{ones} = f\ \mathit{ones}$, where f is the function defined by $f\ \mathit{xs} = 1 : \mathit{xs}$. (Verifying that a function such as f is continuous is normally just a matter of appealing to the fact that any function definable in a programming language is necessarily continuous [38].) Hence, the semantics of the definition is given by $\mathit{ones} = \mathit{fix}\ f$, and by the fixpoint theorem is constructed as the limit of the infinite chain of partial lists containing increasing numbers of 1s:

$$\perp\ \sqsubseteq\ 1:\perp\ \sqsubseteq\ 1:1:\perp\ \sqsubseteq\ 1:1:1:\perp\ \sqsubseteq\ \cdots$$

The basic method for proving properties of programs defined using *fix* is Scott and de Bakker's *fixpoint induction* [2]. Suppose that f is a continuous function on a cpo and that P is a *chain-complete* predicate on the same cpo, that is, whenever P holds of all elements in a chain then it also holds of the limit. Then fixpoint induction is given by the following inference rule:

$$\boxed{\dfrac{P\perp \qquad \forall x.\, P\,x \Rightarrow P\,(f\ x)}{P\ (\mathit{fix}\ f)}}$$

This rule states that if the predicate holds of the least element $\perp$ of the cpo, and whenever it holds of an element x in the cpo then it also holds for $f\,x$, then the predicate also holds for $\mathit{fix}\ f$. Fixpoint induction can be verified by the following simple calculation, in which the limit operator on chains is denoted by $\bigsqcup$ and the n-fold repeated application of a function f is denoted by f^n:

$$
\begin{array}{ll}
 & P\ (\mathit{fix}\ f) \\
\Leftrightarrow & \quad \{\ \text{definition of } \mathit{fix}\ f\ \} \\
 & P\ (\bigsqcup_n \{f^n \perp\}) \\
\Leftarrow & \quad \{\ P \text{ is chain-complete}\ \} \\
 & \forall n.\ P\ (f^n \perp) \\
\Leftarrow & \quad \{\ \text{induction on } n\ \} \\
 & P\ (f^0 \perp)\ \wedge\ \forall n.\ P\ (f^n \perp) \Rightarrow P\ (f^{n+1} \perp) \\
\Leftrightarrow & \quad \{\ \text{definition of } f^n\ \} \\
 & P \perp\ \wedge\ \forall n.\ P\ (f^n \perp) \Rightarrow P\ (f\ (f^n \perp)) \\
\Leftarrow & \quad \{\ \text{generalising } f^n \perp \text{ to } x\ \} \\
 & P \perp\ \wedge\ \forall x.\ P\ x \Rightarrow P\ (f\ x)
\end{array}
$$

As an application of fixpoint induction, let us see how it can be used to prove the *map*-*iterate* property from the previous section: $\mathit{map}\ f\ (\mathit{iterate}\ f\ x)\ =\ \mathit{iterate}\ f\ (f\ x)$. First of all, we abstract from the use of *iterate* and define a predicate P on functions by the following equivalence:

$$P\ g\ \Leftrightarrow\ \forall f, x.\ \mathit{map}\ f\ (g\ f\ x)\ =\ g\ f\ (f\ x)$$

Verifying that a predicate is chain-complete is normally just a matter of appealing to standard recipes for constructing such predicates [38]. For example, chain-completeness of P follows from the fact that any equation between continuous functions is chain-complete, which is easy to verify.

Using the above predicate, the *map*-*iterate* property can be written as $P\ \mathit{iterate}$. In turn, the semantics of the function *iterate* is given by $\mathit{iterate} = \mathit{fix}\ h$, where h is the continuous function defined by $h\ g\ f\ x = x : g\ f\ (f\ x)$. Hence, the *map*-*iterate* property can now be written as $P\ (\mathit{fix}\ h)$, which by fixpoint induction follows from the assumptions $P \perp$ and $\forall g.\ P\ g \Rightarrow P\ (h\ g)$, which are verified as follows (the hint "substitutivity" refers to the fact that functions give equal results for equal arguments):

$$
\begin{array}{ll}
 & P \perp \\
\Leftrightarrow & \quad \{\ \text{definition of } P\ \} \\
 & \forall f, x.\ \mathit{map}\ f\ (\perp f\ x)\ =\ \perp f\ (f\ x) \\
\Leftrightarrow & \quad \{\ \text{definition of } \perp\ \} \\
 & \forall f, x.\ \mathit{map}\ f \perp\ =\ \perp \\
\Leftrightarrow & \quad \{\ \mathit{map}\ f \text{ is strict}\ \} \\
 & \mathit{true}
\end{array}
$$

and

$$
\begin{array}{ll}
 & P\ (h\ g) \\
\Leftrightarrow & \quad \{ \text{definition of } P \} \\
 & \forall f, x.\ map\ f\ (h\ g\ f\ x)\ =\ h\ g\ f\ (f\ x) \\
\Leftrightarrow & \quad \{ \text{definition of } h \} \\
 & \forall f, x.\ map\ f\ (x : g\ f\ (f\ x))\ =\ f\ x : g\ f\ (f\ (f\ x)) \\
\Leftrightarrow & \quad \{ \text{definition of } map \} \\
 & \forall f, x.\ f\ x : map\ f\ (g\ f\ (f\ x)))\ =\ f\ x : g\ f\ (f\ (f\ x)) \\
\Leftarrow & \quad \{ \text{substitutivity} \} \\
 & \forall f, x.\ map\ f\ (g\ f\ (f\ x)))\ =\ g\ f\ (f\ (f\ x)) \\
\Leftarrow & \quad \{ \text{generalising } f\ x \text{ to } y \} \\
 & \forall f, y.\ map\ f\ (g\ f\ y)\ =\ g\ f\ (f\ y) \\
\Leftrightarrow & \quad \{ \text{definition of } P \} \\
 & P\ g
\end{array}
$$

Note that by virtue of being an implication rather than an equivalence, fixpoint induction provides a sufficient condition for establishing a certain form of property, but not a necessary one, and hence may not always be applicable. For example, if P is the chain-complete predicate "is an infinite list" and f is the continuous function $f\ xs = 1 : xs$, then $P\ (\mathit{fix}\ f)$ expresses the true statement that $1 : 1 : 1 : \cdots$ is an infinite list, but this true statement cannot be proved using fixpoint induction because $\perp$ is not an infinite list and hence the base case $P\ \perp$ is false. However, for such examples one can always resort to reasoning explicitly using the definition of $\mathit{fix}\ f$ as the limit of an infinite chain of approximations.

Fixpoint induction is not specific to the type of lists, but is independent of the details of the underlying type, requiring only the fact that the type forms a cpo. However, it is clear that fixpoint induction is a rather low-level proof method. In particular, it is tedious to have to return to first principles and perform proofs at the level of the fixpoint semantics of programs. It is also important to note that proofs using fixpoint induction require careful consideration of the underlying cpos and their properties, particularly when reasoning in the presence of partial and infinite values [10].

4. Approximation lemma

A higher-level method for proving properties of corecursive programs is the *approximation lemma* [5], a recent improvement of the well-known take lemma [7]. Recall the standard function *take n*, which returns the first *n* elements of a list, and is defined as follows:

$$
\begin{array}{lll}
take\ 0\ \quad xs & = & [\,] \\
take\ (n+1)\ [\,] & = & [\,] \\
take\ (n+1)\ (x : xs) & = & x : take\ n\ xs
\end{array}
$$

For example, *take* 3 *ones* returns the finite list $1 : 1 : 1 : [\,]$. The approximation lemma is based upon a function *approx n* defined in the same way as *take n*, except that the case for $n = 0$ is removed:

$$
\begin{array}{lll}
approx\ (n+1)\ [\,] & = & [\,] \\
approx\ (n+1)\ (x : xs) & = & x : approx\ n\ xs
\end{array}
$$

Because $(n+1)$ patterns only match strictly positive integers, removing the $n = 0$ case means that, by case exhaustion, *approx* 0 $xs = \bot$ for all lists *xs*. For example, *approx* 3 *ones* returns the partial list $1 : 1 : 1 : \bot$. The approximation lemma itself is given by the following equivalence:

$$\boxed{xs \;=\; ys \quad\Leftrightarrow\quad \forall n.\; \mathit{approx}\; n\; xs \;=\; \mathit{approx}\; n\; ys}$$

This equivalence states that two lists are equal precisely when all their approximations are equal. The left-to-right direction is trivially true by substitutivity. For the other direction, it is easy to show that

$$\mathit{approx}\; 0 \;\sqsubseteq\; \mathit{approx}\; 1 \;\sqsubseteq\; \mathit{approx}\; 2 \;\sqsubseteq\; \mathit{approx}\; 3 \;\sqsubseteq\; \cdots$$

is a chain that has the identity function *id* on lists as its limit (by induction on natural numbers and lists, respectively), using which result the right-to-left direction can be verified as follows:

$$\begin{array}{ll}
 & xs \;=\; ys \\
\Leftrightarrow & \quad \{ \text{ definition of } \mathit{id} \text{ } \} \\
 & \mathit{id}\; xs \;=\; \mathit{id}\; ys \\
\Leftrightarrow & \quad \{ \text{ above result } \} \\
 & (\bigsqcup_n \{\mathit{approx}\; n\})\; xs \;=\; (\bigsqcup_n \{\mathit{approx}\; n\})\; ys \\
\Leftrightarrow & \quad \{ \text{ continuity of application } \} \\
 & \bigsqcup_n \{\mathit{approx}\; n\; xs\} \;=\; \bigsqcup_n \{\mathit{approx}\; n\; ys\} \\
\Leftarrow & \quad \{ \text{ substitutivity } \} \\
 & \forall n.\; \mathit{approx}\; n\; xs \;=\; \mathit{approx}\; n\; ys
\end{array}$$

The utility of the approximation lemma is that it allows us to prove two lists equal using the simple technique of induction on natural numbers. For example, by the approximation lemma the *map-iterate* property is equivalent to the following property:

$$\forall n.\; \mathit{approx}\; n\; (\mathit{map}\; f\; (\mathit{iterate}\; f\; x)) \;=\; \mathit{approx}\; n\; (\mathit{iterate}\; f\; (f\; x))$$

This property can now be verified by induction on *n*. The base case $n = 0$ is trivially true because *approx* 0 $xs = \bot$ for all lists *xs*, while the inductive case $n = m + 1$ is verified as follows:

$$
\begin{array}{ll}
& \mathit{approx}\ (m+1)\ (\mathit{map}\ f\ (\mathit{iterate}\ f\ x)) \\
= & \quad \{\ \text{definition of } \mathit{iterate}\ \} \\
& \mathit{approx}\ (m+1)\ (\mathit{map}\ f\ (x : \mathit{iterate}\ f\ (f\ x))) \\
= & \quad \{\ \text{definition of } \mathit{map}\ \} \\
& \mathit{approx}\ (m+1)\ (f\ x : \mathit{map}\ f\ (\mathit{iterate}\ f\ (f\ x))) \\
= & \quad \{\ \text{definition of } \mathit{approx}\ \} \\
& f\ x :\ \mathit{approx}\ m\ (\mathit{map}\ f\ (\mathit{iterate}\ f\ (f\ x))) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& f\ x : \mathit{approx}\ m\ (\mathit{iterate}\ f\ (f\ (f\ x))) \\
= & \quad \{\ \text{definition of } \mathit{approx}\ \} \\
& \mathit{approx}\ (m+1)\ (f\ x : \mathit{iterate}\ f\ (f\ (f\ x))) \\
= & \quad \{\ \text{definition of } \mathit{iterate}\ \} \\
& \mathit{approx}\ (m+1)\ (\mathit{iterate}\ f\ (f\ x))
\end{array}
$$

Unlike fixpoint induction, the approximation lemma is an equivalence and hence provides a necessary and sufficient condition, although only for the special (but very common) case of an equality between two lists. For such cases, using the approximation lemma has the advantage that proofs are performed at the level of the syntax of programs, without reference to their underlying fixpoint semantics.

Replacing the use *approx* in the approximation lemma by *take* gives the take lemma, which was popularised by Bird and Wadler's textbook on functional programming [7]:

$$xs\ =\ ys \quad \Leftrightarrow \quad \forall n.\ \mathit{take}\ n\ xs\ =\ \mathit{take}\ n\ ys$$

The take lemma can be used to prove the same properties as the approximation lemma, but the latter is simpler to prove and to apply. More importantly, however, the approximation lemma naturally generalises from lists to a large class of types (all polynomial types, which generalise the sum-of-product types supported by most functional languages), whereas the take lemma does not [21].

5. Coinduction

Another high-level method for proving properties of corecursive programs is *coinduction* [18]. The principle of coinduction is based upon the general notion of a *bisimulation* [22], which in the context of this article is a relation R on lists that has the following property:

$$
xs\ R\ ys \quad \Rightarrow \quad \left\{
\begin{array}{ll}
& xs = ys = \bot \\
\vee & \\
& xs = ys = [\,] \\
\vee & \\
& \exists v, vs, ws.\ xs = v : vs\ \wedge\ ys = v : ws\ \wedge\ vs\ R\ ws
\end{array}
\right.
$$

This property states that two lists that are related by a bisimulation are either both undefined, both empty, or both non-empty with heads (first elements) that are equal and tails (remaining lists of elements) that are themselves related by the bisimulation. Two lists *xs* and *ys* are called *bisimilar*, written $xs \sim ys$, if they are related by such a bisimulation. That is, we have the following definition:

$$xs \ \sim\ ys \quad \Leftrightarrow \quad \exists R.\ R \text{ is a bisimulation} \wedge xs\ R\ ys$$

Coinduction itself is given by the following equivalence:

$$\boxed{xs \ =\ ys \quad \Leftrightarrow \quad xs \ \sim\ ys}$$

This equivalence states that two lists are equal precisely when they are bisimilar. The left-to-right direction is trivially true, because the equality relation on lists is a bisimulation, as is easily verified. Conversely, by the approximation lemma the right-to-left direction is equivalent to:

$$xs \ \sim\ ys \quad \Rightarrow \quad \forall n.\ approx\ n\ xs \ =\ approx\ n\ ys$$

In turn, by making the implicit quantification over *xs* and *ys* explicit, and moving the quantification over *n* to the outermost level, the above implication is equivalent to:

$$\forall n.\ (\forall xs, ys.\ xs \ \sim\ ys \quad \Rightarrow \quad approx\ n\ xs = approx\ n\ ys)$$

This property can now be verified by induction on the natural number *n*. (The rearrangement of quantifiers is necessary to strengthen the induction hypothesis for this proof.) The base case $n = 0$ is trivially true, because $approx\ 0\ xs = \bot$ for all lists *xs*. For the inductive case $n = m + 1$ there are three cases to consider, derived from the premise $xs \sim ys$. The first two cases, $xs = ys = \bot$ and $xs = ys = [\,]$, are trivially true because $approx\ (m+1)\ \bot = \bot$ and $approx\ (m+1)\ [\,] = [\,]$ for all natural numbers *m*. The third case, $xs = v : vs$ and $ys = v : ws$ with $vs \sim ws$, is verified as follows:

$$\begin{array}{ll}
 & approx\ (m+1)\ (v : vs) \\
= & \quad \{ \text{ definition of } approx \ \} \\
 & v\ :\ approx\ m\ vs \\
= & \quad \{ \text{ induction hypothesis with } vs \sim ws \ \} \\
 & v\ :\ approx\ m\ ws \\
= & \quad \{ \text{ definition of } approx \ \} \\
 & approx\ (m+1)\ (v : ws)
\end{array}$$

The utility of coinduction is that it reduces the problem of proving that two lists are equal to the problem of finding a bisimulation that relates the two lists. For example, by coinduction the *map-iterate* property is equivalent to finding a bisimulation *R* that relates $map\ f\ (iterate\ f\ x)$ and $iterate\ f\ (f\ x)$. The latter condition is easily satisfied by defining the relation *R* as follows:

$$R \quad = \quad \{(map\ f\ (iterate\ f\ x), iterate\ f\ (f\ x)) \ \mid\ f, x \text{ of appropriate types}\}$$

To verify that *R* is a bisimulation, suppose that $xs\ R\ ys$, which means that $xs = map\ f\ (iterate\ f\ x)$ and $ys = iterate\ f\ (f\ x)$ for some *f* and *x*. Unfolding these expressions using the definitions for *iterate* and *map*, we see that $xs = f\ x\ :\ map\ f\ (iterate\ f\ (f\ x))$ and $ys = f\ x\ :\ iterate\ f\ (f\ (f\ x))$. Because both resulting expressions have the same head ($f\ x$), and their tails are related by R (with $f\ x$ as the seed value rather than x), we have shown that R is a bisimulation, which completes the proof.

Like the approximation lemma, coinduction gives a necessary and sufficient condition for the equality of two lists, and naturally generalises from lists to a large class of types [22]. However, coinduction has the advantage that proofs directly exploit the structure of programs themselves, whereas the approximation lemma relies on an auxiliary structure, namely natural numbers.

6. Fusion

The use of inductive or coinductive methods when proving properties of corecursive programs can often be avoided altogether by using *fusion* [25]. This method is derived from the use of the standard corecursive function *unfold p h t*, defined by the following equation:

$$\mathit{unfold}\ p\ h\ t\ x \quad = \quad \textbf{if}\ p\ x\ \textbf{then}\ [\,]\ \textbf{else}\ h\ x : \mathit{unfold}\ p\ h\ t\ (t\ x)$$

The function *unfold* encapsulates a simple pattern of corecursion for producing a list from a seed value x, by means of three argument functions p, h, and t. If the predicate p is true for the seed, then the empty list is produced. Otherwise the result is a non-empty list, whose head is produced by applying the function h to the seed, and whose tail is produced by applying the function t to the seed to generate a new seed, which is then itself unfolded in the same way. The function *unfold* encapsulates the natural basic pattern of corecursive definition (technically, it is the witness to the finality of the list type [25].)

Many familiar corecursive functions on lists can be defined using *unfold*. For example, the functions *iterate* f and *map* f can be defined by the following two equations, in which *false* is the constant predicate that holds of no argument, and *null* is the predicate on lists that holds only of the empty list:

$$\begin{array}{lcl}\mathit{iterate}\ f & = & \mathit{unfold}\ \mathit{false}\ \mathit{id}\ f \\ \mathit{map}\ f & = & \mathit{unfold}\ \mathit{null}\ (f \cdot \mathit{head})\ \mathit{tail}\end{array}$$

The basic method for proving properties of programs defined using *unfold* is its *universal property* [25], which is given by the following equivalence:

$$\boxed{f\ =\ \mathit{unfold}\ p\ h\ t \quad \Leftrightarrow \quad \forall x.\ f\ x\ =\ \textbf{if}\ p\ x\ \textbf{then}\ [\,]\ \textbf{else}\ h\ x :\ f\ (t\ x)}$$

This equivalence states that *unfold p h t* is not just a solution to its defining equation, but is in fact the *unique* solution. The left-to-right direction is trivially true, because substituting $f = \mathit{unfold}\ p\ h\ t$ into the right-hand side gives the definition for *unfold*. Conversely, for the other direction, by substitutivity and the approximation lemma the equation $f = \mathit{unfold}\ p\ h\ t$ is equivalent to:

$$\forall x, n.\ \mathit{approx}\ n\ (f\ x)\ =\ \mathit{approx}\ n\ (\mathit{unfold}\ p\ h\ t\ x)$$

This property can now be verified by induction on the natural number n, using the right-hand side of the universal property of *unfold* as an assumption. The base case $n = 0$ is trivially true because $\mathit{approx}\ 0\ \mathit{xs}$ is $\perp$ for all lists xs, while the inductive case $n = m + 1$ is verified as follows:

$$
\begin{array}{ll}
 & \mathit{approx}\ (m+1)\ (f\ x) \\
= & \quad \{\ \text{assumption}\ \} \\
 & \mathit{approx}\ (m+1)\ (\mathbf{if}\ p\ x\ \mathbf{then}\ [\,]\ \mathbf{else}\ h\ x : f\ (t\ x)) \\
= & \quad \{\ \text{distribution over } \mathbf{if}\ \} \\
 & \mathbf{if}\ p\ x\ \mathbf{then}\ \mathit{approx}\ (m+1)\ [\,]\ \mathbf{else}\ \mathit{approx}\ (m+1)\ (h\ x : f\ (t\ x)) \\
= & \quad \{\ \text{definition of } \mathit{approx}\ \} \\
 & \mathbf{if}\ p\ x\ \mathbf{then}\ \mathit{approx}\ (m+1)\ [\,]\ \mathbf{else}\ h\ x : \mathit{approx}\ m\ (f\ (t\ x)) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & \mathbf{if}\ p\ x\ \mathbf{then}\ \mathit{approx}\ (m+1)\ [\,]\ \mathbf{else}\ h\ x : \mathit{approx}\ m\ (\mathit{unfold}\ p\ h\ t\ (t\ x)) \\
= & \quad \{\ \text{definition of } \mathit{approx}\ \} \\
 & \mathbf{if}\ p\ x\ \mathbf{then}\ \mathit{approx}\ (m+1)\ [\,]\ \mathbf{else}\ \mathit{approx}\ (m+1)\ (h\ x : \mathit{unfold}\ p\ h\ t\ (t\ x)) \\
= & \quad \{\ \text{distribution over } \mathbf{if}\ \} \\
 & \mathit{approx}\ (m+1)\ (\mathbf{if}\ p\ x\ \mathbf{then}\ [\,]\ \mathbf{else}\ h\ x : \mathit{unfold}\ p\ h\ t\ (t\ x)) \\
= & \quad \{\ \text{definition of } \mathit{unfold}\ \} \\
 & \mathit{approx}\ (m+1)\ (\mathit{unfold}\ p\ h\ t\ x)
\end{array}
$$

The utility of the universal property of *unfold* is that it makes explicit the assumption required for a certain pattern of proof. For specific cases, by verifying this assumption (which can typically be done without the need for inductive or coinductive methods) we can then appeal to the universal property to complete the proof. In this manner, the universal property of *unfold* encapsulates a simple pattern of proof concerning corecursive programs, just as the function *unfold* itself encapsulates a simple pattern of definition for such programs. In practice, however, a corollary of the universal property called fusion is often preferable, which is given by the following inference rule:

$$
\boxed{\dfrac{p \cdot g = p' \qquad h \cdot g = h' \qquad t \cdot g = g \cdot t'}{\mathit{unfold}\ p\ h\ t\ \cdot\ g\ =\ \mathit{unfold}\ p'\ h'\ t'}}
$$

This rule states three conditions that together ensure that the composition of an *unfold* and a function can be fused together to give a single *unfold*, and can be derived (without using any form of induction or coinduction) from the universal property as follows:

$$
\begin{array}{ll}
& \mathit{unfold}\ p\ h\ t \cdot g \ = \ \mathit{unfold}\ p'\ h'\ t' \\
\Leftrightarrow & \quad \{ \text{ universal property } \} \\
& \forall x.\ (\mathit{unfold}\ p\ h\ t \cdot g)\ x \ = \\
& \quad \textbf{if}\ p'\ x\ \textbf{then}\ [\,]\ \textbf{else}\ h'\ x : (\mathit{unfold}\ p\ h\ t \cdot g)\ (t'\ x) \\
\Leftrightarrow & \quad \{ \text{ definition of composition } \} \\
& \forall x.\ \mathit{unfold}\ p\ h\ t\ (g\ x) \ = \\
& \quad \textbf{if}\ p'\ x\ \textbf{then}\ [\,]\ \textbf{else}\ h'\ x : \mathit{unfold}\ p\ h\ t\ (g\ (t'\ x)) \\
\Leftrightarrow & \quad \{ \text{ definition of } \mathit{unfold} \ \} \\
& \forall x.\ \textbf{if}\ p\ (g\ x)\ \textbf{then}\ [\,]\ \textbf{else}\ h\ (g\ x) : \mathit{unfold}\ p\ h\ t\ (t\ (g\ x)) \ = \\
& \quad \textbf{if}\ p'\ x\ \textbf{then}\ [\,]\ \textbf{else}\ h'\ x : \mathit{unfold}\ p\ h\ t\ (g\ (t'\ x)) \\
\Leftrightarrow & \quad \{ \text{ definition of composition } \} \\
& \forall x.\ \textbf{if}\ (p \cdot g)\ x\ \textbf{then}\ [\,]\ \textbf{else}\ (h \cdot g)\ x : \mathit{unfold}\ p\ h\ t\ ((t \cdot g)\ x) \ = \\
& \quad \textbf{if}\ p'\ x\ \textbf{then}\ [\,]\ \textbf{else}\ h'\ x : \mathit{unfold}\ p\ h\ t\ ((g \cdot t')\ x) \\
\Leftarrow & \quad \{ \text{ substitutivity } \} \\
& p \cdot g = p' \ \wedge\ h \cdot g = h' \ \wedge\ t \cdot g = g \cdot t'
\end{array}
$$

For example, using fusion it is easy to show that the composition of an *unfold* and the function used to generate the new seed value can always be fused together,

$$\mathit{unfold}\ p\ h\ t \ \cdot\ t \quad = \quad \mathit{unfold}\ (p \cdot t)\ (h \cdot t)\ t \qquad (1)$$

and that the composition of a *map* and an *unfold* can always be fused:

$$\mathit{map}\ f \ \cdot\ \mathit{unfold}\ p\ h\ t \quad = \quad \mathit{unfold}\ p\ (f \cdot h)\ t \qquad (2)$$

Using these general fusion laws, the *map-iterate* property can be proved at a higher level than using the other three proof methods that we have discussed in previous sections. First of all, by substitutivity and the definition of composition, the *map-iterate* property is equivalent to the following equation:

$$\mathit{iterate}\ f \cdot f \ = \ \mathit{map}\ f \cdot \mathit{iterate}\ f$$

This equation can now be verified as follows:

$$
\begin{array}{ll}
& \mathit{iterate}\ f \cdot f \\
= & \quad \{ \text{ definition of } \mathit{iterate} \ \} \\
& \mathit{unfold}\ \mathit{false}\ \mathit{id}\ f \cdot f \\
= & \quad \{ \text{ fusion (1) } \} \\
& \mathit{unfold}\ (\mathit{false} \cdot f)\ (\mathit{id} \cdot f)\ f \\
= & \quad \{ \text{ constant functions, composition } \} \\
& \mathit{unfold}\ \mathit{false}\ (f \cdot \mathit{id})\ f \\
= & \quad \{ \text{ fusion (2) } \} \\
& \mathit{map}\ f \cdot \mathit{unfold}\ \mathit{false}\ \mathit{id}\ f \\
= & \quad \{ \text{ definition of } \mathit{iterate} \ \} \\
& \mathit{map}\ f \cdot \mathit{iterate}\ f
\end{array}
$$

Being an implication, fusion is not always applicable as a proof method. However, in such cases one can often resort to reasoning explicitly using the universal property of *unfold*. Moreover, both fusion and the universal property naturally generalise from lists to a large class of types [25].

Using fusion has the advantage that proofs are performed purely algebraically, without reference to the underlying semantics of programs and without necessitating the use of any form of induction or coinduction. On the other hand, using fusion requires that corecursive programs are defined using *unfold*, which for more complicated examples can be unnatural, or impossible [16].

7. Conclusion

We have reviewed the four main methods for proving properties of corecursive programs, namely fixpoint induction, the approximation lemma, coinduction, and fusion. In this concluding section we compare the four methods, and provide some references to further reading.

Fixpoint induction. Using this method, proofs proceed by induction on the chain of approximations underlying the least fixpoint semantics of programs. It is the lowest-level of the four methods, resulting in proofs that can be viewed as being from “first principles”. For this reason, fixpoint induction is primarily used as a foundational tool. For example, our proofs of the other three methods presented in this article are all founded, directly or indirectly, upon fixpoint induction. For further reading, see [11, 26, 29].

Approximation lemma. Using this method, proofs proceed by induction on the depth of the structures being compared, which is governed by the use of an auxiliary function *approx*. The main appeal of the approximation lemma is that it allows proofs to be performed using the simple technique of induction on natural numbers. For further reading, see [5, 13, 21, 30].

Coinduction. This method directly exploits the structure of programs themselves, rather than relying on auxiliary structures such as cpos or natural numbers. Proofs proceed by finding a bisimulation that relates the two programs being compared. Coinduction is widely used for reasoning about concurrent processes, and is rapidly gaining popularity for reasoning about corecursive functional programs, with a range of variants being studied. For further reading, see [9, 18, 19, 22, 24, 31, 35, 36, 37, 40].

Fusion. This is the highest-level of the four methods, with proofs proceeding using properties of the higher-order function *unfold*. The main appeal of fusion is that it allows proofs to be performed purely equationally, without the need for induction or coinduction. Due to the need to define programs in a stylised form using *unfold*, fusion is a somewhat specialist method, but is widely used in the mathematics of program construction community. For further reading, see [3, 6, 15, 17].

Of course, the above methods do not constitute the end of the story, and new proof methods for corecursive programs will continue to be developed and explored. For example, Pitts’ work on exploiting parametricity to reason about types in non-strict languages such as Haskell [33] has recently been used to give the first formal proof of correctness of *short cut* fusion [23], and a categorical approach based upon universal properties has subsequently produced a simpler account [14].

Acknowledgements

We would like to thank David Sands, the anonymous referees and the guest editors of this special issue (Alberto Pettorossi and Maurizio Proietti) for many useful comments and suggestions.

References

[1] Aczel, P.: *Non-Well-Founded Sets,* Number 14 in CSLI Lecture Notes, Stanford: CSLI Publications, 1988.

[2] de Bakker, J.: *Mathematical Theory of Program Correctness,* Prentice-Hall, 1980.

[3] Bartels, F.: Generalised Coinduction, *Mathematical Structures in Computer Science*, **13**, 2003, 321–348.

[4] Barwise, J., Moss, L.: *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena,* Number 60 in CSLI Lecture Notes, Stanford: CSLI Publications, 1996.

[5] Bird, R.: *Introduction to Functional Programming using Haskell (second edition),* Prentice Hall, 1998.

[6] Bird, R., de Moor, O.: *Algebra of Programming,* Prentice Hall, 1997.

[7] Bird, R., Wadler, P.: *An Introduction to Functional Programming,* Prentice Hall, 1988.

[8] Burge, W.: *Recursive Programming Techniques,* Addison-Wesley, 1975.

[9] Coquand, T.: Infinite Objects in Type Theory, in: *Types for Proofs and Programs* (H. Barendregt, T. Nipkow, Eds.), vol. 806 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1994, 62–78.

[10] Danielsson, N., Jansson, P.: Chasing Bottoms: A Case Study in Program Verification in the Presense of Partial and Infinite Values, in: *Proceedings of the 7th International Conference on Mathematics of Program Construction*, vol. 3125 of *Lecture Notes in Computer Science*, Springer, Stirling, Scotland, July 2004.

[11] Davey, B., Priestley, H.: *Introduction to Lattices and Order,* Cambridge University Press, 1990.

[12] Fokkinga, M. M., Meijer, E.: *Program Calculation Properties of Continuous Algebras,* Technical Report CS–R9104, CWI, Amsterdam, January 1991, Available online at `http://wwwhome.cs.utwente.nl/~fokkinga/#detail_0000003528`.

[13] Ghani, N., Luth, C., de Marchi, F., Power, J.: Algebras, Coalgebras, Monads and Comonads, in: *Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Science*, number 44.1 in Electronic Notes in Theoretical Computer Science, Elsevier Science, April 2001.

[14] Ghani, N., Uustalu, T., Vene, V.: Build, Augment, Destroy. Universally, *Proceedings of Programming Languages and Systems: Second Asian Symposium*, LNCS 3302, Springer Verlag, 2004.

[15] Gibbons, J.: Calculating Functional Programs, in: *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, vol. 2297 of *Lecture Notes in Computer Science*, Springer-Verlag, January 2002, 148–203.

[16] Gibbons, J., Hutton, G., Altenkirch, T.: When is a Function a Fold or an Unfold?, in: *Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Science*, number 44.1 in Electronic Notes in Theoretical Computer Science, Elsevier Science, April 2001.

[17] Gibbons, J., Jones, G.: The Under-Appreciated Unfold, in: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998, 273–279.

[18] Gordon, A.: A Tutorial on Co-induction and Functional Programming, in: *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, Springer Workshops in Computing, 1995, 78–95.

[19] Gordon, A.: *Bisimilarity as a Theory of Functional Programming,* BRICS Notes Series NS-95-3, Aarhus University, 1995.

[20] Gumm, H. P., Ed.: *Proceedings of the Sixth International Workshop on Coalgebraic Methods in Computer Science*, Elsevier Science, 2003, Electronic Notes in Theoretical Computer Science Volume 82.1.

[21] Hutton, G., Gibbons, J.: The Generic Approximation Lemma, *Information Processing Letters*, **79**(4), August 2001, 197–201.

[22] Jacobs, B., Rutten, J.: A Tutorial on (Co)Algebras and (Co)Induction, *Bulletin of the European Association for Theoretical Computer Science*, **62**, 1997, 222–259.

[23] Johann, P.: Short Cut Fusion is Correct, *Journal of Functional Programming*, **13**(4), 2003, 797–814.

[24] Lassen, S.: Relational Reasoning About Contexts, in: *Higher Order Operational Techniques in Semantics* (A. Gordon, A. Pitts, Eds.), Cambridge University Press, 1998, 91–135.

[25] Malcolm, G.: Algebraic Data Types and Program Transformation, *Science of Computer Programming*, **14**(2-3), September 1990, 255–280.

[26] Manna, Z., Ness, S., Vuillemin, J.: Inductive Methods for Proving Properties of Programs, *Communications of the ACM*, **16**(8), August 1973, 491–502.

[27] Moss, L., Ed.: *Proceedings of the Fifth International Workshop on Coalgebraic Methods in Computer Science*, Elsevier Science, 2002, Electronic Notes in Theoretical Computer Science Volume 65.1.

[28] Moss, L., Danner, N.: On the Foundations of Corecursion, *Logic Journal of the Interest Group in Pure and Applied Logics*, **5**(2), 1997, 231–257.

[29] Park, D.: Fixpoint Induction and Proofs of Program Properties, *Machine Intelligence*, **5**, 1969, 59–78.

[30] Paulson, L.: Mechanizing Coinduction and Corecursion in Higher-Order Logic, *Journal of Logic and Computation*, **7**, 1997, 175–204.

[31] Pavlovic, D.: Guarded Induction on Final Coalgebras, in: *Coalgebraic Methods in Computer Science* (B. Jacobs, L. Moss, H. Reichel, J. Rutten, Eds.), number 11 in Electronic Notes in Theoretical Computer Science, Elsevier, 2000.

[32] Peyton Jones, S.: *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, 2003.

[33] Pitts, A. M.: Parametric Polymorphism and Operational Equivalence, *Mathematical Structures in Computer Science*, **10**, 2000, 321–359.

[34] Reynolds, J. C.: *Theories of Programming Languages*, Cambridge University Press, 1998.

[35] Sands, D.: Computing with Contexts: A Simple Approach, in: *Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)* (A. D. Gordon, A. M. Pitts, C. L. Talcott, Eds.), vol. 10 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science Publishers B.V., 1998.

[36] Sands, D.: Improvement Theory and Its Applications, in: *Higher Order Operational Techniques in Semantics* (A. Gordon, A. Pitts, Eds.), Cambridge University Press, 1998, 275–306.

[37] Sangiorgi, D.: On the Bisimulation Proof Method, *Mathematical Structures in Computer Science*, **8**, 1998, 447–479.

[38] Schmidt, D. A.: *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon, Inc., 1986.

[39] Smyth, M. B., Plotkin, G. D.: The Category-Theoretic Solution of Recursive Domain Equations, *SIAM Journal on Computing*, **11**(4), November 1982, 761–783.

[40] Turner, D. A.: Elementary Strong Functional Programming, in: *Proceedings of the First International Symposium on Functional Programming Languages in Education*, vol. 1022 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995, 1–13.