Proof methods for corecursive programs

Jeremy Gibbons
University of Oxford
and
Graham Hutton
University of Nottingham

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 four methods for proving properties of corecursive programs: fixpoint induction, the approximation lemma, coinduction, and fusion.

Categories and Subject Descriptors: F.3 [**Theory of Computation**]: Logics and Meanings of Programs; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms: Theory, Verification

Additional Key Words and Phrases: functional programming, corecursion, greatest fixpoints, fixpoint induction, approximation lemma, coinduction, fusion

1. INTRODUCTION

Recursion is a central concept in computing, with applications ranging from the theoretical foundations of computation [Reynolds 1998] to practical programming techniques [Burge 1975]. In recent years, it has become increasingly clear that the dual but less well-known concept of *corecursion* is just as useful [Aczel 1988; Barwise and Moss 1996; Jacobs et al. 1998; Jacobs and Rutten 1999].

As yet, there is no standard definition for the notion of corecursion. We follow the work of Moss and Danner [1997] on the foundations of corecursion, and use the term *corecursive program* for a function whose range is a greatest fixpoint type, such as the type of infinite lists or streams. Dually, we use the term *recursive program* for a function whose domain is a least fixpoint type, such as the type of finite lists. These definitions are rather general — for example, they do not require any form of self-reference — but they will suffice for our purposes here, and nothing in what follows depends crucially on them.

The programming paradigm used in the article is that of a pure functional language with non-strict semantics, such as Haskell [Peyton Jones et al. 1999]. Using a pure functional language permits proofs by simple equational reasoning, while the use of non-strict semantics has the convenient property that recursive types are simultaneously both least and greatest fixpoint types [Freyd 1990]. Hence, in this article recursive programs and corecursive programs are simply functional programs

The second author was supported by EPSRC Research Grant L74491 *Structured Recursive Programming* and ESPRIT Working Group 26142 *Applied Semantics*.
A preliminary version of this article was presented at the 1st Scottish Functional Programming Workshop, Stirling, Scotland, August 1999.

that have recursive types as their domain and range, respectively.

Historically, the basic method for proving properties of corecursive programs is *fixpoint induction* [de Bakker 1980], which is derived from the denotational approach to programming language semantics. Applying fixpoint induction is rather tedious, but for many applications we can use the higher-level *approximation lemma* [Bird 1998], a recent improvement of the better known *take lemma* [Bird and Wadler 1988]. Alternatively, we can adopt the operational approach to semantics, in which case the basic proof method is *coinduction* [Gordon 1995]. However, the use of inductive or coinductive methods can often be avoided altogether by using *fusion* [Malcolm 1990], an algebraic law that encapsulates a common pattern of proof.

This article is a tutorial on the above methods for proving properties of corecursive programs. It is aimed at a reader who is familiar with the basics of recursive programming and proof, but no prior knowledge of corecursive programming and proof is assumed. For simplicity, we restrict our attention to corecursive programs that produce lists, but none of the methods discussed are specific to this type.

2. EXAMPLE: 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 $ones = 1 : ones$ defines the infinite list $ones = 1 : 1 : 1 : \cdots$. 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:

$$iterate\ f\ x\ =\ x : iterate\ f\ (f\ x)$$

For example, if $succ$ is the successor function on natural numbers, then $nats = iterate\ succ\ 0$ defines the infinite list $nats = 0 : 1 : 2 : \cdots$. 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 by two equations:

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

For example, $map\ succ\ nats$ produces the infinite list $1 : 2 : 3 : \cdots$, as does $iterate\ succ\ (succ\ 0)$. Generalising from this example gives the *map-iterate property*, which expresses that iterating a function f and then mapping f produces the same infinite list as applying f to the seed value and then iterating f:

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

This property cannot be proved using the standard method of structural induction on lists, because there is no list argument over which induction can be performed. In the remainder of the article we review and compare four different methods that *can* be used to prove such properties of corecursive functions.

3. FIXPOINT INDUCTION

Fixpoint induction is derived from the denotational approach to programming language semantics [Schmidt 1986]. The central idea in this approach is that types are *complete partial orders* (cpos), that is, partially-ordered sets with a least element

$\bot$ 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 [Davey and Priestley 1990] 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 chain:

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

As a simple example of this approach to semantics, consider again the equation $\mathit{ones} = 1 : \mathit{ones}$ that defines an infinite list ones. This definition can be rewritten as the equation $\mathit{ones} = f\ \mathit{ones}$, where the auxiliary function f is defined without reference to itself by $f\ \mathit{xs} = 1 : \mathit{xs}$. Hence, the semantics of the definition is given by $\mathit{ones} = \mathit{fix}\ f$, which means that ones is constructed as the limit of the following chain of partial lists containing increasing numbers of 1s, as expected:

$$\bot \ \sqsubseteq\ 1 : \bot \ \sqsubseteq\ 1 : 1 : \bot \ \sqsubseteq\ 1 : 1 : 1 : \bot \ \sqsubseteq\ \cdots$$

The basic method for proving properties of programs defined using fix is Scott and de Bakker's *fixpoint induction* [de Bakker 1980]. 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:

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

This inference rule states that if the predicate holds of the least element $\bot$ 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:

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

As a simple 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* above and define a chain-complete predicate P on functions by the following equivalence:

$$P\ g \quad \Leftrightarrow \quad \forall f, x.\ map\ f\ (g\ f\ x)\ =\ g\ f\ (f\ x)$$

Using this definition, the *map-iterate* property can be written as $P\ iterate$. In turn, the semantics of the function *iterate* is given by $iterate = fix\ h$, where the auxiliary function h is defined by $h\ g\ f\ x = x : g\ f\ (f\ x)$. Hence, the *map-iterate* property can now be written as $P\ (fix\ h)$, which by fixpoint induction follows from the assumptions $P\ \bot$ and $\forall g.\ P\ g \Rightarrow P\ (h\ g)$. Finally, these two assumptions can be verified by the following simple calculations:

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

and

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

Note that fixpoint induction is sound but not complete, that is, it is a valid proof method but is not always applicable. For example, if P is the chain-complete predicate "is an infinite list" and $f\ xs = 1 : xs$ then the true statement $P\ (fix\ f)$ expresses that $1 : 1 : 1 : \cdots$ is an infinite list, but cannot be proved using fixpoint induction because $\bot$ is not an infinite list and hence the base case $P\ \bot$ is false. However, for such examples one can always resort to reasoning explicitly using the definition of $fix\ f$ as the limit of a chain of approximations.

Fixpoint induction is not specific to the type of lists, but is independent of the details of the underlying type. Indeed, the only (very weak) assumption on this type is that it 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.

4. THE APPROXIMATION LEMMA

A higher-level method for proving properties of corecursive programs is the approximation lemma, a recent improvement of the better known take lemma. Recall the standard function $take\ n$ that returns the first n elements of a list:

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

For example, $take\ 3\ ones$ returns the finite list $1 : 1 : 1 : [\,]$. The approximation lemma is based upon a function $approx\ n$ that is defined in the same way as $take\ n$, except that the base 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 integers greater than 0, removing the extra base case means that, by case exhaustion, $approx\ 0\ xs = \bot$ for all lists xs. For example, $approx\ 3\ ones$ returns the partial list $1 : 1 : 1 : \bot$.

Suppose that xs and ys are two finite, partial or infinite lists. Then the *approximation lemma* [Bird 1998] is given by 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. Conversely, for the other direction it is easy to show that

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

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 of the approximation lemma can be verified as follows:

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

Returning once again to our running example, by the approximation lemma the *map-iterate* property is equivalent to the following property:

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

This property can now be verified by induction on the natural number 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 by the following calculation:

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

Unlike fixpoint induction, the approximation lemma is both sound and complete, although it is only applicable in the special (but very common) case of proving the equality of 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 of *approx* by *take* gives the take lemma [Bird and Wadler 1988], which is probably currently the most widely-used method for proving properties of corecursive programs that produce lists. The approximation lemma is equivalent to the take lemma, but is slightly simpler to prove and to apply. More importantly, however, the approximation lemma naturally generalises from lists to a large class of types, whereas the take lemma does not [Hutton and Gibbons 2000].

5. COINDUCTION

In contrast to fixpoint induction and the approximation lemma, coinduction is derived from the operational approach to semantics [Plotkin 1981]. For a pure functional language, an operational semantics can be given by defining 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 defining equations for *iterate* and *map* give rise to the following reduction rules:

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

In turn, an operational basis for reasoning about expressions can be given by defining a *labelled transition relation* $\rightarrow$, for which $a \xrightarrow{o} a'$ means that the expression a permits the *observation* o, thereby making the transition to the expression a'. For example, the following two transition rules formalise the observations that can be made of a non-empty list, namely that we can observe its head and tail:

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

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

By repeated application of a labelled transition relation, we can generate a transition tree that captures all possible paths of observations starting from a given

expression. Informally, two expression 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 the observations that they permit.

Bisimilarity is the basic notion of operational equivalence for expressions, and can be formalised as follows [Milner 1989]. Given a labelled transition relation, a *bisimulation* is a relation R on expressions such that if $a\ R\ b$ then:

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

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, denoted by $\sim$ and called *bisimilarity*, which relates as many expressions as possible subject to the two conditions above.

The basic method for proving that two expressions a and b are bisimilar is *coinduction* [Milner 1989], which is given by the following equivalence:

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

This equivalence states that two expressions are bisimilar precisely when they are related by a bisimulation. The left-to-right direction is trivially true by taking $R = \sim$. Conversely, for the other direction any bisimulation R is by definition included in the greatest bisimulation $\sim$, and hence $a\ R\ b$ implies that $a \sim b$.

As a simple application of coinduction, let us see how it can be used to prove the operational variant of the *map-iterate* property [Gordon 1995]:

$$map\ f\ (iterate\ f\ x) \sim iterate\ f\ (f\ x)$$

First of all, we define a relation S that encodes this property:

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

The relation S is not itself a bisimulation, but we will now show that $S \cup \sim$ is a bisimulation. Suppose that $a\ (S \cup \sim)\ b$, which means that either $a\ S\ b$ or $a \sim b$. If $a \sim b$, then any transition for a is matched by one for b and vice versa, because $\sim$ is by definition a bisimulation. If $a\ S\ b$, then $a = map\ f\ (iterate\ f\ x)$ and $b = iterate\ f\ (f\ x)$ for some f and x. Using the reduction rules for *iterate* and *map*, the expressions a and b can be reduced as follows:

$$a \rightsquigarrow map\ f\ (x : iterate\ f\ (f\ x)) \rightsquigarrow f\ x : map\ f\ (iterate\ f\ (f\ x))$$

$$b \rightsquigarrow f\ x : iterate\ f\ (f\ (f\ x))$$

Using these reductions, together with the transition rules for non-empty lists, we can now enumerate all the possible transitions for a and b:

$$a \xrightarrow{head} f\ x \qquad (1)$$

$$a \xrightarrow{tail} map\ f\ (iterate\ f\ (f\ x)) \qquad (2)$$

$$b \xrightarrow{head} f\ x \qquad (3)$$

$$b \xrightarrow{tail} iterate\ f\ (f\ (f\ x)) \qquad (4)$$

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 $f\ x\ (S \cup \sim)\ f\ x$. (This case shows why S itself is not a bisimulation: it is not closed under *head* observations.) Transition (2) is matched by transition (4) and vice versa, because the resulting expressions $map\ f\ (iterate\ f\ (f\ x))$ and $iterate\ f\ (f\ (f\ x))$ are related by S and so also by $S \cup \sim$. This completes the proof that $S \cup \sim$ is a bisimulation, using which result the *map-iterate* property can now be verified by coinduction as follows:

$$
\begin{array}{ll}
 & map\ f\ (iterate\ f\ x)\ \sim\ iterate\ f\ (f\ x) \\
\Leftarrow & \quad \{\ \text{coinduction; } S \cup \sim \text{ is a bisimulation}\ \} \\
 & map\ f\ (iterate\ f\ x)\ (S \cup \sim)\ iterate\ f\ (f\ x) \\
\Leftarrow & \quad \{\ \text{property of union}\ \} \\
 & map\ f\ (iterate\ f\ x)\ S\ iterate\ f\ (f\ x) \\
\Leftrightarrow & \quad \{\ \text{definition of } S\ \} \\
 & true
\end{array}
$$

Note that coinduction is both sound and complete for the problem of proving the bisimilarity of two expressions. Moreover, coinduction is not specific to the type of lists, but is independent of the details of the underlying type.

Proofs using fixpoint induction or the approximation lemma are largely a matter of mechanical calculation, while proofs using coinduction can require a greater degree of insight. In particular, we must somehow construct a bisimulation like S that relates the expressions to be proved bisimilar. For the *map-iterate* property this was straightforward, but in general this can be a non-trivial process.

At present, denotational semantics is still the dominant basis for proofs about functional programs. However, the trend in semantics research has moved towards the use of operational semantics, so we expect that the use of operational methods such as coinduction will become more widespread in the future.

6. FUSION

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

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

The *unfold* operator 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 using the same process.

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 that determines whether a list is empty or not:

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

The basic method for proving properties of programs defined using *unfold* is its *universal property* [Malcolm 1990], which is given by 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 $\mathit{unfold}\ p\ h\ t$ 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 extensionality 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 $\mathit{approx}\ 0\ xs = \bot$ for all lists xs, while the inductive case $n = m + 1$ is verified by the following calculation:

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

The utility of the universal property 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 *unfold* operator itself encapsulates a simple pattern of definition for such programs.

A corollary of the universal property that is often preferable in practice is *fusion* [Malcolm 1990], which is given by the following inference rule:

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

This inference 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 is

derived (without using induction) from the universal property as follows:

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

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

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

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

Using these general fusion laws, the *map-iterate* property can be proved more simply than using the other three proof methods that we have discussed in previous sections. First of all, by extensionality 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:

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

Like fixpoint induction, fusion is sound but not complete. However, in cases where fusion is not applicable one can often resort to reasoning explicitly using the universal property of *unfold*. Moreover, both the fusion and the universal property naturally generalise from lists to a large class of types [Malcolm 1990].

Using high-level proof methods such as 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. However, the *unfold* operator and its associated proof methods can be generalised to cover all corecursive functions [Vene and Uustalu 1998].

7. SUMMARY

To conclude, we briefly compare the four proof methods discussed above.

We believe that the *universal property* and the *fusion laws* that are its corollaries are the most convenient of the proof methods. They allow one to avoid lengthy (co-)inductive proofs in favour of concise equational reasoning, and this reasoning can be performed in terms of the syntax of the programming language instead of some underlying logic. On the other hand, the universal property is only applicable to *structured* corecursive programs, namely those written using recursion operators like *unfold*; for less amenable programs, lower-level tools must be used.

When the universal property is not applicable, but the property to be proved is an equality between two data structures, then the *approximation lemma* is always available. Like the universal property, this has the benefit of allowing reasoning to be performed at the level of the program text, but in contrast to the universal property a slightly tedious inductive proof is still required.

The other two methods discussed were *fixpoint induction* and *coinduction*; we feel that they should be used only as a last resort. In their favour one can say that each is a single complete rule, universally applicable and widely known. But one could say the same thing about doing differential calculus in terms of the limit as interval size tends to zero of the average gradient over that interval: one soon feels the need for higher-level abstractions, however specialized they may be. The two methods are of comparable complexity and operate at comparable levels of abstraction, but the former is based on a denotational view of semantics and the latter on an operational view; the choice between them should be made on the basis of which underlying semantics is most appropriate in the circumstances.

REFERENCES

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

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

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

Bird, R. and Wadler, P. 1988. *An Introduction to Functional Programming*. Prentice Hall.

Burge, W. 1975. *Recursive Programming Techniques*. Addison-Wesley.

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

de Bakker, J. 1980. *Mathematical Theory of Program Correctness*. Prentice-Hall.

Freyd, P. 1990. Algebraically complete categories. In Carboni, A. et al Ed., *Proc. 1990 Como Category Theory Conference*, Volume 1488 of *Lecture Notes in Math* (1990), pp. 95–104. Springer-Verlag, Berlin.

HUTTON, G. AND GIBBONS, J. 2000. The generic approximation lemma. Submitted for publication.

GORDON, A. 1995. Bisimilarity as a theory of functional programming. *Proc. of the Eleventh Conference on the Mathematical Foundations of Programming Semantics* (1995). Elsevier Science B.V. Electronic Notes in Theoretical Computer Science Volume 1.

JACOBS, B., MOSS, L., REICHEL, H., AND RUTTEN, J. Eds. 1998. *Proc. of the First Workshop on Coalgebraic Methods in Computer Science* (1998). Elsevier Science B.V. Electronic Notes in Theoretical Computer Science Volume 11.

JACOBS, B. AND RUTTEN, J. Eds. 1999. *Proc. of the Second Workshop on Coalgebraic Methods in Computer Science* (1999). Elsevier Science B.V. Electronic Notes in Theoretical Computer Science Volume 19.

MALCOLM, G. 1990. *Algebraic Data Types and Program Transformation*. Ph. D. thesis, Groningen University.

MILNER, R. 1989. *Communication and Concurrency*. Prentice Hall.

MOSS, L. AND DANNER, N. 1997. On the foundations of corecursion. *Logic Journal of the IGPL 5*, 2, 231–257.

PEYTON JONES, S. ET AL. 1999. Haskell 98: A non-strict, purely functional language. Available from `http://www.haskell.org`.

PLOTKIN, G. 1981. A structured approach to operational semantics. Report DAIMI-FN-19, Computer Science Department, Aarhus University, Denmark.

REYNOLDS, J. C. 1998. *Theories of Programming Languages*. Cambridge University Press.

SCHMIDT, D. A. 1986. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc.

VENE, V. AND UUSTALU, U. 1998. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics 47*, 3, 147–161.