Submission to J. Functional Programming Special Issue on Theorem Proving & Functional Programming

Automatic Verification of Functions with Accumulating Parameters

ANDREW IRELAND

Department of Computing & Electrical Engineering,
Heriot-Watt University, Riccarton,
Edinburgh, Scotland EH14 4AS,
A.Ireland@hw.ac.uk

ALAN BUNDY

Department of Artificial Intelligence,
University of Edinburgh, 80 South Bridge,
Edinburgh, Scotland EH1 1HN,
bundy@ed.ac.uk

Contents

Abstract

Proof by mathematical induction plays a crucial role in the verification of program transformations. This paper focuses on the automatic verification of transformations which introduce accumulating parameters. Such verification efforts typically require a generalization step. In earlier papers we presented a technique for automating the generalization step by analysing failed proof attempts. Through empirical testing a natural generalization and extension of the basic technique emerged. Here we describe our extended generalization technique together with some promising experimental results.

1 Introduction

The introduction of accumulator parameters is a well documented (Henderson, 1980; Bird & Wadler, 1988; Turner, 1991) technique for deriving efficient functional programs. In order 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. Establishing the correctness of this transformation, however, is not a purely mechanical process. It requires us to prove an inductive conjecture of the form:

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

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 an induction hypothesis of the form:

$$reverse(t) = rev(t, nil)$$

and an inductive conclusion of the form:

$$app(reverse(t), h :: nil) = rev(t, h :: nil)$$

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 behavior 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 will 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 automation of proof by mathematical induction. A generalization step is under pinned 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.

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.

2 Background

2.1 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 & Bundy, 1996) various ways of patching inductive proofs based upon the partial success of the ripple method described below.

2.2 A 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:

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

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

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

Induction and recursion are closely related. The application of an induction hypothesis corresponds to a recursive call while the instantiation of an induction hypothesis corresponds to the modification of an accumulator parameter. The need to instantiate induction hypotheses is commonplace in inductive proof. Our technique, as will be explained below, exploits this fact.

Syntactically an induction hypothesis and conclusion are very similar. More formally, the hypothesis can be expressed as an embedding within the conclusion. 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 will 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

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

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†:

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

Wave-rules are derived automatically from definitions and logical properties like substitution, associativity and distributivity *etc.* In general, a successful ripple will require multiple wave-rule applications. There are three basic patterns of rippling which are summariesed schematically in figure 1. The preconditions for applying wave-rules are given in figure (2). For a complete description of rippling see (Bundy *et al.*, 1993; Basin & Walsh, 1994). To illustrate one of the basic patterns of rippling an inductive proof of conjecture (2) is presented. Structural induction on the list t

† We use $\Rightarrow$ to denote rewrite rules and $\rightarrow$ to denote logical implication.

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.*

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

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 in order for a wave-rule to be applicable both object-level and meta-level term structures must match.

Fig. 2. Preconditions for applying wave-rules

gives rise to a trivial base case. We focus here on the step case where the induction hypothesis takes the form:

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

and the annotated conclusion takes the form:

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

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:

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

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

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

Note that all wave-rules are available during the process of planning a proof. Wave-rule (6) applies on the left-hand-side of (5) to give:

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

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

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

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:

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

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

2.3 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). Schematically this failure pattern can be characterised 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 in order to partially specify the occurrences of a sink variable. In the schematic example presented above this leads to a patched goal of the form:

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

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

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

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

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:

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

Using wave-rule (6) the goal becomes:

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

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

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

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

3 Limitations of the Basic Critic

The basic critic described in §2.3 has proved very successful (Ireland & 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

In order 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:

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

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

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[‡] 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:

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

[‡] 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:

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

Note that $\mathcal{M}_i$ 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 $\mathcal{M}_1(\lfloor l_1 \rfloor)$. Substituting this sink term for d in (16) gives a schematic conclusion of the form:

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

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

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

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:

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

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:

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

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:

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

Again relating the general notion to the specific list reversal example gives rise to 2 alternative patches of the form:

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

$$\mathcal{M}_2(reverse(\boxed{h :: \underline{t}}^{\uparrow}_{2}), \lfloor l_1 \rfloor) = rev(\boxed{h :: \underline{t}}^{\uparrow}_{1}, \mathcal{M}_1(\lfloor l_1 \rfloor)) \tag{20}$$

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 §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 show in detail how the meta-level annotations can be used to constrain the unification process and discuss the benefits of this approach.

5.1 Guiding Second-Order Unification

Our method for constraining the application of rewrite rules within the context of skeleton term structure which contains second-order meta-variables is presented in figure 3. Second-order unification will, in general, lead to a non-terminating sequence of wave-rule applications. For this reason projections are used to eagerly terminate inward ripples. A projection is applied whenever the immediate superterm of a sink term is an inward directed wave-front. An alternative to our eager instantiation strategy is discussed in §9. 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 critic mechanism backtracks and attempts further rippling. A limitation of our method for instantiating sink terms is that it only deals with wave-fronts which contain single wave-holes.

Goal-term (before):

$$\mathcal{M}_1(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \lfloor l_1 \rfloor)$$

Wave-rule:

$$f(\lfloor W \rfloor, \boxed{c_1(\underline{X}, Y)}^{\uparrow}_{N}, Z) \Rightarrow f(\lfloor c_2(Y) \rfloor, X, \boxed{c_3(\underline{Z}, Y)}^{\downarrow}_{N})$$

Guidance:

1. Unify all wave-terms within the left-hand-side of the selected wave-rule with wave-terms within the selected goal-term. Success requires a match between wave-directions, wave-holes and wave-terms. The process is recursive and works from the inside out, *e.g.*

$$f(\lfloor W \rfloor, \boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, Z) \Rightarrow f(\lfloor c_2(b) \rfloor, a, \boxed{c_3(\underline{Z}, b)}^{\downarrow}_{2})$$

2. Unify the erasure of the left-hand-side of the wave-rule with the erasure of the goal-term, *e.g.*

$$f(\left\lfloor \mathcal{M}_2(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \lfloor l_1 \rfloor) \right\rfloor, \boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \mathcal{M}_3(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \lfloor l_1 \rfloor)) \Rightarrow$$

$$f(\lfloor c_2(b) \rfloor, a, \boxed{c_3(\underline{\mathcal{M}_3(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \lfloor l_1 \rfloor)}, b)}^{\downarrow}_{2})$$

3. Match all sinks which appear within the left-hand-side of the selected wave-rule with sinks within the selected goal-term.

$$f(\lfloor l_1 \rfloor, \boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \mathcal{M}_3(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \lfloor l_1 \rfloor)) \Rightarrow f(\lfloor c_2(b) \rfloor, a, \boxed{c_3(\underline{\mathcal{M}_3(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \lfloor l_1 \rfloor)}, b)}^{\downarrow}_{2})$$

Goal-term (after):

$$f(\lfloor c_2(b) \rfloor, a, \boxed{c_3(\underline{\mathcal{M}_3(\boxed{c_1(\underline{a}, b)}^{\uparrow}_{2}, \lfloor l_1 \rfloor)}, b)}^{\downarrow}_{2})$$

Note that in step 3 there may exist multiple sinks giving rise to alternative projections. Each projection results in a distinct ripple. Our use of an iterative deepening search strategy during the planning process enables each alternative ripple to be explored. Although illustrated in terms of an outward directed wave-front, the process described above is also applicable to inward directed wave-fonts.

Fig. 3. Annotated Second-Order Unification

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:

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

Note that the wave-front on the left-hand-side is now classified as secondary. Whenever the movement of a wave-front requires second-order unification then it is classified as secondary. Since both wave-fronts are classified as secondary then either can be rippled at this stage. Consider the wave-term on the left-hand-side:

$$\mathcal{M}_2(\boxed{h :: \underline{t}}^{\uparrow}_{2}, \lfloor l \rfloor) \tag{21}$$

Using the annotated unification process described above wave-rule (7) applies and gives rise to:

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

Note that $\mathcal{M}_2$ is instantiated to be $\lambda x.\lambda y.rev(\mathcal{M}_3(x, y), x)$. By the process of eager instantiation, mentioned in §5.1, $\mathcal{M}_1$ becomes $\lambda x.x$ and $\mathcal{M}_3$ to be $\lambda x.\lambda y.y$. This gives rise to a goal of the form:

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

The induction hypothesis can now be applied by instantiating l' to be $h :: l$. The resulting generalization corresponds to:

$$\forall t : list(A).\forall l : list(A).\ reverse(rev(l, t)) = rev(t, l)$$

Alternatively, if patch (20) is selected then wave-rules (6) and (7) apply 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:

$$\mathcal{M}_2(\boxed{app(\underline{reverse(t)}, h :: nil)}^{\uparrow}_{2}, \lfloor l \rfloor) \tag{22}$$

Using the annotated unification process 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 $\lambda x.\lambda y.app(x, \mathcal{M}_3(x, y))$. Again by the process of eager instantiation $\mathcal{M}_1$ becomes $\lambda x.x$ and $\mathcal{M}_3$ to be $\lambda x.\lambda y.y$ giving:

$$\begin{array}{l} \cdots \forall l' : list(A).\ app(reverse(t), l') = rev(t, l') \cdots \vdash \\ \qquad\qquad app(reverse(t), \lfloor app(h :: nil, l) \rfloor) = rev(t, \lfloor h :: l \rfloor) \end{array}$$

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 (2).

5.3 The Benefits of Meta-level Guidance

Using the list reversal example we now consider the benefits of using meta-level annotations to guide the unification process. We compare the branching rates when applying annotated and unannotated rewrite rules. As mentioned in §2.2 this example gives rise to 49 wave-rules.

Firstly, consider again (21), for this goal-term the annotated unification process eliminates all but the following 4 wave-rules:

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

This should be compared with unannotated unification which gives rise to 18 applicable rewrite rules.

Secondly, in the case of (22) the annotated unification process again eliminates all but 4 wave-rules, *i.e.*

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

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 again the unification of goal-term (22) and the left-hand-side of wave-rule (8). Second-order unification generates two possible unifiers, *i.e.* $\lambda x.\lambda y.app(x, \mathcal{M}_3(x, y))$ and $\lambda x.\lambda y.app(h :: t, \mathcal{M}_2(x, y))$. Note that the first is based upon projection the second uses imitation. The imitation, however, violates the key property of rippling, *i.e.* skeleton preservation (see §2.2), so is rejected by the annotated unification process.

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 based upon the length of ripple paths. Given a wave-front, its associated *ripple paths* are defined to be the sequence(s) of term tree positions which can be reached by the application of wave-rules. The length of a particular ripple path is defined to be the number of wave-rule applications used in its construction.
- Backtracking over the construction of secondary sink terms deals with the choice point issue raised in §4.2.
- 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 §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 & Nadathur, 1988). Below we document the testing of our implementation.

7.1 Experimental Results

The results presented in (Ireland & 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 C. The example conjectures for which the extended critic improves upon the performance of the basic critic are presented in table I. All the examples require accumulator generalization and therefore cannot be proved automatically by other inductive theorem provers such as NQTHM (Boyer & Moore, 1979). The relative performance of the basic and extended critics on the example conjectures is recorded in table II. The lemmata used in motivating the generalizations are presented in table III while the actual generalized conjectures are given in table IV. All these generalizations are discovered automatically, *i.e.* no user intervention.

7.2 A Case Study

We now illustrate our generalization technique using a more realistic example. Consider the functions defined in figure 4. Rewrite rules derived from these definitions

```
fun atend x nil    = (x::nil) |
    atend x (y::z) = y::(atend x z);
val atend = fn:'a -> 'a list ->
          -> 'a list

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

```
fun split x y = split1 1 x nil y;
val split = fn:nat -> 'a list
               -> 'a list list
fun app nil    z = z |
    app (x::y) z = x::(app y z);
val app = fn:'a list -> 'a list
             -> 'a list
fun map x nil    = nil |
    map x (y::z) = (x y)::(map x z);
val map = fn:('a -> 'b) -> 'a list
             -> 'b list
fun reduce x nil    = nil |
    reduce x (y::z) =
                 (x y (reduce x z));
val reduce = fn:('a -> 'b list ->
              'b list) -> 'a list ->
              'b list
```

Fig. 4. Example list processing functions

are among those given in appendix A. Using these definitions we can specify an equivalence between a single and a distributed application of the *map* function by a conjecture of the form:

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

This conjecture was provided by an independent research group working on the development of parallel systems from functional prototypes (Michaelson & Scaife, 1995). Their development process involves transforming functional prototypes so as to make sites of potential parallelism explicit. Conjecture (23) is such a transformation. Proving the correctness of transformations is currently undertaken by hand and represents a time consuming hurdle to the research project. Having failed to prove conjecture (23) by hand it was passed to us as a challenge theorem. Our techniques were successful in automatically finding a proof of (23).

In order to prove (23) we must first unfold the definition of *split*. An application of rewrite rule (26) gives rise to a refined goal of the form:

$$\begin{aligned}&\forall t : list(A).\forall f : A \to B.\forall n : \mathbb{N}.\\ &\quad map(f, t) = reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(f, x), split_1(1, n, nil, t)))\end{aligned} \tag{24}$$

A proof of (24) requires induction. However, (24) must first be generalized in order for an inductive proof attempt to succeed. An accumulator generalization is required. The generalized conjecture takes the form:

$$\begin{aligned}&\forall t : list(A).\forall f : A \to B.\forall n : \mathbb{N}.\forall l_1 : \mathbb{N}.\forall l_2 : list(A).\\ &\quad map(f, app(l_2, t)) =\\ &\qquad reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(f, x), split_1(l_1, n, l_2, t)))\end{aligned} \tag{25}$$

Note the two new universally quantified variables l_1 and l_2. We focus upon the role our extended critic plays in automating the discovery of (25), the required generalization, and its proof. The wave-rules required for this proof are given in appendix B. With the exception of wave-rules (31) and (32) all the wave-rules are derived from definitions.

7.2.1 First proof attempt:

An inductive proof of (24) requires induction on the structure of the list t. The base case goal is trivial. We focus here on the step case goal which gives rise to an induction hypothesis of the form:

$$\begin{aligned}&\forall f' : A \to B.\forall n' : \mathbb{N}.\\ &\quad map(f', t) = reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(f', x), split_1(1, n', nil, t)))\end{aligned}$$

and an induction conclusion of the form:

$$\begin{aligned}&map(\lfloor f \rfloor, \boxed{h :: \underline{t}}^{\uparrow}) =\\ &\quad reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(\lfloor f \rfloor, x), \underbrace{split_1(1, \lfloor n \rfloor, nil, \boxed{h :: \underline{t}}^{\uparrow})}_{\textbf{blocked}}))\end{aligned}$$

Wave-rule (29) is applicable, however, wave-rules (28) and (27) are not because of the blockage terms 1 and *nil* which occur in the first and third argument positions of $split_1$. Triggered by these blockage terms the extended generalization critic generates a schematic hypothesis of the form:

$$\begin{aligned}&\forall f' : A \to B.\forall n' : \mathbb{N}.\forall l_1' : \mathbb{N}.\forall l_2' : list(A)\\ &\quad map(f', \mathcal{M}_3(t, l_1', l_2')) =\\ &\qquad reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(f', x), split_1(\mathcal{M}_1(l_1'), n', \mathcal{M}_2(l_2'), t)))\end{aligned}$$

while the schematic conclusion takes the form:

$$\begin{array}{l} map(\lfloor f \rfloor, \mathcal{M}_3(\boxed{h :: \underline{t}}^{\uparrow}_{2}, \lfloor l_1 \rfloor, \lfloor l_2 \rfloor)) = \\ \qquad reduce(\lambda x.\lambda y.app(x,y), \\ \qquad\qquad map(\lambda x.map(\lfloor f \rfloor, x), split_1(\mathcal{M}_1(\lfloor l_1 \rfloor), \lfloor n \rfloor, \mathcal{M}_2(\lfloor l_2 \rfloor), \boxed{h :: \underline{t}}^{\uparrow}_{1}))) \end{array}$$

Note that the blockage terms 1 and *nil* have been replaced by primary sink terms $\mathcal{M}_1(\lfloor l_1 \rfloor)$ and $\mathcal{M}_2(\lfloor l_2 \rfloor)$ respectively. Note also that the wave-front on the left-hand-side of the goal equation is classified as secondary and consequently it is associated with a secondary sink term which contains occurrences of l_1 and l_2.

7.2.2 Second proof attempt:

The ripple method is now applied to the schematic goal. Priority is given to the rippling of primary wave-fronts so there is no choice as to which wave-rules should be initially applied. The introduction of sink terms $\mathcal{M}_1(\lfloor l_1 \rfloor)$ and $\mathcal{M}_2(\lfloor l_2 \rfloor)$ enable wave-rules (27) and (28) to be applied. Jointly they motivate a case split on $\mathcal{M}_1(l_1)$ and n.

Case: $\mathcal{M}_1(l_1) > n$:

Using wave-rule (27) the right-hand-side of the conclusion ripples to give:

$$\begin{array}{l} \ldots = reduce(\lambda x.\lambda y.app(x,y), \\ \qquad map(\lambda x.map(\lfloor f \rfloor, x), \boxed{l_2 :: \underline{split_1(\lfloor 2 \rfloor, \lfloor n \rfloor, \lfloor h :: nil \rfloor, t)}}^{\uparrow}_{1}) \end{array}$$

Note that the constraints of the annotated unification process (see figure 3, step 3) instantiate $\mathcal{M}_1$ and $\mathcal{M}_2$ to be $\lambda x.x$. By wave-rule (29) the conclusion ripples further to give:

$$\begin{array}{l} \ldots = reduce(\lambda x.\lambda y.app(x,y), \\ \qquad \boxed{map(f, l_2) :: \underline{map(\ldots, split_1(\lfloor 2 \rfloor, \lfloor n \rfloor, \lfloor h :: nil \rfloor, t))}}^{\uparrow}_{1} \end{array}$$

Note that the wave-front has been β-reduced. A further outward ripple using wave-rule (30) gives:

$$\ldots = \boxed{app(map(f, l_2), \underline{reduce(\ldots, map(\ldots, split_1(\lfloor 2 \rfloor, \lfloor n \rfloor, \lfloor h :: nil \rfloor, t)))})}^{\uparrow}_{1}$$

The left-hand-side of the conclusion contains a secondary sink term so rippling involves more search. As mentioned in §6 an iterative deepening search strategy is employed. Using annotated unification wave-rule (31) applies giving rise to:

$$\boxed{app(map(f, l_2), \underline{map(\lfloor f \rfloor, app(\lfloor h :: nil \rfloor, t))})}^{\uparrow}_{2} = \ldots$$

The application of (31) instantiates $\mathcal{M}_3$ to be $\lambda x.\lambda y.\lambda z.app(z, x)$. Finally, by wave-rule (33) the rippling of the conclusion is complete:

$$map(\lfloor f \rfloor, app(\lfloor h :: nil \rfloor, t)) =$$
$$reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(\lfloor f \rfloor, x), split_1(\lfloor 2 \rfloor, \lfloor n \rfloor, \lfloor h :: nil \rfloor, t)))$$

The induction hypothesis can be applied by instantiating l_1' to be 2 and l_2' to be $h :: nil$. The instantiations for $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ are propagated through the remaining branch of the case split.

Case: $l_1 \leq n$:

Using wave-rule (28) a sideways ripple can be applied to the right-hand-side of the conclusion:

$$\ldots = reduce(\lambda x.\lambda y.app(x, y),$$
$$map(\lambda x.map(\lfloor f \rfloor, x), split_1(\lfloor l_1 + 1 \rfloor, \lfloor n \rfloor, \lfloor atend(h, l_2) \rfloor, t)))$$

The left-hand-side of the conclusion is of the form:

$$map(\lfloor f \rfloor, app(\lfloor l_2 \rfloor, \boxed{h :: \underline{t}}^{\uparrow}_{1})) = \ldots$$

to which wave-rule (32) applies giving:

$$map(\lfloor f \rfloor, app(\lfloor atend(h, l_2) \rfloor, t)) = \ldots$$

The rippling in this branch of the case split is complete:

$$map(\lfloor f \rfloor, app(\lfloor atend(h, l_2) \rfloor, t)) =$$
$$reduce(\lambda x.\lambda y.app(x, y), map(\lambda x.map(\lfloor f \rfloor, x),$$
$$split_1(\lfloor l_1 + 1 \rfloor, \lfloor n \rfloor, \lfloor atend(h, l_2) \rfloor, t)))$$

The induction hypothesis can be applied by instantiating l_1' to be $l_1 + 1$ and l_2' to be $atend(h, l_2)$.

To summarize, the ripple method in conjunction with the extended critic have automatically generated (25), the required generalization of (24). A proof of (25) can be constructed by CLAM completely automatically.

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". That is, 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[§] of examples increases. In addition, we use sink annotations explicitly in selecting potential projections for higher-order meta-variables.

Jane'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 strategy of annotated unification is presented in (Hutter & Kohlhase, 1997) where essentially the structure preservation constraints of rippling are embedded within the actual unification algorithm.

9 Future Work

Our results for the extended critic have been promising. We believe that our technique is not restricted to reasoning about functional programs. This is discussed below where we outline the key areas where we are developing this work.

There exists a strong connection between loop invariants and inductive conjectures. This is reflected at the level of proof where the invariant plays a role similar to that of an induction hypothesis within a step case proof. As a result, the ripple technique can be used to guide the search for a proof when verifying loop invariants. Our work on proof critics is being transferred across to this new domain. In particular, we have begun to investigate the automatic discovery of loop invariants using the critic mechanism (Ireland & Stark, 1997). Discovering a loop invariant is typically seen as the major eureka step in the process of verifying an imperative program. A common strategy for discovering invariants is to start with a desired post-condition from which the invariant is derived by a process of weakening. The notion of a *tail invariant* (Kaldewaij, 1990) represents one such way of deriving an invariant. Using rippling to guide the verification of tail invariants gives rise to the same pattern of sideways rippling which occurs within inductive proof where sinks are exploited. Initial experiments have demonstrated that our critic for accumulator generalization can also play a role in the discovery of tail invariants.

The critic mechanism was motivated by a desire to build an automatic theorem prover which was more robust than conventional provers. The high-level representation provided by a proof plan enabled us to achieve this goal. We believe, however, that the critic mechanism also provides a basis for developing effective user interaction. As highlighted in §5, the critic's mechanism may generate a partial generalization. An interactive version of the critic mechanism has been implemented (Ireland *et al.*, 1997) which invites a user to complete the instantiation of such partial generalizations.

We also believe that our technique is applicable in the context of hardware verifi-

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

cation. For instance, we believe that it subsumes the procedure described in (Pierre, 1995) for generalizing hardware specifications.

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 & Bundy, 1996). These extensions have significantly 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 and GR/L11724, as well as ARC grant 438. We would like to thank Greg Michaelson for providing the challenge example 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 McCann, 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¶

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

$$
\begin{array}{rcl}
prod(nil) & \Rightarrow & 1 \\
prod(X :: Y) & \Rightarrow & prod(Y) * X \\
tsum(nil, Z) & \Rightarrow & Z \\
tsum(X :: Y, Z) & \Rightarrow & tsum(Y, Z + X) \\
tprod(nil, Z) & \Rightarrow & Z \\
tprod(X :: Y, Z) & \Rightarrow & tprod(Y, Z * X) \\
sp(nil, Y, Z) & \Rightarrow & \langle Y, Z \rangle \\
sp(W :: X, Y, Z) & \Rightarrow & sp(X, W + Y, W * Z) \\
sp_2(nil, Y, Z) & \Rightarrow & \langle Y, Z \rangle \\
sp_2(W :: X, Y, Z) & \Rightarrow & sp_2(X, Y + W, Z * W) \\
evenel(nil) & \Rightarrow & nil \\
odd(X) \to evenel(X :: Y) & \Rightarrow & evenel(Y) \\
even(X) \to evenel(X :: Y) & \Rightarrow & X :: evenel(Y) \\
oddel(nil) & \Rightarrow & nil \\
odd(X) \to oddel(X :: Y) & \Rightarrow & X :: oddel(Y) \\
even(X) \to oddel(X :: Y) & \Rightarrow & oddel(Y)
\end{array}
$$

$$
\begin{array}{rcl}
perm(nil, nil) & \Rightarrow & true \\
perm(nil, X :: Y) & \Rightarrow & false \\
perm(X :: Y, Z) & \Rightarrow & (perm(Y, del(X, Z)) \ \wedge \ mem(X, Z)) \\
partition(nil, Y, Z) & \Rightarrow & app(Y, Z) \\
even(W) \to partition(W :: X, Y, Z) & \Rightarrow & partition(X, atend(W, Y), Z) \\
odd(W) \to partition(W :: X, Y, Z) & \Rightarrow & partition(X, Y, atend(W, Z)) \\
split_1(V, W, X, nil) & \Rightarrow & X :: nil \\
V > W \to split_1(V, W, X, Y :: Z) & \Rightarrow & X :: split_1(2, W, Y :: nil, Z) \\
V \leq W \to split_1(V, W, X, Y :: Z) & \Rightarrow & split_1(V + 1, W, atend(Y, X), Z) \\
split(X, Y) & \Rightarrow & split_1(1, X, nil, Y)
\end{array} \tag{26}
$$

Appendix B: selection of example wave-rules

$$V > W \to split_1(\lfloor V \rfloor, W, \lfloor X \rfloor, \boxed{Y :: \underline{Z}}^{\uparrow}_{N}) \Rightarrow \boxed{X :: \underline{split_1(\lfloor 2 \rfloor, W, \lfloor Y :: nil \rfloor, Z)}}^{\uparrow}_{N} \tag{27}$$

$$V \leq W \to split_1(V, W, X, \boxed{Y :: \underline{Z}}^{\uparrow}_{N}) \Rightarrow split_1(\boxed{\underline{V} + 1}^{\downarrow}_{N}, W, \boxed{atend(Y, \underline{X})}^{\downarrow}_{N}, Z) \tag{28}$$

$$map(X, \boxed{Y :: \underline{Z}}^{\uparrow}_{N}) \Rightarrow \boxed{X(Y) :: \underline{map(X, Z)}}^{\uparrow}_{N} \tag{29}$$

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

$$reduce(X, \boxed{Y :: \underline{Z}}^{\uparrow}_{N}) \Rightarrow \boxed{X(Y, \underline{reduce(X,Z)})}^{\uparrow}_{N} \tag{30}$$

$$map(W, app(\lfloor X \rfloor, \boxed{Y :: \underline{Z}}^{\uparrow}_{N})) \Rightarrow \boxed{app(map(W,X), \underline{map(W, app(\lfloor Y :: nil \rfloor, Z))})}^{\uparrow}_{N} \tag{31}$$

$$app(X, \boxed{Y :: \underline{Z}}^{\uparrow}_{N}) \Rightarrow app(\boxed{atend(Y, \underline{X})}^{\downarrow}_{N}, Z) \tag{32}$$

$$\boxed{app(X, \underline{Y})}^{\uparrow}_{M} = \boxed{app(X, \underline{Z})}^{\uparrow}_{N} \Rightarrow Y = Z \tag{33}$$

Appendix C: experimental results

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 I: conjectures

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 CLAM 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 I.

Table II: performance of the extended generalization critic

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 III: lemmata used to motivate generalizations

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.

Table IV: generalized conjectures

References

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

Basin, David, & Walsh, Toby. (1994). *A calculus for and termination of rippling*. Tech. rept. MPI. To appear in special issue of the Journal of Automated Reasoning.

Bird, R., & Wadler, P. (1988). *Introduction to functional programming*. Prentice Hall.

Boyer, R. S., & Moore, J S. (1979). *A computational logic*. Academic Press. ACM monograph series.

Bundy, Alan. (1988). The use of explicit plans to guide inductive proofs. *Pages 111–120 of:* Lusk, R., & Overbeek, R. (eds), *9th conference on automated deduction*. Springer-Verlag. Longer version available from Edinburgh as DAI Research Paper No. 349.

Bundy, Alan, van Harmelen, F., Horn, C., & Smaill, A. (1990). The Oyster-Clam system. *Pages 647–648 of:* Stickel, M. E. (ed), *10th international conference on automated deduction*. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

Bundy, Alan, Stevens, A., van Harmelen, F., Ireland, A., & Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial intelligence*, **62**, 185–253. Also available from Edinburgh as DAI Research Paper No. 567.

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

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

Hesketh, J., Bundy, Alan, & Smaill, A. 1992 (June). Using middle-out reasoning to control the synthesis of tail-recursive programs. *Pages 310–324 of:* Kapur, Deepak (ed), *11th conference on automated deduction*. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

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

Hutter, D., & Kohlhase, M. (1997). A colored version of the λ-Calculus. *Pages 291–305 of:* McCune, W. (ed), *14th conference on automated deduction*. Springer-Verlag. Also available as SEKI-Report SR-95-05.

Ireland, A. (1992). The Use of Planning Critics in Mechanizing Inductive Proofs. *Pages 178–189 of:* Voronkov, A. (ed), *International conference on logic programming and automated reasoning – lpar 92, st. petersburg*. Lecture Notes in Artificial Intelligence No. 624. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 592.

Ireland, A., & Bundy, A. (1996). Productive Use of Failure in Inductive Proof. *Journal of automated reasoning*, **16**(1–2), 79–111. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

Ireland, A., & Stark, J. (1997). On the Automatic Discovery of Loop Invariants. *Proceedings of the fourth nasa langley formal methods workshop – nasa conference publication 3356*. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

Ireland, A., Jackson, M., & Reid, G. (1997). A Collaborative Approach to Theorem Proving. *Proceedings of the first international workshop on proof transformation and presentation (ptp-97) — Schloss Dagstuhl Germany*.

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

Kreisel, G. (1965). Mathematical logic. *Pages 95–195 of:* Saaty, T. L. (ed), *Lectures on modern mathematics*, vol. III. John Wiley and Sons.

Michaelson, G., & Scaife, N. (1995). Prototyping a parallel vision system in Standard ML. *Journal of functional programming*, **5**, 345–382.

Miller, D., & Nadathur, G. (1988). An overview of λProlog. Bowen, K. & Kowalski, R. (ed), *Proceedings of the fifth international logic programming conference/ fifth symposium on logic programming.* MIT Press.

Pierre, L. (1995). An automatic generalization method for the inductive proof of replicated and parallel architectures. Kropf, R. Kumar & T. (ed), *Theorem provers in circuit design.* LNCS 901, Springer-Verlag.

Turner, R. (1991). *Constructive foundations for functional languages.* McGraw-Hill.