Review of heuristics for generalisation

Vadera, S

Author version of paper:
Sunil Vadera, Review of heuristics for generalisation,
published in the IEE Software Engineering Journal, July 1995, pp 118-123

Review of heuristics for generalisation

Sunil Vadera
S.Vadera@salford.ac.uk
University of Salford

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. Existing heuristics for generalisation for induction are examined. The applicability of heuristics for generalisation is also examined, and it is shown that the kind of examples on which some of the heuristics work best form a well defined class of problems. A class of generalisation problems is identified for which none of the methods work, and directions for future research are provided.

1 Introduction to generalisation

1.1 Motivation

Formal methods have been advocated as a way of increas-ing the reliability of the developed software. The use of formal methods first involves developing a specification. Development steps or refinements are then made towards an implementation. A key advantage of using a formal method is that we can prove that the development steps are correct with respect to their specification. The initial specification and the early development steps often involve recursive functions (e.g. in specifications [1]). Hence, in using formal methods, we may need to prove that recur-sive functions satisfy their specification. A common way of carrying out such proofs is to use induction. Proof by induction proceeds in two stages. First, the base

case is proved, and then an induction step is proved. Normally, the base case is 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 does 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.

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

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

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 thesis 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 ([1]):

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

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 $n \in \mathbf{N}$. 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 $n \in \mathbf{N}$, and any $y \in \mathbf{N}$. 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 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 [2]. 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).

As a more general lemma applies in more circum-stances, then an induction hypothesis should also be applicable in more cases when an inductive proof is attempted. Furthermore, as the use of the induction hypothesis is an important step in an inductive proof, we can expect an inductive proof

of a more general lemma to be easier than that of a more specialised lemma. Several authors have proposed heuristics for generalisation. In the following, we summarise five published heuristics for generalisation. We examine the success of these heuristics, identify a class of problems for which the methods are most successful, and a class of problems for which the existing heuristics fail. The notation we use is consistent with that found in several texts on VDM [3].

1.2 Generalisation of Minimal Common Subterms

Boyer and Moore's theorem proving system [4] 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:

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

where

$$
\begin{array}{l}
append : X^* \times X^* \to X^* \\
append(a, b) \triangleq \textbf{if } a = [\,] \\
\qquad\qquad\quad \textbf{then } b \\
\qquad\qquad\quad \textbf{else } (\textbf{cons}(\textbf{hd}\ a, append(\textbf{tl}\ a, b)))
\end{array}
$$

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 [5] 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. RRplace 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:

$$g(x\text{-}1, x * 1) = f(x\text{-}1) * x$$

3. Now generalising gives:

$$g(x, x2 * x3) = f(x) * x4 \tag{1}$$

4. Obtaining relationships:

   Taking x as 0 and evaluating the functions gives:

$$x2 * x3 = 1 * x4 \tag{2}$$

   Taking x as 1 and evaluating the functions also gives:

$$1 * x2 * x3 = 1 * 1 * x4 \tag{3}$$

At this stage, given that after simplification, one obtains the same relationship in 2 and 3, Castaing et al. [5] advise that we should stop and utilise the relationship. Thus substituting this relationship back into equation 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 [5] 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 [5] 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* [4].[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 the relationships to simplify the equation.

3. Prove the simplified equation.

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

[1] This step and term is based on a heuristic proposed by Boyer and Moore [4]. In their system, they propose that using cross-fertilisation, and then forgetting the equality in the hypothesis can result in a stronger lemma.

Example: fibo

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

$$\begin{array}{l} \mathit{fib} : \mathbf{N} \to \mathbf{N} \\ \mathit{fib}(x) \triangleq \begin{array}{l} \textbf{if } (x = 0) \textsf{ or } (x = 1) \\ \textbf{then } 1 \\ \textbf{else } \mathit{fib}(x\text{-}1) + \mathit{fib}(x\text{-}2) \end{array} \end{array}$$

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

Suppose the goal is to show that:

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

Following the steps given:

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

   $$\mathit{fibo}(x\text{-}1, \mathit{fibo}(x\text{-}2, 0)) = \mathit{fib}(x\text{-}1) + \mathit{fib}(x\text{-}2)$$

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

   $$\mathit{fibo}(x\text{-}1, \mathit{fibo}(x\text{-}2, 0)) = \mathit{fibo}(x\text{-}1, 0) + \mathit{fibo}(x\text{-}2, 0) \tag{4}$$

2. Generalising:

   $$\mathit{fibo}(x\text{-}1, \mathit{fibo}(x\text{-}2, x2)) = \mathit{fibo}(x\text{-}1, x3) + \mathit{fibo}(x\text{-}2, x4) \tag{5}$$

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

   $$\mathit{fibo}(1, x2 + 1) = x3 + 1 + x4 + 1$$

   simplifying further gives:

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

   We now substitute $x3 + x4$ for $x2$ in equation 5 to obtain:

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

Now, if we can prove this equation, we can use it to prove equation 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. For this example, it also means that we do not obtain the more natural generalisation:

$$\mathit{fibo}(x, z) = \mathit{fib}(x) + z$$

1.5 Generalisation of Primary Terms

Boyer and Moore's approach considers all the subterms of a lemma as potentially generalisable. Aubin [6], 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 [7] 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 involve performing 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 [6] 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.

2 Evaluation of existing methods

In this Section, we evaluate the existing methods described in the previous Section in order to provide a better understanding of their applicability. 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 1 summarises the results of applying the methods to various examples. These examples can be found either in the paper or in the Appendix. 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 deltas in the Aubin-2 column are present because it requires a process of massaging, and the delta in the BM column marks an occasion when an invariant property needs to be specified before generalisation. The table shows that

The table shows that:

- there is some overlap between the methods;
- 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, 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 the 'reverse' example:

For example, these three methods succeed in generalising:

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

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 1: Applicability of methods on examples

where

$$
\begin{array}{l}
\mathit{reverse} : X^* \rightarrow X^* \\
\mathit{reverse}(l) \triangleq \textbf{if } l = [\,] \\
\qquad\qquad\quad \textbf{then } [\,] \\
\qquad\qquad\quad \textbf{else } \mathit{append}(\mathit{reverse}(\textbf{tl}\ l), [\textbf{hd}\ l])
\end{array}
$$

$$
\begin{array}{l}
\mathit{reversea} : X^* \times X^* \rightarrow X^* \\
\mathit{reversea}(x, y) \triangleq \textbf{if } x = [] \\
\qquad\qquad\quad\ \ \textbf{then } y \\
\qquad\qquad\quad\ \ \textbf{else } \mathit{reversea}(\textbf{tl}\ x, \textbf{cons}(\textbf{hd}\ x, y))
\end{array}
$$

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

$$
\begin{array}{l}
f(x) \triangleq \textbf{if } b(x) \\
\qquad\quad \textbf{then } k \\
\qquad\quad \textbf{else } h(f(p(x)), t(x))
\end{array}
$$

$$
\begin{array}{l}
g(x, y) \triangleq \textbf{if } b(x) \\
\qquad\quad\ \ \textbf{then } y \\
\qquad\quad\ \ \textbf{else } g(p(x), h(t(x), y))
\end{array}
$$

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:

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

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

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

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

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

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

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 and the following:

1. h is associative;
2. there exists a k such that $h(k, y) = h(y, k)$ 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.

then we can conclude that

$$f(x) = g(x, k) \tag{6}$$

A proof of this lemma, together with a similar result covering the 'fibo' example, can be found in [8] and similar lemmas can also be found in [9, 10]. 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 heuristics,

Example	CKD-1	CKD-2	Aubin-1	Aubin-2	BM
append-same	×	×	√	×	×
append-rev	×	×	√	×	√
length	×	×	×	×	∂
gcd	×	×	×	×	×
nines	×	×	×	×	×
nsquareprop	×	×	×	∂	×
elemsofrev	×	×	×	×	×
rangebt	×	×	×	×	×

Table 2: Remaining examples

i.e. we only need to show that the requirements hold in order to conclude that equation 6 holds. Omitting this class of examples from further consideration leaves us to analyse the examples in Table 2.

From Table 2 we note that CKD-1 and CKD-2 do not work for any of the remaining examples. Furthermore, Castaing et al. [11] do not provide examples to demonstrate that their methods work for examples that are not in this class.t We therefore remove these techniques from further consideration. The other methods are able to cover four of the eight remaining examples. We can group into two categories the examples for which none of the methods works:

- A closer look at the 'elemsofrev' and 'rangebt' examples (see the Appendix) shows that they are both properties of accumulator functions. As with all the examples, their frequency of occurrence in practice is unknown. Nonetheless, as the given examples show, such problems do exist and are realistic. Furthermore, it is known that accumulator functions are an important class of functions as they are often more efficient than their divide-andconquer equivalents (e.g. tail recursion optimisation in Prolog [12]). It would therefore be a useful area of future research to attempt to develop a heuristic that can generalise such problems.

- None of the approaches is capable of generalising the 'nines' and the 'gcd' examples, given in the Appendix. The manner in which they are generalised appears to require too much ingenuity and is too specific to suggest a useful method of generalisation (readers can find a closer

examination of these problems in the technical report [8]).

3 Conclusions and further work

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 suc-cessful form a well defined class of problems. We have shown that their success on lemmas in this class is not surprising because such lemmas are true subject to some requirements. We have identified a class of problems, called properties of accumulator functions, for which the existing methods fail. This class, together with the generalisation of the 'gcd' and 'nines' problem mentioned in the Appendix, remains the subject of future work.

4 Bibliographic Remarks

Our study has included an examination of the heuristics of Castaing et al.[5]. Castaing [11] describes the use of a generalisation heuristic in their theorem proving system based on rewriting [13]. The heuristic used is based on the method CKD-1 but also utilises Aubin's scheme of identifying primary variables. Castaing also describes the 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 [14, 15, 16]. 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 [17] fold/unfold program transformation method [15] often require inventive definitions before being able to transform programs to their efficient counterpart. In the case of producing accumulator versions of programs from divide-and-conquer versions, these turn out to be the generalisations that we describe. A deeper understanding of the generalisation of properties

of accumulator functions is provided in Vadera [8]. We show that, under some requirements, we can guarantee the generalisation of such problems. However, the requirements make the approach difficult to automate as a general heuristic. Some interesting, but more esoteric, generalisation problems for which the existing methods fail can also be found [8]. Hesketh's thesis [18] takes a failure-driven approach to generalisation. A proof step that is stuck is first analysed to identify the offending subterms. Meta functions are then inserted in a speculative manner around the offending sub-terms in the hope that the proof will progress further. These meta functions take the form $F\_i(t, a)$, where t is an offending subterm and a is a new variable.The meta functions are restricted to be projections on the first argument (no generalisation), projections on to the second argument (generalisation by replacing a term by a variable), or of the form $f(t, a)$ where f is a constant function (to enable insertion of accumulator functions). Subsequent proof steps may then result in the identification of the meta functions by a restricted form of higher order unification with existing lemmas or the induction hypothesis. In the context of this paper, Hesketh's approach does not aim to provide a class of generalisation problems for which it will succeed. Indeed, the approach requires the use of a tentative search strategy as the meta functions may be inserted in an inappropriate place or may need to be inserted at an earlier proof step. The success of the approach also depends on the prior availability of appropriate lemmas that enable the identification of the meta functions. In the broader context, to which the thesis is addressed, the approach provides a unifying framework that uses failure to guide generalisation. Unlike the approaches presented in this paper, an important feature of Hesketh's approach is that a generalisation is explicitly motivated by failure. Generalisation is also useful in revolution-based theorem provers. The Karlsruhe Induction Theorem Proving System based on resolution and paramodulation also uses existing generalisation heuristics [19, 20].

5 Acknowledgements

The author would like to thank Tim Clement for his help and encouragement of this research; and Richard Moore and Bob Fields for help with mural as it was being developed.

A Appendix: List of problems

Example: exp3 (Manna and Waldinger [21])
Show $exp3(x, n, 1) = exp(x, n)$ where

$exp : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$
$exp(x, n) \triangleq \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } exp(x, n\text{-}1) * x$

$exp3 : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$
$exp3(x, n, z) \triangleq \textbf{if } n = 0 \textbf{ then } z \textbf{ else } exp3(x, n\text{-}1, z * x)$

Example: length (Bundy [22])
Show $length(length(x)) = length(x)$ where

$length : X^* \to \mathbf{nil}^*$
$length(x) \triangleq \textbf{if } x = [\,] \textbf{ then } [\,] \textbf{ else } \mathbf{cons}(\mathbf{nil}, length(\mathbf{tl}\, x))$

Example: gcd (Manna and Waldinger [21])
Show $gcd(x, y)$ *divides* x where

$gcd : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$
$gcd(x, y) \triangleq \textbf{if } y = 0 \textbf{ then } x \textbf{ else } gcd(y, rem(x, y))$

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

Example: nines (Manna and Waldinger [21])
Show

$$\forall ns \in No \cdot 9 \textit{ divides } v(ns) \Rightarrow 9 \textit{ divides } sum(ns)$$

where

$No = Digit^*$

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

$v : No \to \mathbf{N}$
$v(ns) \triangleq$ **if** $ns = [\,]$
then 0
else let $k =$ **len** ns-1 **in**
$(\mathbf{hd}\ \ ns) * 10^k + v(\mathbf{tl}\ \ ns)$

$sum : No \to \mathbf{N}$
$sum(ns) \triangleq$ **if** $ns = [\,]$ **then** 0 **else** $\mathbf{hd}\, ns + sum(\mathbf{tl}\, ns)$

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

$g : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$
$g(i, j) \triangleq$ **if** $i = 0$ **then** j **else** $g(i\text{-}1, 2 * i + j)$

Example: elemsofrev
Show **elems** $(greverse(l, [\,]))$ = **elems** (l) where:

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

Example: rangebt
Show:

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

where

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

References

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

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

[3] C. B. Jones. *Software Development: A Rigourous Approach*. Prentice Hall International, 1980.

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

[5] 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.

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

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

[8] Sunil Vadera. Generlisation for induction. Technical report, Univ. of Manchester Tech Report, UMCS-93-6-8, 1993.

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

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

[11] 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.

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

[13] 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.

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

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

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

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

[18] J. Hesketh. *Using middle-out reasoning to guide inductive theorem proving.* PhD thesis, 1991.

[19] 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.

[20] 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.

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

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