Automatic Verification of Functions with Accumulating Parameters

J. Functional Programming **9** (2): 225–245, March 1999. Printed in the United Kingdom

Automatic verification of functions with accumulating parameters

ANDREW IRELAND
Department of Computing & Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, Scotland
(*e-mail:* `A.Ireland@hw.ac.uk`)

ALAN BUNDY
Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland
(*e-mail:* `bundy@ed.ac.uk`)

Abstract

Proof by mathematical induction plays a crucial role in reasoning about functional programs. A generalization step often holds the key to discovering an inductive proof. We present a generalization technique which is particularly applicable when reasoning about functional programs involving accumulating parameters. We provide empirical evidence for the success of our technique and show how it is contributing to the ongoing development of a parallelizing compiler for Standard ML.

1 Introduction and motivations

Functional programs, by their very nature, are highly amenable to formal methods of reasoning. This has been exploited within the formal verification community where the majority of theorem proving based tools have a strong functional bias (Boyer and Moore, 1979; Boyer and Moore, 1988; Bundy *et al.*, 1990; Owre *et al.*, 1992; Kapur and Zhang, 1995; ORA, 1996; Hutter and Sengler, 1996; Kaufmann and Moore, 1997). Proof by mathematical induction plays a crucial role in reasoning about recursively defined functions. The generalization of an inductive conjecture often holds the key to discovering a proof. We present an automatic generalization technique which is particularly applicable when reasoning about functional programs involving accumulating parameters. We are partly motivated by a research project[1] in which a parallelizing compiler for Standard ML (SML) is being developed. This project builds directly upon previous work on the development of parallel systems from functional prototypes (Michaelson and Scaife, 1995). Transformation rules for SML will play an important part within the compilation process. It is a goal of this project to support the formal verification of these transformation rules by embedding

[1] EPSRC grant GR/L42889: *Parallelising compilation of Standard ML through prototype instrumentation and transformation.*

a theorem proving capability within the compiler. We see the work presented here as providing the basis for achieving this goal.

The paper is structured as follows. In section 2 background material on the problem and our approach are presented. An analysis of our prototype technique, what which call the 'basic critic', is given in section 3. This analysis provides the motivation for our extended technique which is documented in sections 4, 5 and 6. The implementation and testing of the extended technique are discussed in section 7, where particular attention is given to a verification obligation generated by the parallelizing compiler project mentioned above. Related and future work are outlined in sections 8 and 9, respectively. Finally, we draw our conclusions in section 10.

2 Background

2.1 Accumulator Ggeneralization

The introduction of accumulator parameters is a well documented (Henderson, 1980; Bird and Wadler, 1988; Turner, 1991; Bird, 1998) technique for deriving efficient functional programs. To illustrate the basic idea we use list reversal, a standard text book example (Henderson, 1980). Consider the following naive definition of list reversal:

$$\begin{aligned} reverse(nil) &= nil \\ reverse(X :: Y) &= app(reverse(Y), X :: nil) \end{aligned}$$

where :: and *app* denote list construction and concatenation respectively. An equivalent, but more efficient, version is derived by introducing an additional 'accumulator' parameter, *i.e.*

$$\begin{aligned} rev(nil, Z) &= Z \\ rev(X :: Y, Z) &= rev(Y, X :: Z) \end{aligned}$$

The resulting function *rev* is tail-recursive. By exploiting the direct correspondence between tail-recursion and iteration further efficiency gains can be achieved by purely mechanical means.

2.1.1 The verification problem

The correctness of the transformation given above is of obvious concern. Establishing the formal correctness, however, is not a purely mechanical process. It requires us to prove an inductive conjecture of the form:

$$\forall t : list(A).\ reverse(t) = rev(t, nil) \qquad (1)$$

In this paper, we are concerned with proving such inductive conjectures automatically. A naive attempt at proving (1) by structural induction on the list t fails. The failure occurs in the step case where we have a proof obligation of the form:

$$\underbrace{reverse(t) = rev(t, nil)}_{\textbf{hypothesis}} \vdash \underbrace{app(reverse(t), h :: nil) = rev(t, h :: nil)}_{\textbf{conclusion}}$$

Note that the conclusion fails to match the hypothesis because it contains mismatching term structures, i.e. $app(\ldots, h :: nil)$ on the left-hand-side and $h :: \ldots$ on the right-hand side. The problem is that the induction hypothesis is not strong enough, i.e. it only tells us about the behaviour of *rev* when its accumulator parameter is set to *nil*. The failed proof attempt can be overcome by generalizing the conjecture. The generalization involves the introduction of a new universally quantified variable into the conjecture, i.e.

$$\forall t : list(A).\forall l : list(A).\ app(reverse(t), l) = rev(t, l) \tag{2}$$

We refer to this as *accumulator generalization.* The generalized conjecture provides a stronger induction hypothesis which enables the step case proof to succeed. The need for generalization represents a major obstacle to the automatic verification of functional programs. A generalization step is underpinned by the cut-rule of inference. In a goal-directed framework, therefore, a generalization introduces an infinite branching point into the search space. It is known (Kreisel, 1965) that the cut-elimination theorem does not hold for inductive theories. Consequently, heuristics for controlling generalization play an important role in the automation of inductive proof.

2.1.2 Our approach

Returning to the list reversal example, the accumulator parameter provides a strong hint as to where the new universal variable should occur within the generalized conjecture. However, even with this elementary example additional guidance is required if the process is to be fully automated. For instance, how is the introduction of the $app(\ldots, l)$ term structure on the left-hand side of (2) motivated? We address this question through the use of a meta-level reasoning technique. Our starting point is a meta-level description of the common structure which characterizes an inductive proof. When a proof attempt fails this description can then be used to bridge the gap between the failure and a subsequent successful proof. We argue that having such a description provides a handle on the infinite search space generated by the generalization problem. Our approach relies upon the richness of the background theory, i.e. the lemmata which are available to the theorem prover. However, as will be shown in section 7, this is not as restrictive as it might first appear.

2.2 Proof methods and critics

We build upon the notion of a proof plan (Bundy, 1988) and tactic-based theorem proving (Gordon *et al.*, 1979). While a *tactic* encodes the low-level structure of a family of proofs a *proof plan* expresses the high-level structure. In terms of automated deduction, a proof plan guides the search for a proof. That is, given a collection of general purpose tactics the associated proof plan can be used automatically to tailor a special purpose tactic to prove a particular conjecture.

The basic building blocks of proof plans are *methods*. Using a meta-logic, methods express the preconditions for tactic application. The benefits of proof plans can be seen when a proof attempt goes wrong. Experienced users of theorem provers, such as NQTHM, are used to intervening when they observe the failure of a proof attempt. Such interventions typically result in the user generalizing their conjecture or supplying additional lemmata to the prover. Through the notion of a proof *critic* (Ireland, 1992), we have attempted to automate this process. Critics provide the proof planning framework with an exception handling mechanism which enables the partial success of a proof plan to be exploited in search for a proof. The mechanism works by allowing proof patches to be associated with different patterns of precondition failure. We previously reported (Ireland and Bundy, 1996) various ways of patching inductive proofs based upon the partial success of the ripple method described below.

2.3 Method for guiding inductive proof

In the context of mathematical induction the *ripple* method plays a pivotal role in guiding the search for a proof. The ripple method controls the selective application of rewrite rules in order to prove step case goals. Schematically, a step case goal can be represented as follows:

$$\cdots \underbrace{\forall b'.\, P[a, b']}_{\textbf{hypothesis}} \cdots \vdash \underbrace{P[c_1(a), b]}_{\textbf{conclusion}}$$

where $c_1(a)$ denotes the induction term. To achieve a step case goal the conclusion must be rewritten so as to allow the hypothesis to be applied:

$$\cdots \forall b'.\, P[a, b'] \cdots \vdash c_2(P[a, c_3(b)])$$

Note that, to apply the induction hypothesis, we must first instantiate b' to be $c_3(b)$ which gives rise to a goal of the form:

$$\cdots P[a, c_3(b)] \cdots \vdash c_2(P[a, c_3(b)])$$

The need to instantiate an inductive hypothesis in this way is commonplace in inductive proof, and plays a crucial role in our technique. We return to this point at the end of this section.

Syntactically an induction hypothesis and conclusion are very similar. More formally, the hypothesis can be expressed as an embedding within the conclusion (Smaill and Green, 1996). Restricting the rewriting of the conclusion so as to preserve this embedding maximizes the chances of applying an induction hypothesis. This is the basic idea behind the *ripple* method. The application of the ripple method, or *rippling*, makes use of meta-level annotations called *wave-fronts* to distinguish the term structures which cause the mismatch between the hypothesis and conclusion. Conversely any term structure within the conclusion which corresponds to the hypothesis is called *skeleton*. In general, embedded within each wave-front will be parts of the skeleton term structure, these are known as *wave-holes*. We use a box and an underline to represent wave-fronts and wave-holes respectively, *e.g.* an annotated version of the goal given above takes the form:

$$\cdots \forall b'.\ P[a, b'] \cdots \vdash P[\boxed{c_1(\underline{a})}^{\uparrow}, \lfloor b \rfloor]$$

We refer to a wave-front and its associated wave-hole, e.g. $\boxed{c_1(\underline{a})}^{\uparrow}$, as a *wave-term*. The arrows are used to indicate the direction in which wave-fronts can be moved through the term structure. A term structure with the annotations removed is called the *erasure*. In order to distinguish terms within the conclusion which can be matched by universal variables in the hypothesis we use annotations called *sinks*, i.e. $\lfloor \ldots \rfloor$. As will be explained below sinks play an important role in identifying the need for accumulator generalization. A successful application of the ripple method can be characterized as follows:

$$\cdots \forall b'.\ P[a, b'] \cdots \vdash \boxed{c_2(\underline{P[a, \lfloor c_3(b) \rfloor]})}^{\uparrow}$$

Note that the term $c_3(b)$, *i.e.* the instantiation for b', occurs within a sink so the wave-front annotation is no longer required. Rippling restricts rewriting to a syntactic class of rules called *wave-rules*. Wave-rules make progress towards eliminating wave-fronts while preserving skeleton term structure. A wave-rule which achieves the ripple given above takes the form[2]:

$$P[\boxed{c_1(\underline{X})}^{\uparrow}, Y] \Rightarrow \boxed{c_2(\underline{P[X, \boxed{c_3(\underline{Y})}^{\downarrow}]})}^{\uparrow} \quad (3)$$

Wave-rules are derived **automatically** from definitions and logical properties like substitution, associativity and distributivity, etc. All wave-rules are available during the process of planning a proof. In general, a successful ripple will require multiple wave-rule applications. There are three basic patterns of rippling which are summarized schematically in figure 1. The preconditions for applying wave-rules are given in figure 2.3. We draw the readers attention to precondition 4, and in particular the notion of *sinkable* wave-fronts. It is the failure of this precondition within the context of a syntactically applicable wave-rule which provides the trigger for our proof patching technique. For a complete description of rippling and the generation of wave-rules see Bundy *et al.* (1993) and Basin and Walsh (1996). To illustrate one of the basic patterns of rippling an inductive proof of conjecture (2) is presented. Structural induction on the list t gives rise to a trivial base case. We focus here on the step case where the induction hypothesis takes the form:

$$\forall l' : list(A).\ app(reverse(t), l') = rev(t, l') \quad (4)$$

and the annotated conclusion takes the form:

$$app(reverse(\boxed{h :: \underline{t}}^{\uparrow}), \lfloor l \rfloor) = rev(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor) \quad (5)$$

[2] We use $\Rightarrow$ to denote rewrite rules and $\rightarrow$ to denote logical implication.

rippling-out:

$$f_1(\ldots(f_n(\boxed{c_1(\underline{\ldots})}^{\uparrow}))\ldots) \qquad \boxed{c_n(\underline{f_1(\ldots(f_n(\ldots))\ldots)})}^{\uparrow}$$

before after

rippling-sideways:

$$f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow},\ldots,f_i(\ldots),\ldots) \qquad f_1(\ldots,\ldots,\boxed{c_i(\underline{f_i(\ldots)})}^{\downarrow},\ldots)$$

before after

rippling-in:

$$\boxed{c_n(\underline{f_1(\ldots f_n(\ldots)\ldots)})}^{\downarrow} \qquad f_1(\ldots f_n(\boxed{c_1(\underline{\ldots})}^{\downarrow})\ldots)$$

before after

An outward ripple involves the movement of wave-fronts into less nested term tree positions. A sideways ripples moves wave-fronts between distinct branches in the term tree while inward ripples movement of wave-fronts into more nested term tree positions. In general, a wave-rule may combine all three forms.

Fig. 1. The three basic rippling patterns.

The proof of the step case requires the definitions of *reverse*, *rev* and *app*, as well as the associativity of *app*. These definitions give rise to 49 wave-rules which include:

$$reverse(\boxed{X :: \underline{Y}}^{\uparrow}) \Rightarrow \boxed{app(\underline{reverse(Y)}, X :: nil)}^{\uparrow} \tag{6}$$

$$rev(\boxed{X :: \underline{Y}}^{\uparrow}, Z) \Rightarrow rev(Y, \boxed{X :: \underline{Z}}^{\downarrow}) \tag{7}$$

$$app(\boxed{app(\underline{X}, Y)}^{\uparrow}, Z) \Rightarrow app(X, \boxed{app(Y, \underline{Z})}^{\downarrow}) \tag{8}$$

Wave-rule (6) applies on the left-hand-side of (5) to give:

$$app(\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow}, \lfloor l \rfloor) = rev(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor) \tag{9}$$

Applying wave-rule (7) on the right-hand-side of (9) gives:

$$app(\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow}, \lfloor l \rfloor) = rev(t, \lfloor h :: l \rfloor)$$

Finally, wave-rule (8) applies on the left-hand-side giving:

$$app(reverse(t), \lfloor app(h :: nil, l) \rfloor) = rev(t, \lfloor h :: l \rfloor)$$

Note that the term structure delimited by the sink annotation on the left-hand-side simplifies to give:

$$app(reverse(t), \lfloor h :: l \rfloor) = rev(t, \lfloor h :: l \rfloor) \tag{10}$$

A match between (10) and (4) is achieved by instantiating l' to be $h :: l$. This completes the step case proof.

Input sequent:

$$H \vdash G[f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_2(\lfloor\ldots\rfloor), f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow}))]$$

Method preconditions:

1. there exists a subterm T of G which contains wave-front(s), *e.g.*

$$f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_2(\lfloor\ldots\rfloor), f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow}))$$

2. there exists a wave-rule which matches T, *e.g.*

$$C \rightarrow f_1(\boxed{c_1(\underline{X})}^{\uparrow}, Y, Z) \Rightarrow \boxed{c_5(\underline{f_1(X, \boxed{c_3(\underline{Y})}^{\downarrow}, \boxed{c_4(\underline{Z})}^{\downarrow})})}^{\uparrow}$$

3. the wave-rule condition follows from the context, *e.g.*

$$H \vdash C$$

4. resulting inward directed wave-fronts are potentially removable, *e.g.*

$$\ldots\underbrace{\boxed{c_3(\underline{f_2(\lfloor\ldots\rfloor)})}^{\downarrow}}_{\text{(sinkable)}}\ldots \qquad \text{or} \qquad \ldots\underbrace{\boxed{c_4(\underline{f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow})})}^{\downarrow}}_{\text{(cancellable)}}\ldots$$

Note that a wave-front is *sinkable* if it is inward directed and one or more of its wave-holes contains a sink. A wave-front is *cancellable* if it is inward directed and one or more of its wave-holes contains an outward directed wave-front.

Output sequent:

$$H \vdash G[\boxed{c_5(\underline{f_1(\ldots, \boxed{c_3(\underline{f_2(\lfloor\ldots\rfloor)})}^{\downarrow}, \boxed{c_4(\underline{f_3(\boxed{c_2(\underline{\ldots})}^{\uparrow})})}^{\downarrow})})}^{\uparrow}]$$

Note that, for a wave-rule to be applicable, both object-level and meta-level term structures must match.

Fig. 2. Preconditions for applying wave-rules.

2.4 A critic for discovering generalizations

In terms of the preconditions for applying wave-rules, the need for an accumulator generalization can be explained by the failure of precondition 4, i.e. a missing sink (see figure 2.3). Schematically this failure pattern can be characterized as follows:

$$\cdots P[a, d] \cdots \vdash P[\boxed{c_1(\underline{a})}^{\uparrow}, d]$$

where d denotes a term which does not contain any sinks. We call the occurrence of d a *blockage term* because it blocks the sideways ripple, in this case the application of wave-rule (3). The identification of a blockage term triggers the generalization critic. The associated proof patch introduces schematic terms into the goal to partially specify the occurrences of a sink variable. In the schematic example presented above this leads to a patched goal of the form:

$$\cdots \forall l'.P[a, \mathscr{M}(l')] \cdots \vdash \forall l.P[\boxed{c_1(\underline{a})}^{\uparrow}, \mathscr{M}(\lfloor l \rfloor)]$$

where $\mathscr{M}$ denotes a second-order meta-variable. Note that wave-rule (3) is now applicable, giving rise to a refined goal of the form:

$$\cdots \forall l'.P[a, \mathscr{M}(l')] \cdots \vdash \forall l.\boxed{c_2(\underline{P[a, \boxed{c_3(\underline{\mathscr{M}(\lfloor l \rfloor)})}^{\downarrow}]})}^{\uparrow}$$

The expectation is that an inward ripple will determine the identity of $\mathscr{M}$.

Relating this proof patch to the list reversal example an inductive proof of conjecture (1) gives rise to the following failure pattern:

$$\cdots reverse(t) = rev(t, nil) \cdots \vdash$$
$$\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow} = \underbrace{rev(\boxed{h :: \underline{t}}^{\uparrow}, nil)}_{\textbf{blocked}} \qquad (11)$$

Note that the occurrence of *nil* on the right-hand side is a blockage term because it prevents the application of wave-rule (7). The patched goal takes the form:

$$\cdots \forall l' : list(A).\ \mathscr{M}_2(reverse(t), l') = rev(t, \mathscr{M}_1(l')) \cdots \vdash$$
$$\mathscr{M}_2(reverse(\boxed{h :: \underline{t}}^{\uparrow}), \lfloor l \rfloor) = rev(\boxed{h :: \underline{t}}^{\uparrow}, \mathscr{M}_1(\lfloor l \rfloor)) \qquad (12)$$

Using wave-rule (6) the goal becomes:

$$\cdots \forall l' : list(A).\ \mathscr{M}_2(reverse(t), l') = rev(t, \mathscr{M}_1(l')) \cdots \vdash$$
$$\mathscr{M}_2(\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow}, \lfloor l \rfloor) = rev(\boxed{h :: \underline{t}}^{\uparrow}, \mathscr{M}_1(\lfloor l \rfloor))$$

Wave-rule (7) is now applicable and gives rise to a goal of the form:

$$\cdots \forall l' : list(A).\ \mathscr{M}_2(reverse(t), l') = rev(t, \mathscr{M}_1(l')) \cdots \vdash$$
$$\mathscr{M}_2(\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow}, \lfloor l \rfloor) = rev(t, \boxed{h :: \underline{\mathscr{M}_1(\lfloor l \rfloor)}}^{\downarrow})$$

Our approach to the problem of constraining the instantiation of schematic terms will be detailed in section 5. We will refer to the above generalization as the *basic critic.*

3 Limitations of the basic critic

The basic critic described in section 2.4 has proved very successful (Ireland and Bundy, 1996). Through our empirical testing, however, a number of limitations have been observed:

1. Certain classes of example require the introduction of multiple sink variables. The basic critic only deals with single sink variables.
2. The basic critic was designed in the context of equational proofs. A sink variable is assumed to occur on both sides of an equation. On the side

opposite to the blockage term it is assumed that in the resulting generalized term structure the sink (auxiliary) will occur as an argument of the outermost functor.

3. Sink term occurrences which are motivated by blockage terms are more constrained than those which are not. This is not exploited by the basic critic during the search for a generalization.

From these observations a number of natural extensions to the basic critic emerged. These extensions are described in the following sections.

4 Specifying sink terms

To exploit the distinction between different sink term occurrences hinted at above we extend the meta-level annotations to include the notions of *primary* and *secondary* wave-fronts. A wave-front which provides the basis for a sideways ripple, but which is not applicable because of the presence of a blockage term is designated to be *primary*. All other wave-fronts are designated to be *secondary*. To illustrate, consider the following schematic conclusion:

$$g(f(\boxed{c_1(\underline{a},b)}^{\uparrow},d),\boxed{c_1(\underline{a},b)}^{\uparrow}) \tag{13}$$

and the following wave-rules:

$$f(\boxed{c_1(\underline{X},Y)}^{\uparrow},Z) \;\Rightarrow\; f(X,\boxed{c_2(\underline{Z},Y)}^{\downarrow}) \tag{14}$$

$$g(X,\boxed{c_1(\underline{Y},Z)}^{\uparrow}) \;\Rightarrow\; \boxed{c_3(\underline{g(X,Y)},Z)}^{\uparrow} \tag{15}$$

Assuming that the occurrence of d in (13) denotes a blockage term then wave-rule (14) is not applicable. Wave-rule (15) is applicable and enables an outwards ripple, i.e.

$$\boxed{c_3(\underline{g(f(\boxed{c_1(\underline{a},b)}^{\uparrow},d),a)},b)}^{\uparrow}$$

Using subscripts[3] to denote primary and secondary wave-fronts then the analysis presented above gives rise to the following classification of the wave-fronts appearing in (13):

$$g(f(\boxed{c_1(\underline{a},b)}^{\uparrow}_{1},d),\boxed{c_1(\underline{a},b)}^{\uparrow}_{2}) \tag{16}$$

Note that the rippling of the secondary wave-fronts is undone. This increases the number of generalizations which may be subsequently discovered. Relating the notion of primary and secondary wave-fronts to blocked goal (11) gives rise to

$$reverse(\boxed{h :: \underline{t}}^{\uparrow}_{2}) = rev(\boxed{h :: \underline{t}}^{\uparrow}_{1}, nil)$$

[3] Note that wave-rules must also take account of the extension to the wave-front annotations.

4.1 Primary sink terms

For each primary wave-front an associated sink term is introduced. We refer to these as *primary sink terms*. The position of a primary sink term corresponds to the position of the blockage term within the conclusion. The structure of a primary sink term is a function of the blockage term and is computed as follows:

$$pri(X) = \begin{cases} \mathscr{M}_i(\lfloor l_i \rfloor) & \text{if } X \text{ is a constant} \\ \mathscr{M}_i(X, \lfloor l_i \rfloor) & \text{if } X \text{ is a wave-front} \\ F(pri(Y_1), \ldots, pri(Y_n)) & \text{otherwise} \\ \quad \text{where } X \equiv F(Y_1, \ldots, Y_n) & \end{cases}$$

Note that $\mathscr{M}_i$ denotes a higher-order meta-variable while l_i denotes a new object-level variable. In general distinct primary sink terms may or may not need to share the same object-level variable. This represents a choice point in the construction of primary sink terms. Assuming d denotes a constant then $pri(d)$ evaluates to $\mathscr{M}_1(\lfloor l_1 \rfloor)$. Substituting this sink term for d in (16) gives a schematic conclusion of the form:

$$g(f(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{1}, \mathscr{M}_1(\lfloor l_1 \rfloor)), \boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}) \qquad (17)$$

Relating the general notion of primary sink terms to the specific list reversal example gives:

$$reverse(\boxed{h :: \underline{t}}^{\uparrow}_{2}) = rev(\boxed{h :: \underline{t}}^{\uparrow}_{1}, \mathscr{M}_1(\lfloor l_1 \rfloor))$$

4.2 Secondary sink terms

For each secondary wave-front we eagerly attempt to apply a sideways ripple by introducing occurrences of the variables associated with the primary sink terms. These occurrences are specified again using schematic term structures and are called *secondary sink terms*. The construction of secondary sink terms are as follows. For each subterm, X, of the conclusion which contains a secondary wave-front, we compute a secondary sink term as follows:

$$sec(X) = \mathscr{M}_i(X, \lfloor l_1 \rfloor, \ldots, \lfloor l_m \rfloor)$$

where $l_1, \ldots, l_m$ denote the vector of variables generated by the construction of the primary sink terms. To illustrate, consider again the schematic conclusion (17). Taking X to be $\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}$ then the process of introducing secondary sink terms gives rise to a new schematic conclusion of the form:

$$g(f(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{1}, \mathscr{M}_1(\lfloor l_1 \rfloor)), \mathscr{M}_2(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \lfloor l_1 \rfloor)) \qquad (18)$$

Note that the selection of X represents a choice point in the construction of secondary sink terms. In the case of (17), another alternative instantiation for X exists, i.e.

$$g(\ldots, \boxed{c_1(\underline{a}, b)}^{\uparrow}_{2})$$

giving rise to a schematic conclusion of the form:

$$\mathscr{M}_2(g(f(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{1}, \mathscr{M}_1(\lfloor l_1 \rfloor)), \boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}), \lfloor l_1 \rfloor)$$

Again, relating the general notion to the specific list reversal example gives rise to two alternative patches of the form:

$$\begin{aligned} reverse(\mathscr{M}_2(\boxed{h :: \underline{t}}^{\uparrow}_{2}, \lfloor l_1 \rfloor)) &= rev(\boxed{h :: \underline{t}}^{\uparrow}_{1}, \mathscr{M}_1(\lfloor l_1 \rfloor)) \\ \mathscr{M}_2(reverse(\boxed{h :: \underline{t}}^{\uparrow}_{2}), \lfloor l_1 \rfloor) &= rev(\boxed{h :: \underline{t}}^{\uparrow}_{1}, \mathscr{M}_1(\lfloor l_1 \rfloor)) \end{aligned} \tag{19}$$

Note that the second of these corresponds to the patched goal (12).

5 Instantiating sink terms

The process of instantiating the sink terms introduced by the generalization critic is guided by the application of wave-rules. In general, the application of wave-rules in the presence of schematic term structure requires higher-order unification. Our implementation therefore exploits a higher-order unification procedure (see section 7). In this application, however, we only require second-order unification. The application of wave-rules in the presence of second-order meta-variables within the goal-term requires narrowing, i.e. rewriting where free variables in the redex can be instantiated through the unification with wave-rules. Below we describe how the meta-level annotations can be used to constrain the unification process and discuss the benefits of this approach.

5.1 Constraining second-order unification

Our procedure for constraining the application of rewrite rules within the context of skeleton term structure which contains second-order meta-variables involves three steps. The applicability of a wave-rule of the form $L \Rightarrow R$ to a wave-term W is computed as follows:

1. For each wave-front within L there exists a wave-front within W which unifies giving a substitution θ_1.
2. The erasures of L' and W' unify giving a substitution θ_2, where $L' = L \cdot \theta_1$ and $W' = W \cdot \theta_1$.
3. For each sink term T of the form $\mathscr{M}_j[\lfloor l_1 \rfloor, \ldots, \lfloor l_n \rfloor]$ within $L' \cdot \theta_2$ there exists a substitution θ_3 such that $(T \cdot \theta_3) = \lfloor l_k \rfloor$ $(1 \leqslant k \leqslant n)$.

If successful then W is replaced by $((R \cdot \theta_1) \cdot \theta_2) \cdot \theta_3$. Note that in the unification of wave-fronts both object-level and meta-level term structure must match, e.g. the wave-fronts $\boxed{c_i(X, \underline{Y})}^{\uparrow}_{N}$ and $\boxed{c_i(f(a), \underline{g(b, c)})}^{\uparrow}_{2}$ match giving rise to the following substitution $\{X \mapsto f(a), Y \mapsto g(b, c), N \mapsto 2\}$. The constraints of rippling significantly reduce the number of unifiers which are considered as will be shown in section 5.3. Our procedure does not, however, eliminate choice completely. In particular, the application of the procedure may give rise to choice with respect to the selection of wave-fronts (step 1) and sinks (step 3). We use an iterative deepening search strategy to enable alternative branches within the search space to be explored. Second-order unification will, in general, lead to a non-terminating sequence of inward directed wave-fronts. For this reason, projections are used to eagerly terminate inward ripples. A projection is applied whenever an inward directed wave-front occurs as the immediate super-term of a sink term. The strategy of eager instantiation of meta-variables may of course give rise to an over-generalization, i.e. a non-theorem. A counter-example checker is used to filter candidate instantiations of the schematic conjecture. The checker evaluates ground instances of the conjecture, typically corresponding to base cases. On detecting a non-theorem the planner backtracks and explores alternative branches within the search space. A complementary instantiation strategy is discussed in section 9 which is appropriate when meta-variables occur out with the scope of our technique.

5.2 List reversal revisited

Returning to the list reversal example, consider again patch (19) which ripples by wave-rules (6) and (7) to give:

$$\cdots \forall l' : list(A).\ \mathcal{M}_2(reverse(t), l') = rev(t, \mathcal{M}_1(l')) \cdots \vdash$$
$$\mathcal{M}_2(\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow}_{2}, \lfloor l \rfloor) = rev(t, \boxed{h :: \underline{\mathcal{M}_1 \lfloor l \rfloor}}^{\downarrow}_{2})$$

Now consider the wave-term on the left-hand side of the form:

$$\mathcal{M}_2(\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow}_{2}, \lfloor l \rfloor) \tag{20}$$

Using the annotated unification procedure wave-rule (8) now applies to give:

$$\cdots \forall l' : list(A).\ app(reverse(t), \mathcal{M}_3(t, l')) = rev(t, \mathcal{M}_1(l')) \cdots \vdash$$
$$app(reverse(t), \boxed{app(h :: nil, \underline{\mathcal{M}_3(\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow}_{2}, \lfloor l \rfloor)})}^{\downarrow}_{2})$$
$$= rev(t, \boxed{h :: \underline{\mathcal{M}_1 \lfloor l \rfloor}}^{\downarrow}_{2})$$

Note that $\mathcal{M}_2$ is instantiated to be $\lambda x.\lambda y.app(x, \mathcal{M}_3(x, y))$. By the process of eager instantiation $\mathcal{M}_1$ becomes $\lambda x.x$ and $\mathcal{M}_3$ becomes $\lambda x.\lambda y.y$ giving:

$$\cdots \forall l' : list(A).\ app(reverse(t), l') = rev(t, l') \cdots \vdash$$
$$app(reverse(t), \lfloor app(h :: nil, l) \rfloor) = rev(t, \lfloor h :: l \rfloor)$$

Simplifying the sink on the left-hand-side and instantiating l' to be $h :: l$ enables the application of induction hypothesis. Note that the resulting generalization corresponds to conjecture (2).

5.3 Benefits of meta-level guidance

Using the list reversal example we now consider the benefits of using meta-level annotations to constrain the unification process. We compare the branching rates when applying annotated and unannotated rewrite rules. As mentioned in section 2.3 the list reversal example gives rise to 49 wave-rules. In the case of goal-term (20) the annotated unification procedure eliminates all but the following 4 wave-rules:

$$\begin{array}{rcl} app(\boxed{app(\underline{X}, Y)}^{\uparrow}_{N}, Z) & \Rightarrow & app(X, \boxed{app(Y, \underline{Z})}^{\downarrow}_{N}) \\ app(X, \boxed{app(\underline{Y}, Z)}^{\uparrow}_{N}) & \Rightarrow & \boxed{app(\underline{app(X, Y)}, Z)}^{\uparrow}_{N} \\ X :: \boxed{app(\underline{Y}, Z)}^{\uparrow}_{N} & \Rightarrow & \boxed{app(\underline{X :: Y}, Z)}^{\uparrow}_{N} \\ \boxed{app(\underline{reverse(Y)}, X :: nil)}^{\uparrow}_{N} & \Rightarrow & reverse(\boxed{X :: \underline{Y}}^{\downarrow}_{N}) \end{array} \qquad (21)$$

Note that only the first three of these will actually apply since the third is ruled-out by precondition 4 of the ripple method, i.e. sink-ability. The 3 remaining applicable wave-rules should then be compared with the results of unannotated unification which again gives rise to 18 applicable rewrite rules.

While the annotations reduce the number of wave-rules considered for unification they also constrain the number of unifiers. To illustrate, consider goal-term (20) and the left-hand side of wave-rule (21). Unification without the constraints of annotations generates two possible unifiers, i.e. $\lambda x.\lambda y.app(x, \mathscr{M}_3(x, y))$ and $\lambda x.\lambda y.app(h :: t, \mathscr{M}_2(x, y))$. Note that the first is based upon projection while the second uses imitation. The imitation, however, violates the key property of rippling, i.e. skeleton preservation (see section 2.3), so is rejected by the annotated unification procedure.

6 Organizing the search space

In controlling the search for a generalization we place a number of constraints on the proof planning process:

- Planning in the context of schematic term structures requires a bounded search strategy. We use an iterative deepening strategy to explore the space of alternative ripple proofs.
- Backtracking over the construction of sink terms deals with the choice point issues raised in section 4.
- Since primary sink terms are more constrained than secondary sink terms priority is given to the rippling of primary wave-fronts.

7 Implementation and testing

The extensions to the basic critic described above directly address the limitations highlighted in section 3:

1. The linkage of blockage terms with the introduction of primary sink terms within the schematic conjecture addresses the issue of multiple sink variables.
2. The issue of positioning auxiliary sink variables is dealt with by the ability to revise the construction of secondary sink terms.
3. By extending the meta-logic to include the notions of primary and secondary wave-fronts we are able to exploit the observation that certain sink terms are more constrained than others during the search for generalizations.

Our extended critic has been implemented and integrated within the CLAM proof planner (Bundy *et al.*, 1990). The implementation makes use of the higher-order features of λ-Prolog (Miller and Nadathur, 1988).

The results presented in Ireland and Bundy (1996) for the basic critic were replicated by the extended critic. The extended critic, however, discovered generalizations which the basic critic missed. Moreover, a number of new examples were generalized by the extended critic for which the application of the basic critic resulted in failure. Our results are documented in the tables given in Appendix B. The example conjectures for which the extended critic improves upon the performance of the basic critic are presented in Table 1. All the examples require accumulator generalization and therefore cannot be proved automatically by other inductive theorem provers such as NQTHM (Boyer and Moore, 1979; Boyer and Moore, 1988). The correspondence between conjectures and generalized conjectures is recorded in Table 2. The time taken to discover each generalization using the extended critic is also given in Table 2. The lemmata used in motivating the generalizations are presented in Table 3, while the actual generalized conjectures are given in Table 4. All the generalized conjectures are computed automatically. Our technique relies upon the existence of appropriate lemmata. However, as can be seen seen from Table 3, the lemmata are relatively general purpose, i.e. properties such as associativity and distributivity. Moreover, we have previously shown how our approach to failure analysis has enabled us to automatically generate such lemmata (Ireland and Bundy, 1996). This gives the opportunity for lemmata discovered during one part of a proof effort to be used to motivate a generalization within another.

To place our contribution within the wider context of functional programming, we focus upon conjecture C10 (see Table 1), which arose within the parallelising compiler project mentioned in section 1. C10 is the proof obligation generated by the verification of a SML transformation rule which specifies an equivalence between a single and a distributed application of the *map* function, i.e.

$$\forall t : list(A).\forall f : A \to B.\forall n : \mathbb{N}.$$
$$map(f,t) = reduce(\lambda x.\lambda y.app(x,y), map(\lambda x.map(f,x), split_1(1,n,nil,t))) \quad (22)$$

Such equivalences enable the correspondence which exists between higher-order functions and generic parallel constructs to be exploited during the parallelisation of SML code. The definitions associated with (22) are included as rewrite rules within Appendix A while the corresponding SML code is given in figure 3. An inductive proof of (22) requires an accumulator generalization. Our extended critic

```
fun atend x nil    = (x::nil) |      fun split x y = split1 1 x nil y;
    atend x (y::z) = y::(atend x z); val split = fn:nat -> 'a list
val atend = fn:'a -> 'a list ->                  -> 'a list list
            -> 'a list               fun app nil    z = z |
                                         app (x::y) z = x::(app y z);
fun split1 v w x nil  = (x::nil) |   val app = fn:'a list -> 'a list
    split1 v w x (y::z) =                   -> 'a list
    if (v > w)                       fun map x nil    = nil |
    then                                 map x (y::z) = (x y)::(map x z);
       x::(split1 2 w (y::nil) z)    val map = fn:('a -> 'b) -> 'a list
    else                                    -> 'b list
       (split1 v+1 w (atend y x) z); fun reduce x nil    = nil |
val split1 = fn:nat -> nat               reduce x (y::z) =
             -> 'a list -> 'a list              (x y (reduce x z));
             -> 'a list list         val reduce = fn:('a -> 'b list ->
                                               'b list) -> 'a list ->
                                               'b list
```

Fig. 3. SML list processing functions.

generates a schematic conjecture of the form:

$$
\begin{aligned}
&\forall f : A \to B.\forall n : \mathbb{N}.\forall l_1 : \mathbb{N}.\forall l_2 : list(A)\\
&\quad map(f, \mathcal{M}_3(t, l_1, l_2)) =\\
&\qquad reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(f, x), split_1(\mathcal{M}_1(l_1), n, \mathcal{M}_2(l_2), t)))
\end{aligned}
$$

The subsequent proof planning instantiates this schematic conjecture giving rise to a generalized conjecture of the form:

$$
\begin{aligned}
&\forall t : list(A).\forall f : A \to B.\forall n : \mathbb{N}.\forall l_1 : \mathbb{N}.\forall l_2 : list(A).\\
&\quad map(f, app(l_2, t)) =\\
&\qquad reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(f, x), split_1(l_1, n, l_2, t))) \qquad (23)
\end{aligned}
$$

Note that the generalization involves the introduction of two new universally quantified variables l_1 and l_2. To summarize, the ripple method in conjunction with the extended critic is able to automatically generate and verify (23) by analysing the failure to prove (22) directly.

8 Related work

In Aubin's thesis (Aubin, 1976) he presents a technique for discovering accumulator generalizations based upon the failure of an unfolding strategy. Basically he used the mismatch between the conclusion and hypothesis to suggest the introduction of what we call primary sinks. With regard to secondary sinks, Aubin appeals to a notion of an equation being 'balanced', i.e. a sink should occur on both sides of an equality.

Hesketh, in her thesis (Hesketh, 1991), tackled the problem of accumulator generalization in the context of proof planning and rippling. Her approach, however, did not deal with multiple sinks. By introducing the primary and secondary classification of wave-fronts we believe that our approach provides greater control in the search for generalizations. This becomes crucial as the complexity[4] of examples increases. In addition, we use sink annotations explicitly in selecting potential projections for higher-order meta-variables. Hesketh's work, however, was much broader than ours in that she unified a number of different kinds of generalization. Moreover, she was also able to synthesize tail-recursive functions given equivalent naive recursive definitions (Hesketh *et al.*, 1992).

An alternative to our approach of annotated unification is presented in (Hutter and Kohlhase, 1997) where essentially the structure preservation constraints of rippling are embedded within the unification algorithm. This approach, however, has not been applied to the problem of generalization so a direct comparison is not possible.

9 Future work

A limitation of the technique as implemented is that it only deals with wave-fronts which contain single wave-holes. This restricts us to proofs which involve a single induction hypothesis. In principle, we see no reason why this restriction should not be removed in the future.

One of the goals of parallelizing SML compiler project is the automatic synthesis of missing transformation rules. We see the work presented here as a starting point for this synthesis task.

Our technique is not restricted to reasoning about functional programs. For instance, we believe that it subsumes the procedure described by Pierre (1995) for generalizing hardware specifications. In addition, by exploiting the close relationship which exists between induction and iteration we have shown (Ireland and Stark, 1997) how our generalization critic can play a role in the automatic discovery of *tail invariants* (Kaldewaij, 1990). We plan to investigate these connections further.

The critic mechanism was motivated by a desire to build an automatic theorem prover which was more robust than conventional provers. We believe, however, that the critic mechanism also provides a basis for developing effective user interaction. An interactive version of the critic mechanism has been implemented (Ireland *et al.*, 1997) which invites a user to complete the instantiation of meta-variables. This represents ongoing work which, as observed in section 5.1, complements the generalization technique presented here.

10 Conclusion

The search for inductive proofs cannot avoid the problem of generalization. In this paper we describe extensions to a proof critic for automatically generalizing inductive conjectures. The ideas presented here build upon a technique for patching proofs reported in Ireland and Bundy (1996). These extensions have significantly

[4] That is, as the number of definitions and lemmata available to the prover increases.

improved the performance of the technique while preserving the spirit of original proof patch. Our implementation of the extended critic has been tested on the verification of functional programs with some promising results. More generally, we believe that our technique has wider application in terms of both software and hardware verification.

Acknowledgements

The research reported in this paper was supported by EPSRC grants GR/J80702, GR/L11724, GR/L42889 as well as ARC grant 438. We would like to thank Greg Michaelson for providing example conjectures and Lincoln Wallen for drawing our attention to the connection between tail invariants and our generalization critic. Thanks also go to David Basin, Alan Smaill, Maria M^{c}Cann, Julian Richardson, Toby Walsh, Richard Boulton, Dieter Hutter and anonymous CADE-13 and JFP referees for their constructive feedback on this paper. An earlier shorter version of this paper appeared in the proceedings of CADE-13.

Appendix A: Definitional rewrite rules[5]

$$
\begin{array}{rcl}
reverse(nil) & \Rightarrow & nil \\
reverse(X :: Y) & \Rightarrow & app(reverse(Y), X :: nil) \\
rev(nil, Z) & \Rightarrow & Z \\
rev(X :: Y, Z) & \Rightarrow & rev(Y, X :: Z) \\
atend(X, nil) & \Rightarrow & X :: nil \\
atend(X, Y :: Z) & \Rightarrow & Y :: atend(X, Z) \\
map(X, nil) & \Rightarrow & nil \\
map(X, Y :: Z) & \Rightarrow & X(Y) :: map(X, Z) \\
reduce(X, nil) & \Rightarrow & nil \\
reduce(X, Y :: Z) & \Rightarrow & X(Y, reduce(X, Z)) \\
foldr(W, X, nil) & \Rightarrow & X \\
foldr(W, X, Y :: Z) & \Rightarrow & W(Y, foldr(W, X, Z)) \\
filter(X, nil) & \Rightarrow & nil \\
X(Y) \rightarrow filter(X, Y :: Z) & \Rightarrow & Y :: filter(X, Z) \\
\neg X(Y) \rightarrow filter(X, Y :: Z) & \Rightarrow & filter(X, Z) \\
sum(nil) & \Rightarrow & 0 \\
sum(X :: Y) & \Rightarrow & sum(Y) + X \\
prod(nil) & \Rightarrow & 1 \\
prod(X :: Y) & \Rightarrow & prod(Y) * X \\
tsum(nil, Z) & \Rightarrow & Z \\
tsum(X :: Y, Z) & \Rightarrow & tsum(Y, Z + X)
\end{array}
$$

[5] We assume standard recursive definitions for *even* and *odd* as well as for list concatenation (*app*), deletion (*del*) and membership (*mem*).

$$\begin{array}{rcl}
tprod(nil, Z) & \Rightarrow & Z \\
tprod(X :: Y, Z) & \Rightarrow & tprod(Y, Z * X) \\
sp(nil, Y, Z) & \Rightarrow & \langle Y, Z \rangle \\
sp(W :: X, Y, Z) & \Rightarrow & sp(X, W + Y, W * Z) \\
sp_2(nil, Y, Z) & \Rightarrow & \langle Y, Z \rangle \\
sp_2(W :: X, Y, Z) & \Rightarrow & sp_2(X, Y + W, Z * W) \\
evenel(nil) & \Rightarrow & nil \\
odd(X) \rightarrow evenel(X :: Y) & \Rightarrow & evenel(Y) \\
even(X) \rightarrow evenel(X :: Y) & \Rightarrow & X :: evenel(Y) \\
oddel(nil) & \Rightarrow & nil \\
odd(X) \rightarrow oddel(X :: Y) & \Rightarrow & X :: oddel(Y) \\
even(X) \rightarrow oddel(X :: Y) & \Rightarrow & oddel(Y) \\
perm(nil, nil) & \Rightarrow & true \\
perm(nil, X :: Y) & \Rightarrow & false \\
perm(X :: Y, Z) & \Rightarrow & (perm(Y, del(X, Z)) \wedge mem(X, Z)) \\
partition(nil, Y, Z) & \Rightarrow & app(Y, Z) \\
even(W) \rightarrow partition(W :: X, Y, Z) & \Rightarrow & partition(X, atend(W, Y), Z) \\
odd(W) \rightarrow partition(W :: X, Y, Z) & \Rightarrow & partition(X, Y, atend(W, Z)) \\
split_1(V, W, X, nil) & \Rightarrow & X :: nil \\
V > W \rightarrow split_1(V, W, X, Y :: Z) & \Rightarrow & X :: split_1(2, W, Y :: nil, Z) \\
V \leqslant W \rightarrow split_1(V, W, X, Y :: Z) & \Rightarrow & split_1(V + 1, W, atend(Y, X), Z) \\
split(X, Y) & \Rightarrow & split_1(1, X, nil, Y)
\end{array}$$

Appendix B: Experimental results

Table 1. *Conjectures*

No	Conjecture
C1	$reverse(X) = rev(X, nil)$
C2	$rev(rev(X, nil), nil) = reverse(reverse(X))$
C3	$perm(reverse(X), rev(X, nil))$
C4	$rev(rev(X, nil), nil) = reverse(reduce(\lambda x.\lambda y.atend(x, y), X))$
C5	$app(evenel(X), oddel(X)) = partition(X, nil, nil)$
C6	$app(filter(\lambda x.even(x), X), filter(\lambda x.odd(x), X)) = partition(X, nil, nil)$
C7	$sp(X, 0, 1) = \langle sum(X), prod(X) \rangle$
C8	$\langle tsum(X, 0), tprod(X, 1) \rangle = \langle foldr(\lambda x.\lambda y.(x + y), 0, X), foldr(\lambda x.\lambda y.(x * y), 1, X) \rangle$
C9	$sp_2(X, 0, 1) = \langle foldr(\lambda x.\lambda y.(x + y), 0, X), foldr(\lambda x.\lambda y.(x * y), 1, X) \rangle$
C10	$map(F, X) = reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(F, x), split_1(1, W, nil, X)))$

Table 2. *Performance of the extended generalization critic*

No	Generalizations (Timings)
C1	G1 (7.9) G2 (7.7)
C2	G3 (25.0) G4 (17.1) G5 (105.8) G6 (15.8) G7 (14.3) G8 (16.5) G9 (11.4) G10 (15.3)
C3	G11 (8.7) G12 (7.4) G13 (7.6)
C4	G14 (10.2)
C5	G15 (108.1)
C6	G16 (95.4)
C7	G17 (24.7)
C8	G18 (42.9)
C9	G19 (30.1)
C10	G20 (68.3)

The timings are given in CPU seconds and were obtained using a SICSTUS implementation of CIAM running on a Sun ULTRA-SPARC. The figures represent the time taken to compute the alternative instantiations of each conjecture schema. Note that in the case of C1 and C2 the basic critic does not discover G2, G8 and G9 while it fails completely on conjectures C3 through to C10. However, the extended critic succeeds on all the conjectures given in Table 1.

Table 3. *Lemmata used to motivate generalizations*

No	Lemma
L1	$app(app(X,Y),Z) = app(X,app(Y,Z))$
L2	$app(app(X,Y :: nil),Z) = app(X,Y :: Z)$
L3	$reverse(app(X,Y :: nil)) = Y :: reverse(X)$
L4	$app(X,Y :: Z) = app(atend(Y,X),Z)$
L5	$X + (Y + Z) = (X + Y) + Z$
L6	$X * (Y * Z) = (X * Y) * Z$
L7	$map(W,app(X,Y :: Z)) = app(map(W,X),map(W,app(Y :: nil,Z)))$

Table 4. *Generalized conjectures*

No	Generalization	Lemmata
G1	$app(reverse(X), Y) = rev(X, Y)$	L1
G2	$reverse(rev(Y, X)) = rev(X, Y)$	
G3	$rev(rev(X, Y), nil) = app(reverse(Y), reverse(reverse(X)))$	L2&L3
G4	$rev(rev(X, Y), nil) = rev(Y, reverse(reverse(X)))$	L3
G5	$rev(rev(X, Y), nil) = rev(reverse(reverse(Y)), reverse(reverse(X)))$	L3
G6	$rev(rev(X, reverse(Y)), nil) = app(Y, reverse(reverse(X)))$	L2&L3
G7	$rev(rev(X, reverse(reverse(Y))), nil) = rev(Y, reverse(reverse(X)))$	L3
G8	$rev(rev(X, reverse(Y)), nil) = rev(reverse(Y), reverse(reverse(X)))$	L3
G9	$rev(rev(X, Y), nil) = reverse(app(reverse(X), Y))$	L1
G10	$rev(rev(X, Y), nil) = reverse(reverse(rev(Y, X)))$	
G11	$perm(reverse(rev(X, Y)), rev(X, Y))$	
G12	$perm(reverse(rev(Y, X)), rev(X, Y))$	
G13	$perm(app(reverse(X), Y), rev(X, Y))$	L1
G14	$rev(rev(X, Y), nil) = reverse(app(reduce(\lambda x.\lambda y.atend(x, y), X), Y))$	L4
G15	$app(app(Y, evenel(X)), app(Z, oddel(X))) = partition(X, Y, Z)$	L4
G16	$app(app(Y, filter(\lambda x.even(x), X)), app(Z, filter(\lambda x.odd(x), X))) =$ $partition(X, Y, Z)$	L4
G17	$sp(X, Y, Z) = \langle sum(X) + Y, prod(X) * Z\rangle$	L5&L6
G18	$\langle tsum(X, Y), tprod(X, Z)\rangle =$ $\langle Y + foldr(\lambda x.\lambda y.(x + y), 0, X), Z * foldr(\lambda x.\lambda y.(x * y), 1, X)\rangle$	L5&L6
G19	$sp_2(X, Y, Z) =$ $\langle Y + foldr(\lambda x.\lambda y.(x + y), 0, X), Z * foldr(\lambda x.\lambda y.(x * y), 1, X)\rangle$	L5&L6
G20	$map(F, app(Y, X)) = reduce(\lambda x.\lambda y.app(x, y),$ $map(\lambda x.map(F, x), split_1(Z, W, Y, X)))$	L4&L7

The lemmata used to suggest generalizations are indicated in the third column. No entry appears if the generalization was discovered using purely definitional rewrite rules.

References

Aubin, R. (1976) *Mechanizing structural induction.* PhD thesis, University of Edinburgh.

Basin, D. and Walsh, T. (1996). *A calculus for and termination of rippling. J. Automated Reasoning*, **16**(1–2), 147–180.

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

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

Boyer, R. S. and Moore, J. S. (1979) *A Computational Logic.* Academic Press (ACM monograph series).

Boyer, R. S. and Moore, J. S. (1988) *A Computational Logic Handbook.* Academic Press (Perspectives in Computing, Vol 23).

Bundy, A. (1988) The use of explicit plans to guide inductive proofs. In: Lusk, R. and Overbeek, R. (eds), *9th Conference on Automated Deduction*, pp. 111–120. Springer-Verlag.

Bundy, A., van Harmelen, F., Horn, C. and Smaill, A. (1990) The Oyster-Clam system. In: Stickel, M. E. (ed), *10th International Conference on Automated Deduction: Lecture Notes in Artificial Intelligence 449.*, pp. 647–648. Springer-Verlag.

Bundy, A, Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1993) Rippling: A heuristic for guiding inductive proofs. *Artificial intelligence*, **62**, 185–253.

Gordon, M. J., Milner, A. J. and Wadsworth, C. P. (1979) *Edinburgh LCF - a mechanised logic of computation: Lecture Notes in Computer Science 78.* Springer-Verlag.

Henderson, P. (1980) *Functional Programming.* Prentice Hall.

Hesketh, J., Bundy, A. and Smaill, A. (1992) Using middle-out reasoning to control the synthesis of tail-recursive programs. In: Kapur, D. (ed), *11th Conference on Automated Deduction: Lecture Notes in Artificial Intelligence 607*, pp. 310–324. Springer-Verlag.

Hesketh, J. T. (1991) *Using middle-out reasoning to guide inductive theorem proving.* PhD thesis, University of Edinburgh.

Hutter, D. and Kohlhase, M. (1997) A colored version of the λ-Calculus. In: McCune, W. (ed), *14th Conference on Automated Deduction*, pp. 291–305. Springer-Verlag.

Hutter, D. and Sengler, C. (1996) INKA: The Next Generation. In: McRobbie, M. A. and Slaney, J. K. (eds), *Proceedings of CADE-13: Lecture Notes in Artificial Intelligence 1104.* Springer-Verlag.

Ireland, A. (1992) The Use of Planning Critics in Mechanizing Inductive Proofs. In: Voronkov, A. (ed), *International Conference on Logic Programming and Automated Reasoning – IPAR 92: Lecture Notes in Artificial Intelligence 624*, pp. 178–189. St. Petersburg. Springer-Verlag.

Ireland, A. and Bundy, A. (1996) Productive Use of Failure in Inductive Proof. *J. Automated Reasoning*, **16**(1–2), 79–111.

Ireland, A. and Stark, J. (1997) On the Automatic Discovery of Loop Invariants. *Proc. 4th NASA Langley Formal Methods Workshop.* NASA Conference Publication 3356.

Ireland, A., Jackson, M. and Reid, G. (1997) A collaborative approach to theorem proving. *Proc. 1st International Workshop on Proof Transformation and Presentation (PTP-97).*Schloss Dagstuhl, Germany.

Kaldewaij, A. (1990) *Programming: The derivation of algorithms.* Prentice Hall.

Kapur, D. and Zhang, H. (1995) An Overview of Rewrite Rule Laboratory (RRL). *J. Computer & Mathematics with Applic.*, **29**(2), 91–114.

Kaufmann, M. and Moore, J. (1997) An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. Software Eng.*, **23**(4), 203–213.

Kreisel, G. (1965) Mathematical logic. In: Saaty, T. L. (ed), *Lectures on Modern Mathematics, vol III*, pp. 95–195. Wiley.

Michaelson, G. and Scaife, N. (1995) Prototyping a parallel vision system in Standard ML. *J. Functional Programming*, **5**, 345–382.

Miller, D. and Nadathur, G. (1988) An overview of λProlog. In: Bowen, K. and Kowalski, R. (eds), *Proc. 5th International Logic Programming Conference/5th Symposium on Logic Programming.* MIT Press.

ORA (1996) Introduction to EVES: Eercises and Notes. *Odyssey Research Associates, Ottawa Ontario, Canada.*

Owre, S., Rushby, J. M. and Shankar, N. (1992) *PVS: An integrated approach to specification and verification.* Forthcoming. SRI International.

Pierre, L. (1995) An automatic generalization method for the inductive proof of replicated and parallel architectures. In: Kropf, R. and Kumar, T. (eds), *Theorem Provers in Circuit Design: Lecture Notes in Computer Science 901.* Springer-Verlag.

Smaill, A. and Green, I. (1996) Higher-Order Annotated Terms for Proof Search. *Proceedings of TPHOLS'96.*

Turner, R. (1991) *Constructive Foundations for Functional Languages.* McGraw-Hill.