# The generic approximation lemma

Graham Hutton
University of Nottingham

Jeremy Gibbons
University of Oxford

January 2000

**Abstract**

The *approximation lemma* is a simplification of the well-known *take lemma*, and is used to prove properties of programs that produce lists of values. We show how the approximation lemma, unlike the take lemma, can naturally be generalised from lists to a large class of datatypes, and present a *generic* approximation lemma that is parametric in the datatype to which it applies. As a useful by-product, we find that generalising the approximation lemma in this way also simplifies its proof.

*Keywords*: Programming calculi; Functional Programming

## 1 Introduction

The standard proof method for programs that *consume* lists of values is structural induction. However, this method is not applicable to the dual case of programs that *produce* lists of values, because in general such programs do not have a list argument over which to perform induction. Proof methods that are applicable to such programs have recently been surveyed in [6], and include *fixpoint induction* [4], the *take lemma* [3], *coinduction* [7], and *fusion* [8].

All but one of the above proof methods for programs that produce lists are not specific to the datatype of lists, but can naturally be generalised to a large class of other datatypes. The exception is the take lemma, which is formulated specifically for lists, and has not proved possible to generalise in a uniform way to other datatypes. This is unfortunate, because the take lemma is perhaps the most widely-used proof method for programs that produce lists.

Recently, the take lemma has been superseded by the *approximation lemma* [2], which is equivalent in power but is simpler to prove and apply. We show how the approximation lemma, unlike the take lemma, can be generalised in a uniform way to other datatypes, and present a *generic* approximation lemma that is parametric in the datatype to which it applies. The generic lemma is proved once, for an arbitrary datatype, and can then simply be instantiated as required for each new datatype. As a useful by-product, we find that generalising the approximation lemma in this way also simplifies its proof.

The programming paradigm used in the article is that of a pure functional language with non-strict semantics, such as SASL, Miranda, Gofer, Lazy ML, or Haskell. Using a pure functional language permits proofs by simple equational reasoning, while the use of non-strict semantics permits a natural treatment of infinite structures, such as infinite lists (streams) and infinite trees.

## 2 The approximation lemma

Suppose that the empty list is denoted by $[\,]$, while non-empty lists are constructed using an infix operator $(:)$ that prepends a value to the start of a list. Now recall the standard list-processing function $take\ n$ that returns the first $n$ elements of a list, defined recursively as follows:

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

For example, if the infinite list $ones = 1:1:1:1:\ldots$ is defined by $ones = 1 : ones$, then $take\ 3\ ones$ evaluates to 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:

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

Because $n+1$ patterns only match integers greater than 0, removing the extra base case means that, by case exhaustion, $approx\ 0\ xs = \perp$ for all lists $xs$, where $\perp$ represents an undefined value. For example, $approx\ 3\ ones$ evaluates to the partial list $1:1:1:\perp$ that ends with $\perp$ rather than with $[\,]$.

Suppose that $xs$ and $ys$ are two finite, partial or infinite lists. Then the approximation lemma [2] 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, as produced by $approx$, are equal. Replacing the use of $approx$ by $take$ gives the well-known take lemma [3], which is formally equivalent to the approximation lemma. However, the approximation lemma supersedes the take lemma, in the sense that it is simpler to prove and apply.

As a simple application of the approximation lemma, consider the standard functions $iterate$ and $map$ defined recursively as follows:

$$\begin{array}{lcl} iterate\ f\ x & = & x : iterate\ f\ (f\ x) \\[4pt] map\ f\ [\,] & = & [\,] \\ map\ f\ (x:xs) & = & f\ x : map\ f\ xs \end{array}$$

Then by the approximation lemma, the familiar property

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

is equivalent to

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

which can be verified by a routine induction on the natural number $n$. Further applications and a proof of the approximation lemma are given in [2].

## 3 Generalising the approximation lemma

The function *approx* is not specific to lists, but can naturally be defined for many other datatypes. The general pattern is that *approx* $n$ is defined in the same way as the recursive identity function for a datatype, except that the numeric argument $n$ is decremented at each recursive call. For example, assuming that new datatypes are defined using a BNF-like syntax, the function *approx* can be defined for a datatype *Tree* of binary trees of integers,

$$Tree \ ::= \ Leaf \ \mid \ Node\ Tree\ Int\ Tree$$

$$\begin{array}{lcl} approx\ (n+1)\ Leaf & = & Leaf \\ approx\ (n+1)\ (Node\ l\ x\ r) & = & Node\ (approx\ n\ l)\ x\ (approx\ n\ r) \end{array}$$

and for a datatype *Expr* of simple arithmetic expressions:

$$Expr \ ::= \ Const\ Int \ \mid \ Add\ Expr\ Expr \ \mid \ Mult\ Expr\ Expr$$

$$\begin{array}{lcl} approx\ (n+1)\ (Const\ x) & = & Const\ x \\ approx\ (n+1)\ (Add\ l\ r) & = & Add\ (approx\ n\ l)\ (approx\ n\ r) \\ approx\ (n+1)\ (Mult\ l\ r) & = & Mult\ (approx\ n\ l)\ (approx\ n\ r) \end{array}$$

Note that the function *take* $n$ can also be defined for the datatype *Tree*, with *take* $0\ t = Leaf$, but cannot be defined for *Expr* because this datatype does not provide a nullary constructor (such as *Leaf*) to return in the case $n = 0$. The function *approx* $n$ avoids this problem by simply removing the case for $n = 0$, which by case exhaustion means that *approx* $0\ x = \perp$ for any value $x$.

Similarly, the approximation lemma itself is not specific to lists, but can naturally be formulated and proved for any datatype for which the function *approx* can be defined. For example, for either of the datatypes *Tree* and *Expr* defined above, the approximation lemma is given by the following equivalence, which has a proof similar to that outlined in [2] for lists:

$$x \ = \ y \quad \Leftrightarrow \quad \forall n.\ approx\ n\ x \ = \ approx\ n\ y$$

While it is straightforward to redefine *approx* and reprove the approximation lemma for new datatypes, repeating similar definitions and (particularly) proofs again and again is tedious, time consuming and prone to error. The solution is to define a *generic* version of *approx* that is parametric in the datatype to which it applies, and then prove a generic version of the approximation lemma based upon this definition. In this way, the function *approx* and the approximation lemma are defined and proved once, for an arbitrary datatype, and can then simply be instantiated as required for each new datatype.

# 4 The generic approximation lemma

The generic approximation lemma is founded upon the standard denotational approach to the semantics of functional languages [9]. In particular, we assume that types are *cpos* (partially-ordered sets with a least element $\perp$ and limits of non-empty chains) and programs are *continuous functions* (functions between cpos that are monotonic and preserve the limit structure). The key to the generic approximation lemma itself is to define recursive datatypes as *least fixpoints of functors*, a standard technique in generic programming [1].

Recall that a *functor* is a mapping $F$ that takes types to types and functions to functions, such that $F$ preserves function typings, identity functions, and the composition of functions. The appropriate notion of a *fixpoint* for a functor $F$ is a type $A$ for which $F A$ is isomorphic to $A$, in the sense that there exist two functions $in : F A \to A$ and $out : A \to F A$ that are each other's inverses. A functor is called *locally continuous* if its mapping on functions is itself continuous. A standard fixpoint theorem [10, 5] states that every locally continuous functor $F$ on cpos and continuous functions has a unique (up to isomorphism) fixpoint $A$ for which the identity function $id : A \to A$ is the unique solution to the recursive equation $f = in \cdot F f \cdot out$. This unique fixpoint is called the *least fixpoint* of the functor $F$, and is denoted by $\mu F$.

A large class of recursive datatypes can be defined as least fixpoints of locally continuous functors. In particular, all polynomial datatypes — for example, any sum-of-products datatype — can be defined in this way. This result generalises to mutually recursive, parameterised, exponential and nested datatypes, but for simplicity we only consider polynomial datatypes in this article.

For example, our datatype of binary trees can be defined by $Tree = \mu T$, where the functor $T$ is defined on types and functions as follows:

$$T\ A\ ::=\ Leaf\ \mid\ Node\ A\ Int\ A$$

$$\begin{array}{lcl} T\ f\ Leaf & = & Leaf \\ T\ f\ (Node\ a\ x\ b) & = & Node\ (f\ a)\ x\ (f\ b) \end{array}$$

Similarly, $Expr = \mu E$, where the functor $E$ is defined by:

$$E\ A\ ::=\ \mathit{Const}\ \mathit{Int}\ \mid\ \mathit{Add}\ A\ A\ \mid\ \mathit{Mult}\ A\ A$$

$$\begin{array}{lcl}
E\ f\ (\mathit{Const}\ x) & = & \mathit{Const}\ x \\
E\ f\ (\mathit{Add}\ a\ b) & = & \mathit{Add}\ (f\ a)\ (f\ b) \\
E\ f\ (\mathit{Mult}\ a\ b) & = & \mathit{Mult}\ (f\ a)\ (f\ b)
\end{array}$$

By the fixpoint theorem for locally continuous functors $F$, a generic identity function $id$ for an arbitrary datatype $\mu F$ can be defined recursively by $id = in \cdot F\ id \cdot out$. Hence, using our intuition that $approx\ n$ is defined in the same way as the recursive identity function except that the numeric argument $n$ is decremented at each recursive call, a generic version of $approx$ for an arbitrary datatype $\mu F$ can now be defined as follows:

$$approx\ (n+1) \quad = \quad in \cdot F\ (approx\ n) \cdot out$$

Instantiating this definition to any specific datatype $\mu F$ gives the expected definition. For example, for the datatype $\mathit{Tree} = \mu T$ of binary trees, with constructors defined by $\mathit{leaf} = in\ \mathit{Leaf}$ and $node\ l\ x\ r = in\ (\mathit{Node}\ l\ x\ r)$, a simple calculation shows that the generic definition of $approx$ (written as $app$ below for conciseness) is equivalent to the specific definition for $\mathit{Tree}$:

$$\begin{array}{ll}
 & app\ (n+1)\ =\ in \cdot T\ (app\ n) \cdot out \\
\Leftrightarrow & \quad \{\ in \text{ and } out \text{ are inverses }\} \\
 & app\ (n+1) \cdot in\ =\ in \cdot T\ (app\ n) \\
\Leftrightarrow & \quad \{\ \text{composition, case analysis }\} \\
 & app\ (n+1)\ (in\ \mathit{Leaf}) \qquad\qquad\quad = \ in\ (T\ (app\ n)\ \mathit{Leaf}) \\
 & app\ (n+1)\ (in\ (\mathit{Node}\ l\ x\ r))\ =\ in\ (T\ (app\ n)\ (\mathit{Node}\ l\ x\ r)) \\
\Leftrightarrow & \quad \{\ \text{definition of } T\ \} \\
 & app\ (n+1)\ (in\ \mathit{Leaf}) \qquad\qquad\quad = \ in\ \mathit{Leaf} \\
 & app\ (n+1)\ (in\ (\mathit{Node}\ l\ x\ r))\ =\ in\ (\mathit{Node}\ (app\ n\ l)\ x\ (app\ n\ r)) \\
\Leftrightarrow & \quad \{\ \text{definition of } \mathit{leaf} \text{ and } node\ \} \\
 & app\ (n+1)\ \mathit{leaf} \qquad\qquad = \ \mathit{leaf} \\
 & app\ (n+1)\ (node\ l\ x\ r)\ =\ node\ (app\ n\ l)\ x\ (app\ n\ r)
\end{array}$$

Prior to stating and proving the generic approximation lemma itself, we present two properties of the generic function $approx$. The first of these properties states that approximation functions form a chain,

$$\forall n.\ approx\ n\ \sqsubseteq\ approx\ (n+1)$$

and can be proved by induction on the natural number $n$. The base case $n = 0$ is trivially true because, by case exhaustion, $approx\ 0\ x = \perp$ for all $x$. For the inductive case $n = m + 1$, we calculate as follows:

$$
\begin{array}{cl}
 & approx\ (m+1) \\
= & \quad \{ \text{ definition of } approx \ \} \\
 & in \cdot F\ (approx\ m) \cdot out \\
\sqsubseteq & \quad \{ \text{ induction hypothesis } \} \\
 & in \cdot F\ (approx\ (m+1)) \cdot out \\
= & \quad \{ \text{ definition of } approx \ \} \\
 & approx\ (m+2)
\end{array}
$$

As well as being generic, the above proof is simpler than the corresponding proof for the special case of lists [2], which requires a (non-inductive) case analysis on lists in addition to the use of induction on natural numbers.

The second property states that the limit of the chain of approximation functions predicted by the first property is the identity function:

$$\bigsqcup_n \{approx\ n\} \ = \ id$$

To prove this property, we exploit the fact that the identify function for an arbitrary datatype $\mu F$ is, by the fixpoint theorem, the unique solution to the equation $f \ = \ in \cdot F\,f \cdot out$. Hence, the above equation is equivalent to

$$\bigsqcup_n \{approx\ n\} \ = \ in \cdot F\ (\bigsqcup_n \{approx\ n\}) \cdot out$$

which we can then verify by the following calculation:

$$
\begin{array}{cl}
 & \bigsqcup_n \{approx\ n\} \\
= & \quad \{ \text{ separating out } n = 0 \ \} \\
 & approx\ 0 \ \sqcup \ \bigsqcup_n \{approx\ (n+1)\} \\
= & \quad \{ \text{ definition of } approx \ \} \\
 & \lambda x.\bot \ \sqcup \ \bigsqcup_n \{in \cdot F\ (approx\ n) \cdot out\} \\
= & \quad \{ \ \lambda x.\bot \text{ is the unit for } \sqcup \text{ on functions } \} \\
 & \bigsqcup_n \{in \cdot F\ (approx\ n) \cdot out\} \\
= & \quad \{ \text{ continuity of } (in\,\cdot) \text{ and } (\cdot\, out) \ \} \\
 & in \cdot \bigsqcup_n \{F\ (approx\ n)\} \cdot out \\
= & \quad \{ \text{ local continuity of } F \ \} \\
 & in \cdot F\ (\bigsqcup_n \{approx\ n\}) \cdot out
\end{array}
$$

Again, the above proof is simpler than the standard proof for the special case of lists [2], which makes use of structural induction on lists. Indeed, by exploiting the fact that the identity function is the unique solution to a certain equation, we have completely avoided the use of any form of induction!

Finally, given an arbitrary locally continuous functor $F$, the generic approximation lemma for the datatype $\mu F$ is stated as follows:

$$x \ = \ y \quad \Leftrightarrow \quad \forall n.\ approx\ n\ x \ = \ approx\ n\ y$$

The $\Rightarrow$ direction is trivially true by the substitutivity property of pure functional languages, which states that applying a function to equal arguments gives equal results. Conversely, the $\Leftarrow$ direction is a simple consequence of our two properties of *approx* and the substitutivity property for limits:

$$
\begin{array}{ll}
 & x \;=\; y \\
\Leftrightarrow & \quad \{ \text{ definition of } id \text{ } \} \\
 & id\ x \;=\; id\ y \\
\Leftrightarrow & \quad \{ \text{ properties of } approx \text{ } \} \\
 & \bigsqcup_n \{approx\ n\}\ x \;=\; \bigsqcup_n \{approx\ n\}\ y \\
\Leftrightarrow & \quad \{ \text{ continuity of application } \} \\
 & \bigsqcup_n \{approx\ n\ x\} \;=\; \bigsqcup_n \{approx\ n\ y\} \\
\Leftarrow & \quad \{ \text{ limits } \} \\
 & \forall n.\ approx\ n\ x \;=\; approx\ n\ y
\end{array}
$$

Applications of instances of the generic approximation lemma abound. We conclude this section with an application of the generic approximation lemma itself. Given an arbitrary locally continuous functor $F$, consider the following definition for a generic function *unfold* for an arbitrary datatype $\mu F$:

$$unfold\ g \quad = \quad in \cdot F\ (unfold\ g) \cdot g$$

This function forms the basis of a simple but powerful calculus for defining and reasoning about programs that produce values of recursive datatypes [8, 6]. The calculus is founded on the fact *unfold* $g$ is not just the least solution to its defining equation, but is in fact the unique solution. That is, we have the following *universal property* for the generic function *unfold*:

$$h \;=\; unfold\ g \quad \Leftrightarrow \quad h \;=\; in \cdot F\, h \cdot g$$

This property is normally proved using fixpoint induction. However, a simpler proof is possible using the generic approximation lemma. The $\Rightarrow$ direction is trivially true because substituting $h = unfold\ g$ into the right-hand side gives the definition for *unfold* $g$. Conversely, in the $\Leftarrow$ direction we have:

$$
\begin{array}{ll}
 & h \;=\; unfold\ g \\
\Leftrightarrow & \quad \{ \text{ extensionality } \} \\
 & \forall x.\ h\ x \;=\; unfold\ g\ x \\
\Leftrightarrow & \quad \{ \text{ generic approximation lemma } \} \\
 & \forall x.\ \forall n.\ approx\ n\ (h\ x) \;=\; approx\ n\ (unfold\ g\ x) \\
\Leftrightarrow & \quad \{ \text{ composition, extensionality } \} \\
 & \forall n.\ approx\ n \cdot h \;=\; approx\ n \cdot unfold\ g
\end{array}
$$

The final line above can now be verified by a routine induction on the natural number $n$, using the assumption that $h = in \cdot F\, h \cdot g$.

## 5 Summary and conclusions

We have shown how a useful proof method for lists can be generalised to a generic proof method that is parametric in the datatype to which it applies. This work is part of a continuing effort in programming language research to increase the reusability of programs and proofs by abstracting away from unnecessary details, in this case the details of the underlying datatype.

## Acknowledgements

Thanks to Roland Backhouse for suggestions that improved the presentation of the paper, and to Paul Blampied for technical expertise regarding the fixpoint theorem. The first author is supported by EPSRC grant *Structured Recursive Programming* and ESPRIT Working Group *Applied Semantics*.

## References

[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Advanced Functional Programming*, LNCS 1608, pages 28–115. Springer-Verlag, 1999.

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

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

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

[5] M. Fokkinga and E. Meijer. Program calculation properties of continuous algebras. Technical Report 91-4, Centre for Mathematics and Computer Science (CWI), Amsterdam, 1991.

[6] J. Gibbons and G. Hutton. Proof methods for structured corecursive programs. In *Proceedings of the 1st Scottish Functional Programming Workshop*, Stirling, Scotland, Aug. 1999.

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

[8] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proc. Conference on Functional Programming and Computer Architecture*, number 523 in LNCS. Springer-Verlag, 1991.

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

[10] M. B. Smyth and G. D. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journal of Computing*, pages 761–783, 1982.