Proof methods for structured corecursive programs

Jeremy Gibbons[1] and Graham Hutton[2]

[1] University of Oxford; email Jeremy.Gibbons@comlab.ox.ac.uk
[2] University of Nottingham; email gmh@cs.nott.ac.uk

Abstract

Corecursive programs produce values of greatest fixpoint types, in contrast to recursive programs, which consume values of least fixpoint types. There are a number of widely used methods for proving properties of corecursive programs, including fixpoint induction, the *take lemma*, and coinduction. However, these methods are all rather low-level, in that they do not exploit the common structure that is often present in corecursive definitions. We argue for a more structured approach to proving properties of corecursive programs. In particular, we show that by writing corecursive programs using a simple operator that encapsulates a common pattern of corecursive definition, we can then use high-level algebraic properties of this operator to conduct proofs in a purely calculational style that avoids the use of inductive or coinductive methods.

1 INTRODUCTION

Recursion is a central concept in computing, with applications ranging from the theoretical foundations of computation [22] to practical programming techniques [5]. In recent years, it has become increasingly clear that the dual but less well-known concept of corecursion is just as central to computing [1, 2, 14, 15]. In this article, we explore methods for proving properties of corecursive programs.

As yet, there appears to be no standard definition for the notion of corecursion. In this article, we follow Moss and Danner's work on the foundations of corecursion [20] and use the term *corecursive program* for a function whose range is a type defined as a greatest fixpoint, such as the type of infinite lists or streams. Dually, we use the term *recursive program* for a function whose domain is a type defined as a least fixpoint, such as finite lists. Of course, these definitions are rather general (for example, they do not require the use of self-reference in any form), but they will suffice for our purposes here.

All the programs in the article are written in the standard lazy functional language Haskell [16]. This language has the convenient property that recursive types are simultaneously both least and greatest fixpoints [8]. Hence, in this article, the terms 'recursive program' and 'corecursive program' simply refer to Haskell functions whose domain and range are recursive types, respectively.

The most widely used method for proving properties of recursive programs is *structural induction*. Unfortunately, this method is not applicable to corecursive programs, because in general such programs do not have an argument over which induction can be performed. Historically, the basic proof method for corecursive programs has been Scott and de Bakker's *fixpoint induction* [7], which arises from the standard denotational semantics of functional programs. Applying fixpoint induction is rather tedious, but for the special case of corecursive programs that produce lists, one can use Bird and Wadler's *take lemma* [4], or the simpler *approximation lemma* [3]. More recently, Gordon [11] and Turner [24] have argued for the use of *coinduction* as the basic proof method for corecursive programs, which arises from work on the operational semantics of functional programs.

In this article we review the above proof methods, and observe that they are all rather low-level, in the sense that they do not exploit the common structure that is often present in corecursive definitions. We argue for a more structured approach to proving properties of corecursive programs. In particular, we show that by writing corecursive programs using a simple operator called *unfold* that encapsulates a common pattern of corecursive definition, we can then use the high-level *universal* and *fusion* properties of this operator to conduct proofs in a purely calculational style that avoids the use of inductive or coinductive methods. This approach is dual to the use of *fold* and its associated properties in recursive programming, which has recently been surveyed in [12].

The article is aimed at a reader who is familiar with the basics of recursive programming and inductive proof, say to the level of [4, 3]. No prior knowledge of corecursive programming and proof is assumed. For simplicity, we restrict our attention to corecursive programs that produce lists, but our approach naturally generalises to a large class of other datatypes.

2 CORECURSIVE PROGRAMS

In this section we present a few simple examples of corecursive programs and their properties. To begin with, consider the following Haskell definitions for corecursive programs that produce infinite lists:

$$
\begin{array}{lcl}
\mathit{repeat} & :: & \alpha \to [\alpha] \\
\mathit{repeat}\ x & = & x : \mathit{repeat}\ x \\[4pt]
\mathit{from} & :: & \mathit{Int} \to [\mathit{Int}] \\
\mathit{from}\ n & = & n : \mathit{from}\ (n+1) \\[4pt]
\mathit{iterate} & :: & (\alpha \to \alpha) \to \alpha \to [\alpha] \\
\mathit{iterate}\ f\ x & = & x : \mathit{iterate}\ f\ (f\ x)
\end{array}
$$

Unwinding these definitions a few steps, we see that:

$$
\begin{array}{lcl}
\mathit{repeat}\ x & = & x : x : x : x : \ldots \\
\mathit{from}\ n & = & n : (n+1) : (n+2) : (n+3) : \ldots \\
\mathit{iterate}\ f\ x & = & x : f\ x : f\ (f\ x) : f\ (f\ (f\ x)) : \ldots
\end{array}
$$

From these examples, it is clear that *repeat* and *from* are special cases of *iterate*, given by $\mathit{repeat} = \mathit{iterate}\ \mathit{id}$ and $\mathit{from} = \mathit{iterate}\ (+1)$. The *iterate* operator encapsulates a general pattern for producing an infinite list, in which the first value in the list is provided as a seed, and each subsequent value in the list is generated by applying a given function to the previous value.

Another general pattern is encapsulated by the well-known *map* operator, which produces a list by applying a function to each value in a given list:

$$
\begin{array}{lcl}
\mathit{map} & :: & (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\mathit{map}\ f\ \mathit{xs} & = & \mathbf{if}\ \mathit{null}\ \mathit{xs}\ \mathbf{then}\ [\,]\ \mathbf{else}\ f\ (\mathit{head}\ \mathit{xs}) : \mathit{map}\ f\ (\mathit{tail}\ \mathit{xs})
\end{array}
$$

For example:

$$\mathit{map}\ f\ (x_1 : x_2 : x_3 : x_4 : \ldots) \quad = \quad f\ x_1 : f\ x_2 : f\ x_3 : f\ x_4 : \ldots$$

Note that the function *map f* is recursive as well as being corecursive, because it both consumes and produces a list. The definition of *map* above using *null*, *head* and *tail* is more verbose than the standard definition using pattern matching. However, we choose to define *map* in this manner because it reveals a structure that we will find to be common to many corecursive programs.

In general, the same infinite list may be produced by many different corecursive programs. For example, by again unwinding the appropriate definitions a few steps, we see that we can produce the infinite list

$$f\ x : f\ (f\ x) : f\ (f\ (f\ x)) : f\ (f\ (f\ (f\ x))) : \ldots$$

by at least three different programs: $\mathit{tail}\ (\mathit{iterate}\ f\ x)$, $\mathit{map}\ f\ (\mathit{iterate}\ f\ x)$, and $\mathit{iterate}\ f\ (f\ x)$. That is, we expect the following equations to be true:

$$tail \cdot iterate\, f \;=\; map\, f \cdot iterate\, f \;=\; iterate\, f \cdot f$$

We conclude this section with four further examples of corecursive programs: *copy x* which produces a list of *x*'s of a given length, *digits* which extracts the decimal digits (in reverse order) from a number, *tails* which returns all the (non-empty) final segments of a list, and *sort* which implements selection sort using an auxiliary function *delmin* that removes the minimum number from a list:

$$\begin{array}{lcl}
copy & :: & \alpha \to Int \to [\alpha] \\
copy\ x\ n & = & \mathbf{if}\ n == 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ x : copy\ x\ (n-1) \\
digits & :: & Int \to [Int] \\
digits\ n & = & \mathbf{if}\ n == 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ (n\ \mathbf{mod}\ 10) : digits\ (n\ \mathbf{div}\ 10) \\
tails & :: & [\alpha] \to [[\alpha]] \\
tails\ xs & = & \mathbf{if}\ null\ xs\ \mathbf{then}\ [\,]\ \mathbf{else}\ xs : tails\ (tail\ xs) \\
sort & :: & [Int] \to [Int] \\
sort\ ns & = & \mathbf{if}\ null\ ns\ \mathbf{then}\ [\,]\ \mathbf{else}\ minimum\ ns : sort\ (delmin\ ns)
\end{array}$$

The structure of these definitions is clearly similar to that for *map*. In fact, we will see later on that all the functions defined above are instances of a common pattern of corecursive definition, and that we can exploit this fact when proving properties of these functions. Prior to this, however, we review and compare the standard proof techniques for corecursive programs.

3 FIXPOINT INDUCTION

Recall that in the standard denotational approach to the semantics of functional languages [23], types are partially ordered sets with a least element $\bot$ and limits of all non-empty chains (that is, types are *cpos*), and programs are functions between cpos that preserve these limits (that is, programs are *continuous functions*). In this setting, the meaning of a definition $x = f\ x$ for a value x in terms of itself and some auxiliary function f is given by the limit of the following chain:

$$\bot \;\sqsubseteq f\ \bot \;\sqsubseteq f\ (f\ \bot) \;\sqsubseteq f\ (f\ (f\ \bot)) \;\sqsubseteq\; \cdots$$

The well-known fixpoint theorem [6] states that the limit of this chain, denoted by *fix f*, is the least solution to the equation $x = f\ x$ (that is, *fix f* is the *least fixpoint* of f), and is hence an appropriate choice for the meaning of x. As an example of the least fixpoint approach to semantics, recall the defining equation for the corecursive program *repeat* given earlier:

$$\begin{array}{lcl}
repeat & :: & \alpha \to [\alpha] \\
repeat\ x & = & x : repeat\ x
\end{array}$$

This definition can be rewritten as $repeat = f\ repeat$, where the auxiliary function f is defined without reference to itself as follows:

$$\begin{array}{lcl}
f & :: & (\alpha \to [\alpha]) \to \alpha \to [\alpha] \\
f\ g\ x & = & x : g\ x
\end{array}$$

Hence, the meaning of *repeat* is given by $\mathit{repeat} = \mathit{fix}\ f$, from which a simple calculation shows that $\mathit{repeat}\ x$ is the limit of the following chain of partial lists containing increasing numbers of x's, as expected:

$$\bot \ \sqsubseteq\ x : \bot \ \sqsubseteq\ x : x : \bot \ \sqsubseteq\ x : x : x : \bot \ \sqsubseteq\ \ldots$$

The basic method for proving properties of programs defined using *fix* is Scott and de Bakker's *fixpoint induction* [7]. Suppose that f is a continuous function on a cpo and P is a predicate on the same cpo, such that whenever P holds of all the elements in a chain then it also holds of the limit (that is, P is *chain complete*.) Then fixpoint induction can be stated as the following inference rule:

$$\frac{P\ \bot \qquad \forall x.\ P\ x \Rightarrow P\ (f\ x)}{P\ (\mathit{fix}\ f)}$$

This rule states that if the predicate holds of the least element $\bot$ of the cpo, and whenever it holds of an element x then it also holds of $f\ x$, then the predicate also holds of the least fixpoint $\mathit{fix}\ f$. The proof of the fixpoint induction rule is a simple consequence of the definition of $\mathit{fix}\ f$ as the limit of a chain.

As an example, let us see how fixpoint induction can be used to prove the property $\mathit{map}\ f \cdot \mathit{iterate}\ f\ =\ \mathit{iterate}\ f\ \cdot f$ from the previous section. First of all, we abstract from $\mathit{iterate}\ f$ and define a predicate $P\ g\ \Leftrightarrow\ \mathit{map}\ f \cdot g\ =\ g \cdot f$, so that the property to be proved can now be written as $P\ (\mathit{iterate}\ f)$. The predicate P is chain complete, because any predicate expressed as an equation between continuous functions is chain complete. Next, the meaning of *iterate* is given by $\mathit{iterate}\ f = \mathit{fix}\ (h\ f)$, where the function h is defined as follows:

$$\begin{array}{lcl} h & :: & (\alpha \to \alpha) \to (\alpha \to [\alpha]) \to \alpha \to [\alpha] \\ h\ f\ g\ x & = & x : g\ (f\ x) \end{array}$$

Hence, the property to be proved can be written as $P\ (\mathit{fix}\ (h\ f))$, which, by fixpoint induction, is implied by the following two conditions:

$$P\ \bot \qquad \forall g.\ P\ g \Rightarrow P\ (h\ f\ g)$$

Finally, these conditions are verified by two calculations:

$$\begin{array}{ll} & P\ \bot \\ \Leftrightarrow & \quad \{\ \text{definition of } P\ \} \\ & \mathit{map}\ f\ \cdot\ \bot\ =\ \bot\ \cdot\ f \\ \Leftrightarrow & \quad \{\ \text{extensionality, composition}\ \} \\ & \forall x.\ \mathit{map}\ f\ (\bot\ x)\ =\ \bot\ (f\ x) \\ \Leftrightarrow & \quad \{\ \beta\text{-reduction}\ \} \\ & \forall x.\ \mathit{map}\ f\ \bot\ =\ \bot \\ \Leftrightarrow & \quad \{\ \mathit{map}\ f \text{ is strict}\ \} \\ & \mathit{true} \end{array}$$

and

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

This completes the proof. Note that because fixpoint induction is an implication rather than an equivalence, it is sound but not complete, and is hence not always applicable. For example, with P defined as “is an infinite list” (a chain-complete predicate) and $f\ x = 1 : x$, the true statement $P\ (\mathit{fix}\,f)$ expresses that $1 : 1 : 1 : 1 : \ldots$ is an infinite list, but cannot be proved using fixpoint induction because $\perp$ is not an infinite list and hence the base case $P\ \perp$ is false.

4 THE APPROXIMATION LEMMA

Fixpoint induction is a rather basic 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. Fortunately, for corecursive programs that produce lists, Bird and Wadler’s *take lemma* [4] allows us to perform proofs at the level of the syntax of programs, without reference to their underlying fixpoint semantics.

Recently, the take lemma has been superseded by the *approximation lemma* [3], which is formally equivalent to the take lemma, but is easier to prove, slightly simpler to apply, and naturally generalises from lists to a large class of other datatypes [9]. The basis of the approximation lemma is the *approx* function:

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

The *approx* function is similar to the well-known function *take*, except that there is no base case for *approx* 0, so by case exhaustion $approx\ 0\ xs = \perp$ for all xs. For example, $approx\ 3\ (from\ 0)\ =\ 0 : 1 : 2 : \perp$, while $approx\ 3\ [0,1]\ =\ [0,1]$.

Suppose that xs and ys are two infinite, partial, or finite lists. Then the approximation lemma can be stated as the following equivalence:

$$xs = ys \quad \Leftrightarrow \quad \forall n.\ approx\ n\ xs = approx\ n\ ys$$

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

$$approx\ 0 \sqsubseteq approx\ 1 \sqsubseteq approx\ 2 \sqsubseteq approx\ 3 \sqsubseteq \ \ldots$$

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

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

As an example, let us see how $map\ f \cdot iterate\ f = iterate\ f \cdot f$ can be proved more simply by using the approximation lemma than by fixpoint induction. First of all, we use extensionality and the definition of composition to rewrite the property to be proved in the equivalent form:

$$\forall x.\ map\ f\ (iterate\ f\ x) = iterate\ f\ (f\ x)$$

Now, by the approximation lemma, this is equivalent to:

$$\forall x, n.\ \ approx\ n\ (map\ f\ (iterate\ f\ x)) \ = \ approx\ n\ (iterate\ f\ (f\ x))$$

Finally, this property can be verified by induction on the natural number n. The base case $n = 0$ is trivially true because $approx\ 0\ xs = \bot$ for all xs. For the inductive case $n = m + 1$, we calculate as follows:

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

which completes the proof.

As well as leading to simpler proofs, the approximation lemma is an equivalence so is both sound and complete, in contrast to the fixpoint induction rule.

5 BISIMILARITY AND COINDUCTION

Both fixpoint induction and the approximation lemma are based on denotational semantics. Another popular approach to semantics is the operational approach [21], for which a widely-used notion of equivalence for programs is *bisimilarity*, and the basic proof method for establishing that two programs are bisimilar is *coinduction* [19]. In this section, we review Gordon's work [11] on applying bisimilarity and coinduction to prove properties of corecursive functional programs, techniques also advocated by Turner [24].

The operational semantics of a Haskell-like language can be defined by a reduction relation $\leadsto$, for which $a \leadsto a'$ means that the expression a can be reduced to the expression a' in a single execution step. For example, the following reductions formalise how *iterate* and *map* are executed:

$$\begin{array}{lll} \mathit{iterate}\ f\ x & \leadsto & x : \mathit{iterate}\ f\ (f\ x) \\ \mathit{map}\ f\ [\,] & \leadsto & [\,] \\ \mathit{map}\ f\ (x : xs) & \leadsto & f\ x : \mathit{map}\ f\ xs \end{array}$$

Using the reduction relation $\leadsto$ we can then define a labelled transition relation $\rightarrow$, for which $a \xrightarrow{o} a'$ means that the expression a immediately permits the *observation* o, thereby making the transition to the expression a'. For example, the following transitions formalise the observations that can be made of a non-empty list, namely that we can observe its *head* and *tail*:

$$x : xs \xrightarrow{\mathit{head}} x$$

$$x : xs \xrightarrow{\mathit{tail}} xs$$

By repeated application of the transition relation $\rightarrow$, we can generate a (possibly infinite) transition tree that captures all possible sequences of observations for a given expression. Informally, two expressions are called *bisimilar* if their transition trees are identical when we ignore the expressions at the nodes in the trees, and only consider the observations that label the edges. That is, two expressions are bisimilar if they cannot be distinguished by an observer who has no knowledge of the internal details of the expressions.

Formally, a *bisimulation* in this context is a relation R on expressions such that if $a\ R\ b$ then the following two conditions are satisfied:

Whenever $a \xrightarrow{o} a'$ there is some b' for which $b \xrightarrow{o} b'$ and $a'\ R\ b'$

Whenever $b \xrightarrow{o} b'$ there is some a' for which $a \xrightarrow{o} a'$ and $a'\ R\ b'$

That is, expressions that are related by a bisimulation permit the same observations and thereby make transitions to related expressions. The above definition does not determine a unique relation, but there is always a greatest bisimulation under the inclusion ordering on relations, which relates as many expressions as possible subject to the two conditions above. The greatest bisimulation is written $\sim$, and two expressions for which $a \sim b$ are called *bisimilar*. The fact that, by definition, all other bisimulations are included in the greatest bisimulation is known as *coinduction*; it forms a simple but powerful proof method.

To prove that $a \sim b$ using coinduction, we must construct a bisimulation R for which $a\ R\ b$. Then, by coinduction we know that R is included in $\sim$, and hence because $a\ R\ b$ we conclude that $a \sim b$, as required. That is, by coinduction the problem of showing that two expressions are bisimilar can be reduced to the problem of constructing a bisimulation that relates the two expressions.

As an example, let us prove that $\mathit{map}\ f \cdot \mathit{iterate}\ f \sim \mathit{iterate}\ f \cdot f$ using coinduction. First of all, we rewrite the property in the equivalent form:

$$\forall f, x.\ \ \mathit{map}\ f\ (\mathit{iterate}\ f\ x) \sim \mathit{iterate}\ f\ (f\ x)$$

Next, we construct a relation R that encodes this property:

$$R \quad = \quad \{(\mathit{map}\ f\ (\mathit{iterate}\ f\ x), \mathit{iterate}\ f\ (f\ x)) \mid f :: \alpha \to \alpha, x :: \alpha\}$$

The relation R is not itself a bisimulation, but we will now show that $R \cup \sim$ is a bisimulation. Suppose that $a\,(R \cup \sim)\,b$, which means that either $a\,R\,b$ or $a \sim b$. If $a \sim b$, clearly any transition for a is matched by one for b and vice versa, because $\sim$ is by definition a bisimulation. If $a\,R\,b$, then $a = \mathit{map}\ f\ (\mathit{iterate}\ f\ x)$ and $b = \mathit{iterate}\ f\ (f\ x)$ for some f and x. From the definitions of the functions *map* and *iterate* under the reduction relation $\leadsto$, we have that

$$\begin{array}{lll} a & \leadsto^* & f\ x : \mathit{map}\ f\ (\mathit{iterate}\ f\ (f\ x)) \\ b & \leadsto^* & f\ x : \mathit{iterate}\ f\ (f\ (f\ x)) \end{array}$$

where $\leadsto^*$ is the reflexive and transitive closure of $\leadsto$. Now, using the fact that if $c \leadsto^* c'$ then $c \xrightarrow{o} c''$ is equivalent to $c' \xrightarrow{o} c''$, together with the definition of the transition relation $\to$ for the case of non-empty lists, we can enumerate all the possible transitions for the expressions a and b:

$$\begin{array}{lll} a & \overset{\mathit{head}}{\to} & f\ x \\ a & \overset{\mathit{tail}}{\to} & \mathit{map}\ f\ (\mathit{iterate}\ f\ (f\ x)) \\ b & \overset{\mathit{head}}{\to} & f\ x \\ b & \overset{\mathit{tail}}{\to} & \mathit{iterate}\ f\ (f\ (f\ x)) \end{array} \qquad \begin{array}{r} (1) \\ (2) \\ (3) \\ (4) \end{array}$$

Transition (1) is matched by transition (3) and vice versa, because $\sim$ is necessarily reflexive and hence $f\ x \sim f\ x$, which implies that $f\ x\ (R \cup \sim)\ f\ x$. Transition (2) is matched by transition (4) and vice versa, because the resulting expressions $\mathit{map}\ f\ (\mathit{iterate}\ f\ (f\ x))$ and $\mathit{iterate}\ f\ (f\ (f\ x))$ are related by R and so also by $R \cup \sim$. This completes the proof that $R \cup \sim$ is a bisimulation, from which we conclude by coinduction that $R \cup \sim$ is included in $\sim$, and hence that R itself is included in $\sim$, and so the two expressions are bisimilar.

Coinduction is certainly an elegant and powerful proof method. However, denotational semantics is still the dominant basis for proofs about functional programs, and it seems unfortunate to have to change the basis to operational semantics in order to perform proofs using coinduction.

6 UNFOLD AND UNIVERSALITY

The three proof methods for corecursive programs that we have considered so far — fixpoint induction, the approximation lemma, and coinduction — are all rather low-level. In particular, they do not exploit the common structure that is often present in corecursive definitions. In this section, we show that by writing corecursive programs using a simple operator called *unfold* that encapsulates a common pattern of corecursive definition, we can then use the high-level *universal* and *fusion* properties [18, 10, 13] of this operator to conduct proofs in a purely calculational style that avoids the use of inductive or coinductive methods.

The *unfold* operator for lists may be defined as follows:

$$\begin{array}{lcl} \mathit{unfold} & :: & (\alpha \to \mathit{Bool}) \to (\alpha \to \beta) \to (\alpha \to \alpha) \to \alpha \to [\beta] \\ \mathit{unfold}\ p\ h\ t\ x & = & \mathbf{if}\ p\ x\ \mathbf{then}\ [\,]\ \mathbf{else}\ h\ x : \mathit{unfold}\ p\ h\ t\ (t\ x) \end{array}$$

The *unfold* operator encapsulates a general pattern 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 x, then the empty list is produced. Otherwise, the head of the list is produced by applying the function h to the seed x, and the tail is produced by first applying the function t to x to generate a new seed, which is then itself unfolded using the same process to produce the tail of the list.

Many corecursive programs have a simple definition using *unfold*. For example, the corecursive programs from Section 2 can naturally be defined by:

$$\begin{array}{lcl} \mathit{iterate}\ f & = & \mathit{unfold}\ (\mathit{const}\ \mathit{False})\ \mathit{id}\ f \\ \mathit{map}\ f & = & \mathit{unfold}\ \mathit{null}\ (f \cdot \mathit{head})\ \mathit{tail} \\ \mathit{copy}\ x & = & \mathit{unfold}\ (==\ 0)\ (\mathit{const}\ x)\ (-1) \\ \mathit{digits} & = & \mathit{unfold}\ (==\ 0)\ (\mathbf{mod}\ 10)\ (\mathbf{div}\ 10) \\ \mathit{tails} & = & \mathit{unfold}\ \mathit{null}\ \mathit{id}\ \mathit{tail} \\ \mathit{sort} & = & \mathit{unfold}\ \mathit{null}\ \mathit{minimum}\ \mathit{delmin} \end{array}$$

Here, *const c* is the constant function that always returns c.

The basic method for proving properties of programs defined using *unfold* is the universal property [18], which can be stated as the following equivalence:

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

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*. To prove the other direction, we first use extensionality and the approximation lemma to rewrite $f = \mathit{unfold}\ p\ h\ t$ in the equivalent form:

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

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 $\mathit{approx}\ 0\ \mathit{xs} = \perp$ for all xs. For the inductive case $n = m + 1$, we calculate as follows:

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

The universal property makes explicit the precise condition required to prove that $f = unfold\ p\ h\ t$. For specific cases, verifying this condition typically does not require inductive or coinductive methods. In this manner, the universal property of *unfold* encapsulates a general pattern of proof for corecursive programs, just as *unfold* itself encapsulates a general pattern of definition for such programs.

As a first example, let us see how the equation $repeat = iterate\ id$ from Section 2 can be proved by simple calculation using the universal property:

$$
\begin{array}{ll}
& repeat = iterate\ id \\
\Leftrightarrow & \quad \{ \text{ definition of } iterate\ \} \\
& repeat = unfold\ (const\ False)\ id\ f \\
\Leftrightarrow & \quad \{ \text{ universal property } \} \\
& \forall x.\ repeat\ x = \textbf{if}\ const\ False\ x\ \textbf{then}\ []\ \textbf{else}\ id\ x : repeat\ (id\ x) \\
\Leftrightarrow & \quad \{ \text{ simplification } \} \\
& \forall x.\ repeat\ x = x : repeat\ x \\
\Leftrightarrow & \quad \{ \text{ definition of } repeat\ \} \\
& true
\end{array}
$$

As a more general example, the universal property can be used to calculate the fusion law, which gives conditions under which the composition of an *unfold* and a function can be fused together to give a single *unfold*:

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

That is, fusion can be stated as the following inference rule:

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

Many properties of functions defined using *unfold* have a simple proof using fusion. For example, it is easy to show that the composition of an *unfold* and the function used to generate the new seed can always be fused, and that the composition of a *map* and an *unfold* can always be fused:

$$\begin{array}{lcll} \mathit{unfold}\ p\ h\ t\ \cdot\ t & = & \mathit{unfold}\ (p \cdot t)\ (h \cdot t)\ t & (1) \\ \mathit{map}\ h'\ \cdot\ \mathit{unfold}\ p\ h\ t & = & \mathit{unfold}\ p\ (h' \cdot h)\ t & (2) \end{array}$$

Using fusion, $\mathit{map}\ f\ \cdot\ \mathit{iterate}\ f\ =\ \mathit{iterate}\ f\ \cdot f$ can be proved much more simply than using the other three methods that we have discussed:

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

As our final example, we prove that the corecursive program *tails* satisfies the property $\mathit{tails} \cdot \mathit{tails}\ =\ \mathit{map\ tails} \cdot \mathit{tails}$, using fusion:

$$\begin{array}{ll} & \mathit{tails} \cdot \mathit{tails} \\ = & \quad \{\ \text{definition of } \mathit{tails}\ \} \\ & \mathit{unfold\ null\ id\ tail} \cdot \mathit{tails} \\ = & \quad \{\ \text{fusion, lemma (see below)}\ \} \\ & \mathit{unfold\ null\ tails\ tail} \\ = & \quad \{\ \text{fusion (2)}\ \} \\ & \mathit{map\ tails} \cdot \mathit{unfold\ null\ id\ tail} \\ = & \quad \{\ \text{definition of } \mathit{tails}\ \} \\ & \mathit{map\ tails} \cdot \mathit{tails} \end{array}$$

The lemma used in the first fusion step above is that $\mathit{tail} \cdot \mathit{tails} = \mathit{tails} \cdot \mathit{tail}$, which can easily be verified by a (non-inductive) case analysis on lists.

7 SUMMARY

We have explored a number of widely-used proof methods for corecursive programs, and argued for a more structured approach using the universal and fusion properties of the *unfold* operator. In particular, we have shown that these properties allow proofs to be conducted using simple equational reasoning, without having to refer to the underlying semantics of programs (denotational, as with fixpoint induction, or operational, as with coinduction) or use any form of induction (as with fixpoint induction and the approximation lemma).

For simplicity we have focussed on the *unfold* operator for lists, but our approach naturally generalises to operators that encapsulate more general patterns of corecursive definition (for example, primitive corecursion [25]), and to any datatype that can be defined as the greatest fixpoint of a functor [17].

ACKNOWLEDGEMENTS

The second author is supported by EPSRC grant *Structured Recursive Programming*, and ESPRIT Working Group *Applied Semantics*.

REFERENCES

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

[2] Jon Barwise and Lawrence Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. Number 60 in CSLI Lecture Notes. Stanford: CSLI Publications, 1996.

[3] Richard Bird. *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall, 1998.

[4] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.

[5] W.H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

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

[7] Jaco de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[8] Peter Freyd. Algebraically complete categories. In A. Carboni et al, editor, *Proc. 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Math*, pages 95–104. Springer-Verlag, Berlin, 1990.

[9] Jeremy Gibbons and Graham Hutton. The generic approximation lemma. In preparation, 1999.

[10] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.

[11] Andrew Gordon. Bisimilarity as a theory of functional programming. BRICS Notes Series NS-95-3, Aarhus University, 1995.

[12] Graham Hutton. A tutorial on the universality and expressiveness of fold. To appear in the Journal of Functional Programming.

[13] Graham Hutton. Fold and unfold for program semantics. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, 1998.

[14] Bart Jacobs, Larry Moss, Horst Reichel, and Jan Rutten, editors. *Proc. of the First Workshop on Coalgebraic Methods in Computer Science*. Elsevier Science B.V., 1998. Electronic Notes in Theoretical Computer Science Volume 11.

[15] Bart Jacobs and Jan Rutten, editors. *Proc. of the Second Workshop on Coalgebraic Methods in Computer Science*. Elsevier Science B.V., 1999. Electronic Notes in Theoretical Computer Science Volume 19.

[16] Simon Peyton Jones et al. Haskell 98: A non-strict, purely functional language. Available on the World-Wide-Web from `http://www.haskell.org`, February 1999.

[17] Grant Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Groningen University, 1990.

[18] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proc. Conference on Functional Programming and Computer Architecture*, number 523 in LNCS. Springer-Verlag, 1991.

[19] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[20] Lawrence Moss and Norman Danner. On the foundations of corecursion. *Logic Journal of the IGPL*, 5(2):231–257, 1997.

[21] Gordon Plotkin. A structured approach to operational semantics. Report DAIMI-FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[22] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

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

[24] David A. Turner. Elementary strong functional programming. In *Proc. First Internation Symposium on Functional Programming Languages in Education*, LNCS 1022, pages 1–13. Springer-Verlag, 1995.

[25] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). In *9th Nordic Workshop on Programming Theory*, Oct 1997.