Computer Science
University of Manchester

Generalisation for Induction

Sunil Vadera

Technical Report UMCS–93–6–8

Generalisation for Induction

Sunil Vadera [1]

July 22, 1993

Abstract

Proof by induction plays a central role in showing that recursive programs satisfy their specification. Sometimes a key step is to generalise a lemma so that its inductive proof is easier. This report examines existing heuristics for *generalisation for induction*.

We first examine the applicability of heuristics for generalisation and then develop a new approach for a class of problems for which the existing methods fail. Unlike the existing methods, we give the conditions under which our method guarantees a valid generalisation.

[1] My sincere thanks to Tim Clement for his help and encouragement of this research. Thanks are also due to Richard Moore and Bob Fields for help with *mural* as it was being developed.

1

An Introduction to Generalisation

1.1 Motivation

Proof by induction proceeds in two stages. First, the base case is proved, and then an induction step is proved. The base case is normally easily established. The induction step proceeds by replacing function applications by the recursive part(s) of their definition (i.e., by unfolding) and relies on the fact that the induction hypothesis can be used at some stage during the proof. Sometimes, the unfolding process will not yield a property in a form that can utilise the induction hypothesis.

Example: fact

As an example, consider how one might attempt to prove that the following two functions, which are intended to compute the factorial function, are equivalent.

$$\begin{array}{l} f : \mathbb{N} \to \mathbb{N} \\ f(x) \quad \triangleq \quad \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ f(x-1) * x \end{array}$$

$$\begin{array}{l} g : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ g(x, y) \quad \triangleq \quad \mathsf{if}\ x = 0\ \mathsf{then}\ y\ \mathsf{else}\ g(x-1, x * y) \end{array}$$

That is, we need to prove:

$$g(x, 1) = f(x)$$

The first step in carrying out an inductive proof is to select an appropriate induction rule. In this report, we do not discuss the problem of selecting an appropriate induction scheme automatically. Instead we select an appropriate rule where necessary. For this problem, suppose we use the following induction rule ([Jon90]):

$$\boxed{\mathbb{N}\text{-}ind}\ \frac{p(0);\quad n \in \mathbb{N}, p(n) \vdash p(n+1)}{n \in \mathbb{N} \vdash p(n)}$$

Then our proof attempt proceeds as follows.

Proof Attempt

Base Case

Since $f(0) = 1$ and $g(0, 1) = 1$ we conclude that the base case is true.

Induction Step

Assume $g(n, 1) = f(n)$ is true for an arbitrary $n \in \mathbb{N}$. We need to show that:

$$g(n+1, 1) = f(n+1).$$

Unfolding the left hand side and simplifying gives:

$$g(n, (n+1) * 1).$$

We cannot use the induction hypothesis on this expression since the second arguments of g in this expression is $(n+1) * 1$ while the second argument of the left hand side of the hypothesis requires it to be 1. Hence, our proof has been obstructed.

We can, however prove the more general result:

$$g(x, y) = f(x) * y$$

as follows.

Proof of the Generalised lemma

Base Case

Since $f(0) = 1$ and $g(0, y) = y$ we conclude that $g(0, y) = f(0) * y$.

Induction Step

Assume $g(n, y) = f(n) * y$ is true for an arbitrary $n \in \mathbb{N}$, and any $y \in \mathbb{N}$. We need to show that:

$$g(n+1, y) = f(n+1) * y.$$

Unfolding the left hand side and simplifying gives:

$$g(n, (n+1) * y).$$

Now applying the induction hypothesis, we can rewrite this to:

$$f(n) * (n+1) * y.$$

Folding $f(n) * (n+1)$ to $f(n+1)$ now enables us to obtain:

$$f(n+1) * y.$$

Hence we have carried out the induction step.

Further this can be specialised to the original lemma:

$$g(x, 1) = f(x)$$

by setting y to 1 and simplifying.

As this example illustrates, generalising a lemma can result in a lemma that is easier to prove. We now give a working definition of generalisation.

Definition of Generalisation

Generalisation is the process of going from the consideration of a set of objects to a larger set [Pol57]. Hence, we say that a lemma L_1 is a *generalisation* of a lemma L_2 if L_1 can be specialised to L_2 by restricting the set of objects under consideration by L_1. Thus in the above example, the lemma $g(x, y) = f(x) * y$ is more general than $g(x, 1) = f(x)$ because the latter is obtained by restricting the y in the former to 1 (and simplifying).

Since, by this definition a more general lemma applies in more circumstances, then when an inductive proof is attempted, the induction hypothesis should also be applicable in more cases. Further since the use of the induction hypothesis is an important step in an inductive proof, one can expect an inductive proof of a more general lemma to be easier than that of a more specialised lemma.

The notion of an *easier* proof is a subjective one. Nevertheless, we think of a solution to a problem being easier if it follows a path that we would normally expect and which has been followed in solving many other problems. In our study of generalisation the majority of lemmas that we have encountered have a natural left and right hand side that are related by a relational operator. In this context, we would expect an inductive proof to follow the steps (for the induction step):

1. Take one side of a property that contains the recursive function and unfold the recursive function.

2. Simplify and hope to apply the induction hypothesis.

3. Simplify and hope to fold in order to obtain the other side of the property.

The extent to which one would expect to carry out an induction step in this manner, is based on the fact that most introductions to induction give examples which fit into that category; presumably because they are considered easier. However, it is not a precise measure, but only a guide which we consider more appropriate than counting the number of lines in a proof.

Hence, we can obtain an indication of the difficulty of a proof by examining the extent to which a proof by induction departs from this scheme. Two situations where a lemma, L_1, is easier to prove than another lemma L_2 are:

- L_1's induction hypothesis is usable but L_2's is not. That is, when we attempt a proof of L_2, we find that within step 2 above, we are obstructed from using the induction hypothesis. For example, in our earlier example we were unable to use the hypothesis on $g(n, (n+1) * 1)$.

- L_1's induction hypothesis is strong enough to prove the result but L_2's is not. That is, when we attempt a proof of L_2, we find that although we can use the induction hypothesis in step 2, we are unable to complete step 3.

Several authors have proposed heuristics for generalisation. We describe and examine the following heuristics:

- Generalisation of minimal common subterms.
- Generalisation after unfolding.
- Generalisation after cross-fertilisation.
- Generalisation of primary terms.
- Generalisation of accumulator constants.

1.2 Generalisation of Minimal Common Subterms

Boyer and Moore's theorem proving system [BM79] is the first notable system to provide a heuristic for generalisation. They use a heuristic that the smallest subterm (excluding variables) that occurs on both sides of a lemma could be generalised to a variable. Such a subterm is known as the *minimal common subterm.*

Example: append-rev

Given the following functions:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{reverse : X^* \rightarrow X^*} \\
reverse(l) & \triangleq & \textsf{if } l = [\,] \\
 & & \textsf{then } [\,] \\
 & & \textsf{else } append(reverse(\textsf{tl}\, l), [\textsf{hd}\, l])
\end{array}
$$

where

$$
\begin{array}{lcl}
\multicolumn{3}{l}{append : X^* \times X^* \rightarrow X^*} \\
append(a, b) & \triangleq & \textsf{if } a = [\,] \\
 & & \textsf{then } b \\
 & & \textsf{else } (\textsf{cons}(\textsf{hd}\, a, append(\textsf{tl}\, a, b)))
\end{array}
$$

show that:

$$append(reverse(x), append(y, z)) = append(append(reverse(x), y), z).$$

The minimal common subterm is $reverse(x)$, and one therefore obtains the following generalisation:

$$append(w, append(y, z)) = append(append(w, y), z).$$

Boyer and Moore recognize that such a generalisation may lose an important aspect of the lemma and result in an invalid formula. For example, applying the heuristic to:

$$sort(sort(l)) = sort(l)$$

where $sort$ is some stable sorting scheme, would give the incorrect formula:

$$sort(x) = x.$$

To cater for such problems, their theorem proving system requires the user to mark those lemmas that are important properties that should not be lost. A property whose argument matches the minimal common subterm is then added as an assumption to the lemma before generalisation. Thus if the following lemma was marked:

$$ordered(sort(x))$$

then it would be added as an assumption:

$$ordered(sort(l)) \;\Rightarrow\; sort(sort(l)) = sort(l)$$

before the generalisation of $sort(l)$ to give:

$$ordered(x) \;\Rightarrow\; sort(x) = x.$$

1.3 Generalisation after Unfolding

Boyer and Moore's approach generalises a lemma without using any information about the definition of the functions in the lemma. In contrast, Castaing, Kodratoff, and Degano [CKD83] use an approach that proceeds by unfolding a function definition. The approach, which we label CKD-1, can be summarised as:

1. Choose the induction variable.
2. Unfold all function definitions once.
3. Generalise any constants and variables excluding those argument positions that correspond to the position of the induction variable. The generalisation performed is a wild generalisation in that a new variable is introduced for each argument position that is generalised. Replace the induction argument positions by the same new variable.
4. In general, the result of the preceding step will be an invalid lemma. This step aims to obtain the relationships between the new variables. These relationships are then used to carry out substitutions on the expression obtained in step 3 in order to compensate for the wild generalisation. The relationships between the new variables may be obtained by tracing the generalised relationship with the induction variable instantiated appropriately.

Example: fact

For our introductory example, the generalisation of $g(x, 1) = f(x)$ is obtained as follows:

1. The induction variable in $g(x,1) = f(x)$ is x.
2. Unfolding once gives the relationship:

$$g(x-1, x*1) = f(x-1)*x$$

3. Now generalising gives:

$$g(x, x2*x3) = f(x)*x4 \tag{1.1}$$

4. Obtaining relationships:

   Taking x as 0 and evaluating the functions gives:

$$x2*x3 = 1*x4 \tag{1.2}$$

   Taking x as 1 and evaluating the functions also gives:

$$1*x2*x3 = 1*1*x4 \tag{1.3}$$

   At this stage, given that after simplification, one obtains the same relationship in 1.2 and 1.3, Castaing et al. advise that we should stop and utilise the relationship. Thus substituting this relationship back into equation 1.1 gives the generalisation:

$$g(x, x4) = f(x)*x4$$

The proof of this more general lemma then follows as given in the introduction.

Castaing, Kodratoff, and Degano give no suggestion for how the induction variable could be identified. They also admit that in general the problem of obtaining and solving the equations is a difficult one. Hence it would be difficult to automate this method in a theorem proving environment.

1.4 Generalisation after Cross-fertilization

When method CKD-1 fails to produce a suitable generalisation, Castaing, Kodratoff, and Degano [CKD83] suggest that the following method (which we label CKD-2) should be attempted:

1. Unfold the definitions once. Assuming that the induction variable of at least one of the functions is reduced, one should then use the equality in the hypothesis and produce an equation of one function. This process of using an equality in the hypothesis to remove a term from a goal is known as *cross-fertilisation* [BM79].[1] The choice of which function should be eliminated is left to the user.

2. Generalise the equation of one function as in CKD-1. Discover the relationship between the variables, and use it to simplify the equation.

3. Prove the simplified equation.

4. Prove the original lemma with the aid of the simplified equation.

Example: fibo

Consider the following functions which are both intended to compute the n^{th} fibonacci number:

$$
\begin{array}{lll}
\mathit{fib} : \mathbb{N} \to \mathbb{N} & & \\
\mathit{fib}(x) & \triangleq & \textsf{if } (x = 0) \textsf{ or } (x = 1) \\
 & & \textsf{then } 1 \\
 & & \textsf{else } \mathit{fib}(x-1) + \mathit{fib}(x-2)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{fibo} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} & & \\
\mathit{fibo}(x, z) & \triangleq & \textsf{if } (x = 0) \textsf{ or } (x = 1) \\
 & & \textsf{then } z + 1 \\
 & & \textsf{else } \mathit{fibo}(x-1, \mathit{fibo}(x-2, z))
\end{array}
$$

Suppose the goal is to show that:

$$\mathit{fibo}(x, 0) = \mathit{fib}(x).$$

Following the steps given:

[1] This step and term is based on a heuristic proposed by Boyer and Moore [BM79]. In their system, they propose that using cross-fertilisation, and then forgetting the equality in the hypothesis can result in a stronger lemma.

1. Unfolding the definitions once and assuming $x > 1$:

$$fibo(x-1, fibo(x-2, 0)) = fib(x-1) + fib(x-2)$$

   Now assuming the result is true for x-1, and x-2 we obtain an equation of one function:

$$fibo(x-1, fibo(x-2, 0)) = fibo(x-1, 0) + fibo(x-2, 0) \tag{1.4}$$

2. Generalising:

$$fibo(x\text{-}1, fibo(x-2, x2)) = fibo(x-1, x3) + fibo(x-2, x4) \tag{1.5}$$

3. Tracing this equation for $x = 2$ we have:

$$fibo(1, x2+1) = x3 + 1 + x4 + 1$$

   Simplifying further gives:

$$x2 + 2 = x3 + x4 + 2.$$

   We now substitute $x3 + x4$ for $x2$ in equation 1.5 to obtain:

$$fibo(x-1, fibo(x-2, x3+x4)) = fibo(x-1, x3) + fibo(x-2, x4)$$

Now, if we can prove this equation, we can use it to prove equation 1.4 by substituting 0 for $x3$ and $x4$. Further, since the base case for the original lemma holds, the original lemma would follow by induction. Thus, unlike CKD-1, this method normally requires two inductive proofs once the generalisation has been obtained.

1.5 Generalisation of Primary Terms

Boyer and Moore's approach considers all the subterms of a lemma as potentially generalisable. Aubin [Aub79], however, constrains the subterms to be only the primary terms of the lemma. The *primary terms* are defined as those terms that are encountered in a call-by-need evaluation of the lemma. The call-by-need [M$^+$73] evaluation is defined as those terms that are needed in a leftmost-outermost evaluation. That is, when possible we unfold the left-most function symbol. If we are unable to unfold the left-most symbol, then we repeat the process only on the arguments that are needed (i.e., those terms that occur as the condition of an if or at the head of case). The process stops on variables, or constants.

We can now summarise Aubin's approach, which we label Aubin-1, by the following steps:

1. Determine the primary terms that occur on the left and right hand sides of the lemma.
2. Select a primary term which occurs on both sides of the lemma.
3. Replace those primary terms in the lemma that are equal to the selected primary term by a new variable. The same new variable is used, so that unlike the approach taken in CKD-1, this step does not perform a wild generalisation.

Aubin recognises that several candidates may exist in the second step. Hence trial and error may be needed in making the selection. Aubin also recognises that an invalid generalisation may be produced, and suggests that the generalisations should be tested with examples before proceeding to carry out a proof.

Example: append-rev

Consider the lemma we gave earlier:

$$append(reverse(x), append(y, z)) = append(append(reverse(x), y), z)$$

where the functions *append*, and *reverse* are as defined earlier. Aubin-1 proceeds as follows:

1. The primary terms on the left hand side are:

$$\{reverse(x), x\}.$$

   The primary terms on the right hand side are:

$$\{append(reverse(x), y), reverse(x), x\}.$$

2. The common terms are: $\{reverse(x), x\}$.

3. It is possible that we may select the variable x as the term to be generalised. However, in this example, this simply results in the renaming of a variable without making the lemma any easier to prove. Hence, selecting the term $reverse(x)$ gives a generalisation:

$$append(append(v, y), z) = append(v, append(y, z))$$

It is perhaps worth emphasising that this approach marks the primary terms to be replaced. That is, we do not simply generalise all the terms equivalent to the selected term.

Example: append-same

So for example, the lemma:

$$append(j, append(j, j)) = append(append(j, j), j)$$

generalises to:

$$append(k, append(j, j)) = append(append(k, j), j)$$

since only the first and fourth occurrence of j are primary.

This example also illustrates the advantage of this method over Boyer and Moore's approach. The generalisation of minimal common subterms would result in the following invalid generalisation that claims that *append* is commutative:

$$append(j, x) = append(x, j).$$

1.6 Generalisation of Constants

Aubin's method of generalising primary terms does not work for problems like the fibo example that was presented above. In recognition of this, Aubin [Aub79] suggests that such lemmas can often be massaged so that the constant occurs on both sides of the equality. The generalisation can then be obtained by replacing the constants with a variable. We label this method Aubin-2.

Example: fibo

$$fibo(x, 0) = fib(x)$$

can be massaged to:

$$fibo(x, 0) = fib(x) + 0$$

which can be generalised to:

$$fibo(x, m) = fib(x) + m.$$

Aubin does not elaborate on this process of massaging. In this example, if one interprets massaging to mean the use of equality substitution with:

$$y = y + 0$$

then the following, incorrect form could also have resulted:

$$fibo(x, 0) = fib(x + 0)$$

$$fibo(x, m) = fib(x + m).$$

Aubin does, however, suggest that the generalisation heuristics may result in an invalid lemma, and that counter examples may be needed to eliminate such cases.

In summary, we have described several methods of generalisation for induction in this chapter. The next chapter includes an analysis of their applicability.

2

A New Method for Generalisation

This chapter develops a new method for the generalisation of properties of an important class of functions.

It begins by analysing the applicability of the existing methods that we described in the previous chapter. This analysis provides a better understanding of the applicability of existing methods and provides the motivation for the development of a new method that is more appropriate for an important class of generalisation problems.

2.1 Analysis of Existing Methods

In order to evaluate the applicability of the existing methods, we have tested them on several examples. Most of the examples are translated from the literature.

Table 2.1 summarises the results of applying the methods to the examples given in appendix A. A tick in the table indicates that the method works on the example, and a cross means that the method fails. A delta marks those situations in which some additional information is required for a method to succeed. Thus the delta's in the Aubin-2 column are present since it requires a process of massaging, and the delta in the BM column marks an occasion when an invariant property needs to be specified prior to generalisation. The table shows that:

- there is some overlap between the methods;

Example	C-K-D-1	C-K-D-2	Aubin-1	Aubin-2	BM
fact	$\surd$	$\surd$	$\times$	∂	$\times$
fibo	$\times$	$\surd$	$\times$	∂	$\times$
exp3	$\surd$	$\surd$	$\times$	∂	$\times$
append-same	$\times$	$\times$	$\surd$	$\times$	$\times$
reverse	$\surd$	$\surd$	$\times$	∂	$\times$
append-rev	$\times$	$\times$	$\surd$	$\times$	$\surd$
length	$\times$	$\times$	$\times$	$\times$	∂
gcd	$\times$	$\times$	$\times$	$\times$	$\times$
nines	$\times$	$\times$	$\times$	$\times$	$\times$
nsquareprop	$\times$	$\times$	$\times$	∂	$\times$
elemsofrev	$\times$	$\times$	$\times$	$\times$	$\times$
rangebt	$\times$	$\times$	$\times$	$\times$	$\times$

Table 2.1: Applicability of methods on examples

- none of the methods is always successful;
- there are examples for which none of the methods work.

It would therefore be naive simply to provide all these heuristics in a theorem proving environment. Hence, our aim now is to gain a better understanding of the applicability of these methods in order to identify a good subset of the heuristics that will cover most of the examples.

The first point to note from this table is that three of the methods, namely CKD-1, CKD-2, and Aubin-2 are most successful when generalising lemmas that express the equivalence between an accumulator version of a function and a divide and conquer version of a function. For example, these three methods succeed in generalising:

$$reverse(l) = reversea(l, [])$$

where

$$\begin{array}{lcl} reverse : X^* \to X^* & & \\ reverse(l) & \triangleq & \mathsf{if}\ l = [\,] \\ & & \mathsf{then}\ [\,] \\ & & \mathsf{else}\ append(reverse(\mathsf{tl}\, l), [\mathsf{hd}\, l]) \end{array}$$

$$\begin{array}{lcl} reversea : X^* \times X^* \to X^* & & \\ reversea(x, y) & \triangleq & \mathsf{if}\ x = [] \\ & & \mathsf{then}\ y \\ & & \mathsf{else}\ reversea(\mathsf{tl}\, x, \mathsf{cons}(\mathsf{hd}\, x, y)) \end{array}$$

In general, the functions involved in these examples can be expressed in the form:

$$\begin{array}{lcl} f(x) & \triangleq & \mathsf{if}\ b(x) \\ & & \mathsf{then}\ k \\ & & \mathsf{else}\ h(f(p(x)), t(x)) \end{array}$$

$$\begin{array}{lcl} g(x, y) & \triangleq & \mathsf{if}\ b(x) \\ & & \mathsf{then}\ y \\ & & \mathsf{else}\ g(p(x), h(t(x), y)) \end{array}$$

where f, g, p, h, t are functions with an appropriate signature, b is a boolean function, and k is a constant.

For instance, the above example is obtained as follows:

$$h(x, y) \triangleq append(x, y)$$

$$p(x) \triangleq \mathsf{tl}\, x$$

$$t(x) \triangleq [\mathsf{hd}\, x]$$

$$b(x) \triangleq x = [\,].$$

Then if we take k as $[\,]$ we obtain *reverse* immediately. We obtain *reversea* by using the following simple property of *append*:

$$append([x], y) = \mathsf{cons}(x, y).$$

As the following lemma shows, the success of these methods on this class of examples is not too surprising.

Lemma

Given the above class of functions f, and g, then a proof of:

$$f(x) = g(x, k) \tag{2.1}$$

is dependent on some requirements that hold for the kind of examples for which CKD-1, CKD-2, and Aubin-2 are most successful.

Proof

We first show how an inductive proof of the following:

$$h(f(x), y) = g(x, h(y, k)) \tag{2.2}$$

would proceed if h is associative. Then we give the requirements for this lemma to specialise to equation 2.1

Unfolding g in equation 2.2 and assuming $\neg b(x)$ gives:

$$g(p(x), h(t(x), h(y, k)))$$

Since h is associative we can rewrite this to:

$$g(p(x), h(h(t(x), y), k))$$

Suppose that we are using well-founded induction and have a well-founded ordering $\prec$ where $p(x) \prec x$, then we can use the induction hypothesis to obtain:

$$h(f(p(x)), h(t(x), y)).$$

Further, using the associativity of h again, this rewrites to:

$$h(h(f(p(x)), t(x)), y)$$

and folding f gives:

$$h(f(x), y).$$

Hence we are able to carry out the induction step for equation 2.2 if h is associative.

To show the base case, we also require h to have the property that:

$$h(k, y) = h(y, k) \; for \; all \; y.$$

We say that h is required to be commutative with respect to k.

For equation 2.2 to specialise to 2.1 we require:

$$h(f(x), y) = g(x, h(y, k))$$

to specialise to

$$f(x) = g(x, k)$$

which is possible only if there is a y such that:

$$h(f(x), y) = f(x), \text{ and } h(y, k) = k.$$

Note that the condition $h(y, k) = k$ is implied by the conditions $h(f(x), y) = f(x)$, and the commutative requirement $h(k, y) = h(y, k)$.
To summarise, $f(x) = g(x, k)$ if:

1. h is associative;
2. $h(k, y) = h(y, k)$ for all y;
3. there is a y such that $h(f(x), y) = f(x)$;
4. $p(x) \prec x$ for all x such that $\neg\, b(x)$ where $\prec$ is a well-founded ordering.

It is also possible to develop a similar scheme to cater for the fibo example which involves nested recursion. That is, given the form:

$$\begin{array}{lll} f(x) & \triangleq & \textsf{if } b(x) \\ & & \textsf{then } k \\ & & \textsf{else } h(f(p1(x)), h(f(p2(x)), t(x))) \end{array}$$

$$\begin{array}{lll} g(x, y) & \triangleq & \textsf{if } b(x) \\ & & \textsf{then } h(y, k) \\ & & \textsf{else } g(p1(x), g(p2(x), h(t(x), y))) \end{array}$$

Example	CKD-1	CKD-2	Aubin-1	Aubin-2	BM
append-same	$\times$	$\times$	$\surd$	$\times$	$\times$
append-rev	$\times$	$\times$	$\surd$	$\times$	$\surd$
length	$\times$	$\times$	$\times$	$\times$	∂
gcd	$\times$	$\times$	$\times$	$\times$	$\times$
nines	$\times$	$\times$	$\times$	$\times$	$\times$
nsquareprop	$\times$	$\times$	$\times$	∂	$\times$
elemsofrev	$\times$	$\times$	$\times$	$\times$	$\times$
rangebt	$\times$	$\times$	$\times$	$\times$	$\times$

Table 2.2: Remaining examples

one can prove $h(f(x), m) = g(x, m)$ provided that the above conditions hold.

Hence for such cases, in which CKD-1, CKD-2, and Aubin-2 are particularly successful, it seems more appropriate to use this result instead of the heuristics. That is, one only needs to show that the requirements hold in order to conclude that equation 2.1 holds.

Omitting this class of examples from further consideration leaves us to analyse the examples in table 2.2.

¿From table 2.2 we note that CKD-1 and CKD-2 do not work for any of the remaining examples. Further the authors of CKD-1 and CKD-2 do not give examples to demonstrate that their methods work for examples that are not in this class, and we have been unable to find such examples. We therefore remove them from further consideration. The other methods are able to cover four of the eight remaining examples. We pursue the examples for which none of the methods work as follows:

- A closer look at the elemsofrev, and the rangebt examples shows that they are both properties of accumulator functions. Accumulator functions are an important class of functions since they are often more efficient than their divide and conquer equivalents. For instance they can result in tail recursive functions which can be optimised by a compiler (e.g. tail recursion optimisation in Prolog [Dod90]).

  Hence in the next section we develop a method capable of handling such examples. The approach that we develop also works on the nsquareprop example. Hence, the inclusion of the new approach will allow us to cover six of the eight remaining examples without the need for Aubin-2

- None of the approaches, including the one we develop, is capable of generalising the nines and the gcd examples. The manner in which they are generalised appears to require too much ingenuity and is too specific to suggest a useful method of generalisation (the reader can refer to [Vad92] for a closer examination of these problems).

2.2 The New Approach

In our evaluation of existing methods in the previous section, we encountered four examples for which none of the existing methods work. Two of these examples can be characterised as properties of accumulator functions. In this chapter we develop a new method that is more appropriate for the generalisation of properties of accumulator functions. That is, given an accumulator function:

$$g(x, y) \quad \triangleq \quad \textsf{if } b(x) \textsf{ then } y \textsf{ else } g(p(x), h(t(x), y))$$

we aim to generalise a property

$$P(g(x, k_0), x, k_0)$$

to

$$Q(g(x, y), x, y)$$

where k_0 is a constant, and so that if P is valid then Q is valid. Further, Q should be easier to prove by induction than P, and Q can be specialised to P.

When the accumulator function can be translated to a divide and conquer functions by the transformation given on page 18, we could first prove the property in terms of the divide and conquer recursive function, and then use its equivalence to the accumulator function in order to show that the property holds for the accumulator function.

However, there are situations where this approach appears to be unnatural. For example, consider the proof of:

$$g(n, 0) = n^2 + n$$

where

$$
\begin{array}{lll}
g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} & & \\
g(i, j) & \triangleq & \textsf{if } i = 0 \\
 & & \textsf{then } j \\
 & & \textsf{else } g(i-1, 2 * i + j)
\end{array}
$$

To prove this lemma by first finding a divide and conquer function that is equivalent to g, and then proving the property in terms of the new function is a little convoluted. It is much more direct to prove a generalisation:

$$g(n, a) = n^2 + n + a$$

We therefore develop an approach to the generalisation of properties of accumulator functions that does not rely on first finding an equivalent divide and conquer function.

Given a property P of an accumulator function g, the basic approach is as follows:

1. Obtain a relationship between $g(x, a)$ and $g(x, k_0)$, which accounts for the effect of introducing an arbitrary accumulator a in place of the constant k_0.[1]

2. Use the relationship to generalise the original lemma. Unlike, the approaches we described earlier, we give a class of lemmas for which the approach guarantees to maintain validity subject to some requirements.

2.3 Obtaining a Relationship

Given an arbitrary accumulator function, the process of obtaining a relationship that accounts for the introduction of an arbitrary initial value is a difficult task. Hence, we restrict ourselves to some special cases. First we consider a case when the accumulator function g takes the following form in which h is associative:

[1] For simplicity, we will describe the process in terms of a function that has two parameters, the second of which is the accumulator.

$$
\begin{aligned}
g(x, y) \quad \triangleq \quad & \textsf{if } b(x) \\
& \textsf{then } y \\
& \textsf{else } g(p(x), h(t(x), y)).
\end{aligned}
$$

In general,

$$g(x, a) = h(t(ppp \ldots px), \ldots h(t(ppx), h(t(px), h(t(x), a))))$$

and

$$g(x, k_0) = h(t(ppp \ldots px), \ldots h(t(ppx), h(t(px), h(t(x), k_0))))$$

where $p(x)$ is abbreviated to px, and $b(pppp..px)$ is true.

Now, in the case that h is associative, the difference is between:

$$g(x, a) = h(\ldots h(h(h(t(ppp \ldots px), \ldots t(ppx)), t(px)), t(x))), a))))$$

and

$$g(x, k_0) = h(\ldots h(h(h(t(ppp \ldots px), \ldots t(ppx)), t(px)), t(x))), k_0))))$$

Now provided $h(y, k_0) = y$, we can express the difference as:

$$h(g(x, k_0), a) = g(x, a) \tag{2.3}$$

Notice that although $h(x, y)$ is required to be associative, there is no need for $h(t(x), y)$ to be associative. Thus, for our earlier example:

$$
\begin{aligned}
& g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\
& g(i, j) \quad \triangleq \quad \textsf{if } i = 0 \\
& \qquad\qquad\quad\; \textsf{then } j \\
& \qquad\qquad\quad\; \textsf{else } g(i - 1, 2 * i + j)
\end{aligned}
$$

if we define:

$$h(t(x), y) = 2 * x + y$$

where

$$t(x) = 2 * x$$

then $h(t(x), y)$ is not associative. But $h(x, y)$ is associative and one can therefore express the difference as in equation 2.3.

This class of accumulator functions represent most of the accumulator functions that we have encountered in the literature. However, there are functions that are not covered by this class. For example, suppose we want to develop a function to compute the following product for $n \in \mathbb{N}_1$:

$$\prod_{i=1}^{n} i^{2^{(i-2)}}$$

Suppose we dream up[2] the following function as an implementation:

$$f : \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$$
$$f(n, a) \quad \triangleq \quad \textsf{if } n = 1 \textsf{ then } a \textsf{ else } f(n-1, n * a^2)$$

We would, of course, like to prove that it does indeed compute what we claim:

$$f(n, 1) = \prod_{i=1}^{n} i^{2^{(i-2)}}$$

Unfortunately, the above scheme for obtaining a difference is not applicable since taking:

$$h(x, y) \quad \triangleq \quad x * y^2$$

we note that h is not associative. It is, however, possible to carry out the proof from the following generalisation:

$$f(n, a) = a^{2^{(n-1)}} * \prod_{i=1}^{n} i^{2^{(i-2)}}$$

If a wider class of functions is worth covering, then one approach would be to encorporate some symbolic execution facility together with an equation solver (like PRESS [Sil85]) which could determine the difference.

2.4 Obtaining a Balanced Generalisation

In this section we consider how we can introduce the relationship obtained in the previous section into the original lemma without producing an invalid property. We say that this step obtains a *balanced generalisation.*

Thus, if the relationship between the value of an accumulator function at an arbitrary variable, and at its initial value can be expressed by:

$$g(x, a) = h(g(x, k_0), a) \tag{2.4}$$

[2] I leave it to the program transformation community to attempt to derive it.

where k_0 is the initial value of the accumulator, then subject to some further requirements, we show how our approach proceeds to obtain a balanced generalisation of an initial lemma of the form:

$$\mu(g(x, k_0)) \,\mathsf{OP}\, f(x) \tag{2.5}$$

where OP is a relational operator, and μ, OP, and f are compatible functions of appropriate type.

Then we can use equation 2.4 and obtain:

$$\mu(g(x, a)) = \mu(h(g(x, k_0), a)) \tag{2.6}$$

Now if we have the following property for μ:

$$\mu(h(x, y)) = h'(\mu(x), \mu(y)) \tag{2.7}$$

where μ is *homomorphic* with respect to h, and h' corresponds to h, then using this property on equation 2.6 gives:

$$\mu(g(x, a)) = h'(\mu(g(x, k_0)), \mu(a)) \tag{2.8}$$

Further suppose we have a property:

$$p \,\mathsf{OP}\, q \;\Rightarrow\; h'(p, r) \,\mathsf{OP}\, h'(q, r) \tag{2.9}$$

where OP is *monotonic* with respect to h', then using this on our original lemma (2.5) we can obtain:

$$h'(\mu(g(x, k_0)), \mu(a)) \,\mathsf{OP}\, h'(f(x), \mu(a))$$

Now using equation 2.8 to replace the left hand side gives the generalised version:

$$\mu(g(x, a)) \,\mathsf{OP}\, h'(f(x), \mu(a))$$

Thus, if the requirements expressed by 2.4, 2.7, and 2.9 are met, then we have obtained a generalisation that has preserved validity.

We also require that this should specialise to the original lemma. This leads to the additional requirement:

$$h'(f(x), \mu(k_0)) = f(x)$$

As we will see in the following example, it is not always the only form to preserve validity.

Example: nsquareprop

Consider the function we presented earlier:

$$\begin{array}{lll} g : \mathbb{N} \times \mathbb{N} \to \mathbb{N} & & \\ g(i,j) & \triangleq & \mathsf{if}\ i = 0 \\ & & \mathsf{then}\ j \\ & & \mathsf{else}\ g(i-1, 2*i+j) \end{array}$$

and suppose we wish to prove the following property:

$$g(n, 0) > n^2 + n - 1.$$

We can express the difference by:

$$g(x, a) = g(x, 0) + a$$

The homomorphic requirement holds trivially since μ is the identity function. The monotonic requirement also holds:

$$x = y \;\Rightarrow\; x + z = y + z$$

Hence the generalisation is:

$$g(n, a) > n^2 + n - 1 + a \tag{2.10}$$

Notice, however that this is not the only valid form. The following two forms are also valid for this example:

$$g(n, a) > n^2 + n - 1 \tag{2.11}$$

$$g(n, a) + a > n^2 + n - 1 \tag{2.12}$$

Lemma 2.11 certainly looks neater than the proposed form, and is easier to specialise to the required lemma. However, we should use our overall criteria for judging which one is better. That is, which one is easier to prove by induction? We present the attempts to prove the induction step below.

Induction step for lemma 2.10

$$\begin{aligned} g(n+1,a) &= g(n, 2*(n+1)+a) \\ &> n^2+n-1+2*(n+1)+a \\ &> (n+1)^2+(n+1)-1+a \end{aligned}$$

Induction step for lemma 2.11

$$\begin{aligned} g(n+1,a) &= g(n, 2*(n+1)+a) \\ &> n^2+n-1 \end{aligned}$$

Now since the right hand side is smaller than the right hand side of the goal, we are unable to use the transitivity of $>$ to complete the induction step.

Induction step for lemma 2.12

$$\begin{aligned} g(n+1,a)+a &= g(n, 2*(n+1)+a)+a \\ &> n^2+n-1-(2*(n+1)+a)+a \\ &> n^2-n-3 \end{aligned}$$

Once again, since the right hand side is smaller than the right hand side of the goal, we are unable to complete the induction step.

In chapter 2 we gave an example that showed that generalisation can lead to an invalid statement. In contrast, this example shows that it is possible to generalise a constant so that the resulting generalisation remains valid but still remains difficult to prove. Thus although our approach guarantees a valid generalisation we do not guarantee that it will always lead to an appropriate generalisation. It does however obtain the appropriate generalisation in the above example, and we have not found an example where it does not produce an appropriate generalisation.

Example: elemsofrev

We now consider an example where the homomorphism requirement plays a role. Consider the following accumulator function for reversing a sequence:

$$
\begin{array}{lll}
greverse : X^* \times X^* \to X^* & & \\
greverse(l, a) & \triangleq & \mathsf{if}\ l = [\,] \\
& & \mathsf{then}\ a \\
& & \mathsf{else}\ greverse(\mathsf{tl}\, l, [\mathsf{hd}\, l] \curvearrowright a)
\end{array}
$$

Suppose we wish to prove the property:

$$\mathsf{elems}\,(greverse(l, [\,])) = \mathsf{elems}\,(l)$$

The difference can be expressed as:

$$greverse(x, a) = greverse(x, [\,]) \curvearrowright a$$

The homomorphic requirement holds:

$$\mathsf{elems}\,(x \curvearrowright y) = \mathsf{elems}\,(x) \cup \mathsf{elems}\,(y)$$

The monotonic requirement holds:

$$x = y \;\Rightarrow\; x \cup z = y \cup z$$

Hence we have a generalisation:

$$\mathsf{elems}\,(greverse(l, a)) = \mathsf{elems}\,(l) \cup \mathsf{elems}\,(a)$$

Example: rangebt

As a third example, we show how the need for generalisation could arise when we develop a function to satisfy a specification.

Consider how one might write a procedure to find a list of all the numbers in an ordered binary tree which are in a given range. Given the following definition of ordered binary trees in VDM:[3]

[3] Readers unfamiliar with this definition should consult [Jon90].

$$Bt = [Node]$$

$$\begin{aligned} Node :: \; lt &: Bt \\ v &: \mathbb{N} \\ rt &: Bt \end{aligned}$$

where

$$inv\text{-}Node(mk\text{-}Node(lt, v, rt)) \quad \triangleq$$
$$\quad (\forall i \in retrns(lt) \cdot i < v) \wedge (\forall j \in retrns(rt) \cdot j > v)$$

$$retrns : Bt \to \mathbb{N}\text{-}\mathsf{set}$$
$$retrns(bt) \quad \triangleq$$
$$\begin{aligned} &\mathsf{cases}\ bt\ \mathsf{of} \\ &\qquad\qquad\qquad \mathsf{nil} \to \{\,\} \\ &mk\text{-}Node(lt, v, rt) \to retrns(lt) \cup retrns(rt) \cup \{v\} \\ &\mathsf{end} \end{aligned}$$

We wish to develop a function to satisfy:

$$range\ (bt \colon Bt, x, y \colon \mathbb{N})\ r \colon \mathbb{N}^*$$
$$\mathsf{pre\ true}$$
$$\mathsf{post\ elems}\ r = \{i \mid i \in retrns(bt) \wedge i \geq x \wedge i \leq y\} \wedge no\_duplicates(r)$$

where

$$no\_duplicates : \mathbb{N}^* \to \mathbb{B}$$
$$no\_duplicates(s) \quad \triangleq \quad \mathsf{len}\ s = \mathsf{card\ elems}\ s$$

There are, of course, several functions that could be written to satisfy this specification. For example, a straightforward divide and conquer version could take the form:

$$range : Bt \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}^*$$
$$range(bt, x, y) \quad \triangleq$$
$$\begin{aligned} &\mathsf{cases}\ bt\ \mathsf{of} \\ &\qquad\qquad\qquad \mathsf{nil} \to [\,] \\ &mk\text{-}Node(lt, v, rt) \to \mathsf{if}\ (v \geq x)\ \mathsf{and}\ (v \leq y) \\ &\qquad\qquad\qquad\qquad \mathsf{then}\ range(lt, x, y) \frown [v] \frown range(rt, x, y) \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ range(lt, x, y) \frown range(rt, x, y) \\ &\mathsf{end} \end{aligned}$$

However, given that the tree is ordered, this is not a particularly efficient function. An improvement on this would be to use the fact that the tree is ordered. An accumulator function that makes use of the ordering takes the form:[4]

$$
\begin{array}{l}
grange : Bt \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}^* \to \mathbb{N}^* \\
grange(bt, x, y, a) \quad \triangleq \\
\quad \textsf{cases } bt \textsf{ of} \\
\qquad \qquad \qquad \textsf{nil} \to a \\
\qquad mk\text{-}Node(lt, v, rt) \to \textsf{if } (y < v) \\
\qquad \qquad \qquad \qquad \textsf{then } grange(lt, x, y, a) \\
\qquad \qquad \qquad \qquad \textsf{else if } (x > v) \\
\qquad \qquad \qquad \qquad \quad \textsf{then } grange(rt, x, y, a) \\
\qquad \qquad \qquad \qquad \quad \textsf{else } grange(lt, x, y, [v] \frown grange(rt, x, y, a)) \\
\quad \textsf{end}
\end{array}
$$

Then we can define *range* by:

$$
\begin{array}{l}
range : Bt \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}^* \\
range(bt, x, y) \quad \triangleq \quad grange(bt, x, y, [\,])
\end{array}
$$

The proof obligation that this definition satisfies the specification takes the form:

$$
\begin{array}{l}
\forall bt \in Bt, x, y \in \mathbb{N} \cdot pre\text{-}range(bt, x, y) \\
\qquad \Rightarrow \quad range(bt, x, y) \in \mathbb{N}^* \wedge post\text{-}range(bt, x, y, range(bt, x, y))
\end{array}
$$

[4] Note that a divide and conquer version would make heavier use of concatenation.

Simplifying by using the definitions and the fact that we can introduce $\forall$, we need to prove:

$$\begin{array}{l} bt \in Bt, x, y \in \mathbb{N} \\ \quad \vdash grange(bt, x, y, [\,]) \in \mathbb{N}^* \wedge \\ \qquad no\_duplicates(grange(bt, x, y, [\,])) \wedge \\ \qquad \mathsf{elems}\, grange(bt, x, y, [\,]) = \{i \mid i \in retrns(bt) \wedge i \geq x \wedge i \leq y\} \end{array}$$

One strategy for proceeding to prove such a conjunction is to prove the conjuncts independently and then to use the $\wedge$-I rule. For illustrative purposes, we concentrate on the last conjunct:

$$\mathsf{elems}\, grange(bt, x, y, [\,]) = \{i \mid i \in retrns(bt) \wedge i \geq x \wedge i \leq y\}$$

A proof of this requirement is not however straightforward. Suppose we use the induction rule given in Jones [Jon90, p. 123]:

$$\boxed{\text{Bt-ind}}\ \frac{\begin{array}{c} p(\mathsf{nil});\ v \in \mathbb{N}, lt, rt \in Bt, \\ inv\text{-}Node(mk\text{-}Node(lt, v, rt)), p(lt), p(rt) \vdash p(mk\text{-}Node(lt, v, rt)) \end{array}}{bt \in Bt \vdash p(bt)}$$

Consider what happens in the induction step. Under the assumption of the induction hypothesis, we need to show:

$$\begin{array}{l} \mathsf{elems}\, grange(mk\text{-}Node(lt, v, rt), x, y, []) = \\ \qquad\qquad \{i \mid i \in retrns(mk\text{-}Node(lt, v, rt)) \wedge i \geq x \wedge i \leq y\}. \end{array}$$

In the case that: $x \geq v \leq y$ if we unfold $grange$ on the left we obtain a left hand side:

$$\mathsf{elems}\,(grange(lt, x, y, [v] \curvearrowright grange(rt, x, y, [\,]).)$$

Hence, once again, we cannot use the induction hypothesis because the accumulator in the outer application of $grange$ is not the empty sequence.

We now attempt to generalise this lemma. The first point to note is that the function $grange$ does not fit the format of the function given in section 2.3. Hence, we cannot use the theory presented there to obtain a relationship that accounts for the effect of introducing an arbitrary accumulator. However, we might guess the following relationship:[5]

$$grange(bt, x, y, a) = grange(bt, x, y, [\,]) \curvearrowright a.$$

[5] In practice a proof of this is not worthwhile.

The approach given in section 2.4 can now be applied. The homomorphic requirement holds with $\cup$ corresponding to $\curvearrowright$:

$$\mathsf{elems}\,(x \curvearrowright y) = \mathsf{elems}\,(x) \cup \mathsf{elems}\,(y).$$

The monotonic requirement also holds:

$$x = y \;\Rightarrow\; (x \cup r) = (y \cup r).$$

Hence the generalisation is:

$$\mathsf{elems}\,(grange(bt, x, y, a)) = \{i \mid i \in retrns(bt) \wedge i \geq x \wedge i \leq y\} \cup \mathsf{elems}\, a.$$

An inductive proof of this now proceeds without obstruction.

2.5 Properties of Twisted Accumulator Functions

We define a twisted accumulator function to be one that swaps its accumulators in the recursive call. For instance, the following example is a twisted accumulator function which splits a list into two roughly even parts:[6]

$$f : X^* \times X^* \times X^* \to Pair(X^*, X^*)$$
$$f(l, ll, lr) \quad \triangleq \quad \mathsf{if}\ l = [\,]\ \mathsf{then}\ (ll, lr)\ \mathsf{else}\ f(\mathsf{tl}\, l, lr, \mathsf{hd}\, l :: ll)$$

where we use the infix notation $a :: s$ instead of $\mathsf{cons}(a, s)$ to improve the readability of the following description.

In practice, this class of functions is probably a little esoteric. Our purpose in studying them is to illustrate that they do stretch the approach that we used in the previous section. They also help us to illustrate the point that there are generalisation problems for which we must utilise more information about the control (i.e., the recursion) variable.

Properties of twisted accumulator functions cause similar problems to properties of accumulator functions. That is, generalisation of initial accumulator values to variables may be necessary in order to enable the use of an induction hypothesis. In addition, the generalisation becomes more

[6] We also leave this function for the program transformation community to derive.

complex due to the argument swapping. For the above example, the values of the accumulators on each recursive call take the form:

$$\begin{array}{llllll} ll & lr & l_1::ll & l_2::lr & & \ldots \\ lr & l_1::ll & l_2::lr & l_3::l_1::ll & & \ldots \end{array}$$

where l_i is the ith element in the sequence l.

Hence, when l has an odd number of elements, say $2i+1$, then the difference between $f(l, ll, lr)$ and $f(l, [\,], [\,])$ is between the pairs:

$$(l_{2i}::l_{2(i\text{-}1)}::\ldots\ lr,\ l_{2i+1}::l_{2i\text{-}1}::\ldots\ ll)$$

and

$$(l_{2i}::l_{2(i\text{-}1)}::\ldots\ [\,],\ l_{2i+1}, l_{2i\text{-}1}::\ldots\ [\,]).$$

That is, when the number of elements in l is odd we can express the difference by:

$$f(l, ll, lr) = (\mathsf{fst} f(l, [\,], [\,]) \curvearrowright lr, \mathsf{snd} f(l, [\,], [\,]) \curvearrowright ll).$$

When the number of elements in l is even the difference can be expressed by:

$$f(l, ll, lr) = (\mathsf{fst} f(l, [\,], [\,]) \curvearrowright ll, \mathsf{snd} f(l, [\,], [\,]) \curvearrowright lr)$$

Thus, unlike the cases considered earlier, the difference is dependent on the recursion (control) variable. Hence, one cannot, in general, produce a generalisation independently of the control variable. For example, given the following property:

$$(\mathsf{len}\ l)\ \mathsf{div}\ 2 \leq \mathsf{len}\ \mathsf{fst}\ f(l, [\,], [\,]).$$

We note that if we attempt to generalise this lemma with the approach of the previous section, then if we use the even difference we obtain the invalid lemma:

$$(\mathsf{len}\ l)\ \mathsf{div}\ 2 + \mathsf{len}\ ll \leq \mathsf{len}\ \mathsf{fst}\ f(l, ll, lr).$$

For example, it does not hold for:

$$f([1], [2, 3, 4, 5, 6], [7]) = ([7], [1, 2, 3, 4, 5, 6]).$$

The odd difference leads to a similar invalid form. The other methods considered in chapter 1 do not work for this example either. The example can, however be generalised to:

$$odd(\mathsf{len}\ l) \Rightarrow (\mathsf{len}\ l)\ \mathsf{div}\ 2 + \mathsf{len}\ lr \leq \mathsf{len}\ \mathsf{fst}\ f(l, ll, lr)$$

$$\wedge$$

$$even(\mathsf{len}\ l) \Rightarrow (\mathsf{len}\ l)\ \mathsf{div}\ 2 + \mathsf{len}\ ll \leq \mathsf{len}\ \mathsf{fst}\ f(l, ll, lr).$$

However, there are properties of twisted accumulator functions for which the dependency on the control variable does not matter. For example generalising:

$$\mathsf{len}\ \mathsf{fst}(f(l, [\,], [\,])) + \mathsf{len}\ \mathsf{snd}(f(l, [\,], [\,])) = \mathsf{len}\ l$$

gives:

$$\mathsf{len}\ \mathsf{fst}(f(l, ll, lr)) + \mathsf{len}\ \mathsf{snd}(f(l, ll, lr)) = \mathsf{len}\ l + \mathsf{len}\ ll + \mathsf{len}\ lr.$$

The approach that we have presented in this chapter can be extended to handle such examples. Although further work would be needed to cater for examples which are dependent on the control variable. However, as we mentioned earlier, such functions are a little esoteric, and our purpose has been only to emphasise that generalisation is dependent on the kind of functions involved in the lemma.

3

Bibliographic Remarks and Conclusion

Bibliographic Remarks

Our study has included an examination of the heuristics of Castaing, Kodratoff, and Degano. Castaing [Cas85] describes the use of a generalisation heuristic in their theorem proving system based on rewriting [KC83]. The heuristic she uses is based on the method CKD-1 but also utilizes Aubin's scheme of identifying primary variables. She also gives steps that correspond to Aubin-2, and Boyer and Moore's cross fertilisation heuristic.

The process of unfolding definitions in a lemma and then using the induction hypothesis is the basis of a number of program transformation and synthesis systems [MW83, Dar81, Tra86]. It is therefore not surprising that these systems also need to use some degree of generalisation. For example, systems based on Burstall and Darlington's [BD77] fold/unfold program transformation method (e.g. [Dar81]) often require inventive definitions (called eureka steps) before being able to transform programs to their efficient counter part. In the case of producing accumulator versions of programs from divide and conquer versions, these turn out to be the generalisations that we describe.

The transformation of a divide and conquer recursive function into a repetitive form under the associative requirement can also be found in [Pau87, Pep87].

Generalisation is also useful in resolution based theorem provers. The

Karlsruhe Induction Theorem Proving System based on resolution and paramodulation also uses existing generalisation heuristics [BHHW86, Hum87].

Bundy's [Bun87] research on the use of explicit proof plans to guide inductive proofs also recognises the need for generalisation. Bundy uses Aubin-1 for an example inorder to enable the use of a particular tactic (called ripple-out).

We have implemented the method for properties of accumulator functions, Aubin-1, and BM in *mural* . A description of our implementaion and the lessons learned about tactic langauges can be found in [Vad92].

Conclusion

We have described existing methods of generalisation. These methods have been tried on several examples. None of the methods worked on all the examples. Indeed, there are examples for which none of the methods work. The success of a particular method depends on the kind of functions involved in the lemma.

We have shown that the examples for which the methods CKD-1, CKD-2, and Aubin-2 are particularly successful form a well-defined class of problems. We have shown that their success on lemmas in this class is not surprising since such lemmas are true subject to some requirements.

We have developed a new method for properties of accumulator functions. The new method succeeds in generalising examples for which the other methods fail. Unlike the existing methods, we present the requirements for a class of generalisation problems for which the method guarantees to maintain validity.

A

Generalisation Problems

Example:fact (Castaing et al. [CKD83])

Show $g(x,1) = f(x)$ where

$$f : \mathbb{N} \to \mathbb{N}$$
$$f(x) \triangleq \textsf{ if } x = 0 \textsf{ then } 1 \textsf{ else } f(x-1) * x$$

$$g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$g(x,y) \triangleq \textsf{ if } x = 0 \textsf{ then } y \textsf{ else } g(x-1, x * y)$$

Example: fibo (Castaing et al. [CKD83])

Show $\mathit{fibo}(x,0) = \mathit{fib}(x)$ where

$$\mathit{fib} : \mathbb{N} \to \mathbb{N}$$
$$\mathit{fib}(x) \triangleq \textsf{ if } (x = 0) \textsf{ or } (x = 1) \textsf{ then } 1 \textsf{ else } \mathit{fib}(x-1) + \mathit{fib}(x-2)$$

$$\mathit{fibo} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$\mathit{fibo}(x,z) \triangleq \textsf{ if } (x = 0) or (x = 1) \textsf{ then } z + 1 \textsf{ else } \mathit{fibo}(x-1, \mathit{fibo}(x-2, z))$$

Example: exp3 (Manna and Waldinger [MW85])

Show $exp3(x, n, 1) = exp(x, n)$ where

$$exp : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$exp(x, n) \;\triangleq\; \textsf{if } n = 0 \textsf{ then } 1 \textsf{ else } exp(x, n-1) * x$$

$$exp3 : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$exp3(x, n, z) \;\triangleq\; \textsf{if } n = 0 \textsf{ then } z \textsf{ else } exp3(x, n-1, z * x)$$

Example: append-same (Aubin [Aub79])

Show $append(j, append(j, j)) = append(append(j, j), j)$ where

$$append : X^* \times X^* \rightarrow X^*$$
$$append(a, b) \;\triangleq\; \textsf{if } a = [\,] \textsf{ then } b \textsf{ else } (\textsf{cons}(\textsf{hd } a, append(\textsf{tl } a, b)))$$

Example: reverse (Aubin [Aub79])

Show $reverse(l) = reversea(l, [\,])$ where

$$reverse : X^* \rightarrow X^*$$
$$reverse(l) \;\triangleq\; \textsf{if } l = [\,] \textsf{ then } [\,] \textsf{ else } append(reverse(\textsf{tl } l), [\textsf{hd } l])$$

$$reversea : X^* \times X^* \rightarrow X^*$$
$$reversea(x, y) \;\triangleq\; \textsf{if } x = [\,] \textsf{ then } y \textsf{ else } reversea(\textsf{tl } x, \textsf{cons}(\textsf{hd } x, y))$$

Example: append-rev (Aubin [Aub79])

Show $append(reverse(a), append(b, c)) = append(append(reverse(a), b), c)$

Example: length (Bundy [Bun79])

Show $length(length(x)) = length(x)$ where

$$length : X^* \rightarrow \textsf{nil}^*$$
$$length(x) \;\triangleq\; \textsf{if } x = [\,] \textsf{ then } [\,] \textsf{ else } \textsf{cons}(\textsf{nil}, length(\textsf{tl } x))$$

Example: gcd (Manna and Waldinger [MW85])

Show $gcd(x, y)$ *divides* x where

$$gcd : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$gcd(x, y) \;\triangleq\; \textsf{if } y = 0 \textsf{ then } x \textsf{ else } gcd(y, rem(x, y))$$

$$\_\ divides\ \_ : \mathbb{N} \times \mathbb{N} \to \mathbb{B}$$
$$x\ divides\ y \triangleq y\ rem\ x = 0$$

Example: nines (Manna and Waldinger [MW85])

Show

$$\forall ns \in Digit^* \cdot 9\ divides\ v(ns) \;\Rightarrow\; 9\ divides\ sum(ns)$$

where

$$Digit = \{0, 1, .., 9\}$$

$$\begin{array}{l} v : Digit^* \to \mathbb{N} \\ v(ns) \quad \triangleq \quad \mathsf{if}\ ns = [\,] \\ \qquad\qquad\quad \mathsf{then}\ 0 \\ \qquad\qquad\quad \mathsf{else\ let}\ k = \mathsf{len}\ ns - 1\ \ \mathsf{in} \\ \qquad\qquad\qquad\quad (\mathsf{hd}\ \ ns) * 10^k + v(\mathsf{tl}\ \ ns) \end{array}$$

$$\begin{array}{l} sum : Digit^* \to \mathbb{N} \\ sum(ns) \quad \triangleq \quad \mathsf{if}\ ns = [\,]\ \ \mathsf{then}\ 0\ \ \mathsf{else\ hd}\ ns + sum(\mathsf{tl}\ ns) \end{array}$$

Example: nsquareprop

Show $g(n, 0) = n^2 + n$ where

$$\begin{array}{l} g : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ g(i, j) \quad \triangleq \quad \mathsf{if}\ i = 0\ \ \mathsf{then}\ j\ \ \mathsf{else}\ g(i - 1, 2 * i + j) \end{array}$$

Example: elemsofrev

Show $\mathsf{elems}\,(greverse(l,[\,]))=\mathsf{elems}\,(l)$ where:

$$greverse : X^* \times X^* \to X^*$$
$$greverse(l,a) \quad \triangleq \quad \mathsf{if}\ l=[\,]\ \mathsf{then}\ a\ \mathsf{else}\ greverse(\mathsf{tl}\,l,[\mathsf{hd}\,l] \frown a)$$

Example: rangebt

Show:

$$grange(bt,x,y,[\,])=\{i \mid i \in retrns(bt) \wedge i \geq x \wedge i \leq y\}$$

where

$$grange : Bt \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}^* \to \mathbb{N}^*$$
$$\begin{array}{l}
grange(bt,x,y,a) \quad \triangleq \\
\quad \mathsf{cases}\ bt\ \mathsf{of} \\
\qquad \qquad \qquad \mathsf{nil} \to a \\
\quad mk\text{-}Node(lt,v,rt) \to \mathsf{if}\ (y<v) \\
\qquad \qquad \qquad \qquad \mathsf{then}\ grange(lt,x,y,a) \\
\qquad \qquad \qquad \qquad \mathsf{else\ if}\ (x>v) \\
\qquad \qquad \qquad \qquad \quad \mathsf{then}\ grange(rt,x,y,a) \\
\qquad \qquad \qquad \qquad \quad \mathsf{else}\ grange(lt,x,y,[v] \frown grange(rt,x,y,a)) \\
\quad \mathsf{end}
\end{array}$$

Bibliography

[Aub79] R. Aubin. Mechanising structural induction II: Strategies. *Theoretical Computer Science*, 9:347–361, 1979.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of ACM*, 24:44–67, 1977.

[BHHW86] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe Induction theorem proving system. In *8th Int. Conf on Automated Deduction, LNCS230*, pages 672–673, Institut fur Informatik I, Universitat Karlsruhe,D-7500, Karlsruhe, 1986.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

[Bun79] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1979.

[Bun87] Alan Bundy. The use of explicit plans to guide inductive proofs. Technical Report DAI 349, Department of Artificial Intelligence, University of Edinburgh, 1987.

[Cas85] J. Castaing. How to facilitate the proof of theorems by using induction-matching and by generalisation. In *Proceedings of the ninth International Joint Conference on Artificial Intelligence*, pages 1208–1213, 1985.

[CKD83] J. Castaing, Y. Kodratoff, and P. Degano. Theorem proving by the study of example proof traces. In Alan W. Bierman, Gerard Guiho, and Yves Kodratoff, editors, *Automatic Program Construction Techniques*, pages 421–433. Macmillan, 1983.

[Dar81] John Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46, 1981.

[Dod90] Tony Dodd. *An Advanced Logic Programming Language: Prolog-2 User Guide*. Intellect, 1990.

[Hum87] Brigit Hummel. An investigation of formula generalisation for induction proofs. Technical report, Institut fur Logik, Komplexitat und Deduktionssystem, Universitat Karlsruhe, Postfache 6980, Karlsruhe 1, Germany, 1987.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[KC83] Y. Kodratoff and J. Castaing. Trivializing the proof of trivial theorems. In *Proceedings of the eight International Joint Conference on Artificial Intelligence*, pages 930–932, Karlsruhe, Germany, August 1983.

[M+73] Z. Manna et al. Inductive methods for proving properties of programs. *CACM*, 16(8), 1973.

[MW83] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. In *Automatic Program Construction Techniques*, pages 33–68. Macmillan, 1983.

[MW85] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming I*. Addison Wesley, Reading, Massachusetts, 1985.

[Pau87] Marvin C. Paul. *A Recursion Transformation framework*. Wiley, 1987.

[Pep87] Peter Pepper. A simple calculus for program transformation. *Science of Programming*, 9(3):221–262, 1987.

[Pol57] George Polya. *How to solve it*. Princeton University Press, 1957. 2nd edition.

[Sil85] B. Silver. *Meta-Level Inference*. North Holland, 1985.

[Tra86] Jonathan Traugott. Deductive synthesis of sorting programs. In *8th Int. Conf on Automated Deduction, LNCS 230*, pages 641–660, 1986.

[Vad92] Sunil Vadera. *Heuristics for Proofs*. PhD thesis, University of Manchester, Manchester M13 9PL, UK, 1992.