Computación y Sistemas Vol. 5 No.1 pp. 25 - 37
© 2001, CIC - IPN. ISSN 1405-5546 Impreso en México

On the Correction of Faulty Formulae

Sobre el Uso de Planificación de Demostraciones en la Corrección de Fórmulas Imperfectas

Raúl Monroy[1] and **Alan Bundy**[2]
[1]Departamento de Ciencias Computacionales, ITESM-CEM
Carretera del Lago de Guadalupe KM. 3.5
Atizapán de Zaragoza, Estado de México, C.P. 52926
[2]Division of Informatics, University of Edinburgh
80 South Bridge , EH1 1HN, Scotland, UK
e-mail: raulm@campus.cem.itesm.mx, A.Bundy@ed.ac.uk

Article received on February 02, 2000; accepted on June 22, 2001

Abstract

We present an abduction mechanism capable of correcting faulty formulae. A formula, G, is said to be faulty *if it is not derivable from a theory, Γ, written $\Gamma \nvdash G$, but we intended it to be. Given a theory, Γ, and a faulty formula, G, the mechanism aims to build a* corrective condition, *P, that transforms G into a theorem, $\Gamma \vdash P \rightarrow G$. The method imposes restrictions upon the quality of a corrective condition. The mechanism is fully automatic. It is given as a collection of heuristics. Each heuristic control captures the restricted way in which the search for a proof of a faulty formula can fail, and provides knowledge to recover from such a failure.*

Keywords: Abduction, Theorem Proving, Program Synthesis/Transformation, Proof Planning.

Resumen

Presentamos un mecanismo abductivo capaz de corregir fórmulas imperfectas. Una fórmula, G, es imperfecta si, contrario a lo esperado, no puede deducirse de una teoría, Γ, en símbolos $\Gamma \nvdash G$. Dada una teoría , Γ, y una fórmula imperfecta, G, el mecanismo tiene como objetivo construir una condición correctiva, *P, tal que transforme G en un teorema, $\Gamma \vdash P \rightarrow G$. El mecanismo impone restricciones en la calidad de una condición correctiva. El mecanismo es completamente automático. Está definido como un conjunto de heurísticas. Cada una captura la forma restringida en como la búsqueda de un plan de demostración puede fallar y provee conocimiento para recuperar dicha falla.*

Palabras clave: Abducción, Demostración de Teoremas, Síntesis y Transformación de Programas, Planificación de Demostraciones.

1 Introduction

The paper is concerned with understanding and correcting mal-formulations, a fundamental process of theory refinement. We present a mechanism capable of correcting a faulty formula. A *faulty formula* is a contingent formula that we expected or intended to be a theorem. A *contingent formula* is a first-order formula, G, such that neither G nor $\neg G$ are derivable from some axiomatic theory, Γ, written $\Gamma \nvdash G$ and $\Gamma \nvdash \neg G$.

Given a theory, Γ, and a faulty formula, G, the mechanism aims to identify or build—if necessary—, a *corrective condition*, P, with the following properties:

correctness: P, together with the working theory, turns G into a theorem, $\Gamma \vdash P \rightarrow G$;

consistency: P is not a contradiction with respect to the working theory, $\Gamma \cup \{P\} \nvdash$ false; and

non-triviality: P is a nontrivial explanation of G, $\nvdash P \rightarrow G$.

Building a corrective condition amounts to providing an algorithm for computing the condition. The mechanism renders an algorithm as a collection of equational cases, each of which might be conditional. The algorithm output might be recursive and, in that case, it is guaranteed to be terminating. An algorithm is said to be *terminating* if it does not fall into infinite computation.

The process of correcting a faulty formula via a corrective condition is based on abduction (Peirce, 1959). Abduction is a form of logical inference that allows us to find causes that explain an observed phenomenon. Abduction is closely related to deduction; it is performed when no further deduction is possible, exploring the associated partial proof tree. These trees are usually huge and might well be infinite. Special care thus needs to be taken in order to tackle the combinatorial explosion phenomenon. Fortunately, proof planning provides, at least partially, an answer to this problem.

Proof planning (Bundy, 1988) is a meta-level reasoning technique. Ideally, it splits the problem of proving a theorem into two stages, one in which an appropriate plan is assembled by the planning engine, and another in which the plan is executed by a theorem prover. In practice, object level steps are interleaved with the meta-level ones. Yet, proof planning is cheaper than searching for a proof in the underlying theory. This is both because each plan step covers a lot of proof steps, and because a proof plan contains many heuristics that dramatically restrict the search planning space. We have used proof planning to implement an abductive mechanism for the correction of faulty formulae.

The abductive mechanism is given as a collection of heuristics. Each heuristic captures the restricted way in which the search for a proof plan can fail and provides a basis for exploiting any failure or partial success in plan formation. This general knowledge is then used to turn a faulty formula into a theorem. The abductive mechanism is an extension of that introduced by Monroy, Bundy, and Ireland (1994).

1.1 Paper Overview

The rest of the paper is organised as follows: First, we describe the abduction rule of inference underlying the detection and isolation of faults, §2. Then, we argue that the meta-level reasoning embedded in proof planning makes it possible to exploit any failure or partial success in the search for a proof, §3. Next, we show how to implement an abduction mechanism within proof planning, via an exception handling mechanism, §4. Also, we show how to exploit proof planning failure so as to guide the correction process, hence pruning the explanation search space. After summarising experimental results, we compare the abduction mechanism with rival techniques, §5, and draw attention to the lessons that we have learned through our investigation, §6.

2 Abduction

Abduction (Peirce, 1959) is a form of non-monotonic reasoning used to discover explanations of phenomena whose cause is not clear. Peirce pointed out that abduction is the only kind of reasoning that supplies new ideas. So, abduction is synthetic.

The simplest form of abduction is as follows:

From a rule, $A \to B$, and a result, B
Infer A as a plausible explanation of B

In a logical setting (Levesque, 1989), abduction and deduction are closely related. Abduction is commonly applied when no further deduction is possible, exploring the open leaf nodes—called *dead ends*—of the associated partial deduction tree.

More generally, Kakas, Kowalski, and Toni (1998) characterise the abductive task as follows: Given a set of sentences, Γ, and a set of observations, G, find a set of sentences, C, such that:

- $\Gamma \cup C \vdash G$;
- $\Gamma \cup C$ is consistent; and
- $C \subseteq \Delta$.

where Δ is a set of abducible sentences. A sentence is said to be *abducible* if it explains an observation and if it itself cannot be explained in terms of some other cause.

The major problem with computing an abductive hypothesis is the combinatorial explosion phenomenon. A failed proof attempt quickly yields a huge, possibly infinite, deduction tree. This, in turn, gives rise to exponentially many explanations (De Kleer, 1986), most of which are not relevant. Choosing a good candidate explanation automatically is hence an extremely difficult task. The problem is further magnified if minimal sized explanations are required. Computing minimal explanations has been proved to be NP-hard, even in acyclic Horn theories (Selman & Levesque, 1989). Thus, care needs to be taken in order to make a productive use of failure.

We suggest that a proof planning approach can structure the task of correcting a faulty formula in such a way as to allow significant automation, while dramatically restricting the search space.

3 Proof Planning

Proof planning (Bundy, 1988) is a meta-level reasoning technique specially developed as a search control engine to automate theorem proving. A proof plan captures general knowledge about the commonality between the members of a proof family, and it is used to guide the search for more proofs in that family.

Proof planning works in the context of a tactical style of reasoning (Gordon, Milner, & Wadsworth, 1979). It applies AI planning techniques to build large complex tactics from simpler ones. Proof planning is a two step approach. In the first step, a tactic is assembled by the planning engine; in the second one, the tactic is executed by a theorem prover. In the normal case of success, the second step yields an actual proof. Sometimes, object-level steps need to be interleaved whilst assembling a tactic: Some of the later planning steps cannot be made without the detail provided by the earlier object-level ones.

Methods are the building-blocks of proof planning. A *method* is a partial specification of a tactic. It is a 5-tuple, consisting of the tactic name, the goal, the preconditions, the effects and the output formulae. Proof planning checks the preconditions against the goal to predict whether the tactic associated with the method is applicable and, if so, uses the effects to anticipate its application, which results in the output formulae. *Proof planning* is the recursive process that reasons about and composes tactics. Upon success, it returns a compound tactic, called a *proof plan*, tailored to prove the conjecture at hand.

Proof planning is cheaper than searching for a proof in the underlying object theory. This is for two reasons: First, each plan step covers a lot of object-level theorem proving steps: proof planning emphasises proof structure, filling in direct but tedious, onerous reasoning. Second, the method preconditions dramatically restrict the search space: backtracking hardly occurs.

Proof planning has been implemented within the *Clam* proof planning system (Bundy, van Harmelen, Horn, & Smaill, 1990) and successfully applied in formal methods, including synthesis (Armando, Smaill, & Green, 1999) and verification (Monroy, Bundy, & Green, 2000; Cantu, Bundy, Smaill, & Basin, 1996).

3.1 Proof Planning with Critics

The incorporation of an exception handler to proof planning is called *proof planning with critics* (Ireland, 1992). The exception handler is invoked if the goal at hand is blocked and if the applicability preconditions of a method partially hold. A goal is said to be *blocked* if no method applies to it. Upon request, the exception handler will try the proof critics associated to the partially applicable method, if any, one at a time, in the order of appearance.

A *proof critic* contains a high-level specification of a failure pattern or partial attainment, as well as providing the corrective action. It is a 4-tuple, containing the critic name, the goal, the preconditions and the corrective action. The application of a proof critic to a given goal consists of checking the preconditions against the goal and then executing the corrective action.

Unlike the application of a method, the application of a proof critic does not refine the proof plan under construction. It just enables further proof planning by rendering side effects. Critics effects may include a modification to the input formulae, e.g., generalisation or fault correction, or to the working theory, e.g., lemma discovery. Proof planning with critics has been successfully used for lemma discovery (Ireland & Bundy, 1996), formula generalisation (Ireland & Bundy, 1996) and faulty formula correction (Monroy et al., 1994).

3.2 Inductive Proof Planning

The application of proof planning to theorem proving by mathematical induction is called *inductive proof planning* (Bundy, 1988).[1] It is characterised by the following methods: The *induction* method selects the most promising induction scheme via a *rippling analysis* (Bundy, van Harmelen, Hesketh, Smaill, & Stevens, 1989). The base case(s) of proofs by induction are dealt with by the *elementary* and *sym_eval* methods. Elementary is a tautology checker for propositional logic and has limited knowledge of intuitionistic propositional sequents, type structures and properties of equality. Sym_eval simplifies the goal by means of exhaustive symbolic evaluation and other routine reasoning.

Similarly, the step case(s) of proofs by induction are dealt with by the *wave* and *fertilise* methods. Wave applies *rippling* (Bundy, Stevens, van Harmelen, Ireland, & Smaill, 1993), a heuristic that guides transformations in the induction conclusion to enable the use of an induction hypothesis. This use of an induction hypothesis is called *fertilisation* by Boyer and Moore (1979), hence the name of the method. The inductive proof plan, summarised in Table 1, is both the most studied and the most successful (Bundy et al., 1989). Given that rippling is the key for success within the inductive proof plan, we shall discuss it further.

3.2.1 Rippling

The key idea behind rippling lies in the observation that one or more induction hypotheses are (simultaneously) embedded in the (initial) induction conclusion. The differences between the conclusion and the hypotheses consist of extra terms, e.g., a constructor function wrapping an induction variable. By marking them explicitly, rippling can attempt to place these differences at positions where they no longer preclude the conclusion and hypothesis from matching. Rippling is therefore an annotated term-rewriting system. It applies a special kind of rewrite rules, called a *wave-rule*, which manipulates the differences between two terms while keeping their common structure intact.

[1] Mathematical induction should not be confused with philosophical induction, a form of synthetic reasoning that infers a rule from a case and a result (Peirce, 1959). Both abduction and philosophical induction are weak kinds of inference. Their relation and integration are still under development (Flach & Kakas, 2000). Henceforth, we shall use the term induction as a shorthand of mathematical induction.

Method	Description
elementary	is a tautology checker for propositional logic and has limited knowledge about intuitionistic propositional sequents, type structures and properties of equality
equal	looks for expressions of the form *var*=*term* amongst the hypotheses, and replaces every occurrence of *term* in the current goal for *var*
sym_eval	puts the input expression into canonical form by means of exhaustive rewrite rule application
generalise	replaces a subterm, appearing in both halves of an equality, an implication or an inequality, by a simple variable
normalise	simplifies sequent formulae using rules for manipulating sequents, such as implication introduction $\dfrac{A, \Gamma \vdash B}{\Gamma \vdash A \to B}$
wave	applies *wave-rules*, as dictated by *rippling*, a heuristic that guides transformations in the induction conclusion to enable the use of an induction hypothesis
casesplit	divides a proof into cases, considering the partition defined by a set of complementary rewrite rules
fertilise	simplifies goals using an induction hypothesis
induction	applies induction, this involves the selection of a suitable induction scheme and one or more induction variables

Table 1: The Standard Method Data-Base

Wave annotations are introduced by induction,[2] as illustrated by the rule below:[3]

$$\frac{H \vdash G(0) \qquad x : \text{nat}, G(x) \vdash G(\boxed{\text{s}(\underline{x})})}{H \vdash \forall x : \text{nat}.\ G(x)} \tag{1}$$

Wave-terms, e.g., $\boxed{\text{s}(\underline{x})}$, are composed of a wave-front, and one or more wave-holes. *Wave-fronts*, e.g., $\boxed{\text{s}(\ldots)}$, are expressions that appear in the induction conclusion but not in the induction hypothesis. Inversely, *wave-holes*, e.g., $\underline{x}$, are expressions that appear in wave-terms and also in the induction hypothesis. By deleting the symbols that are within wave-fronts but not within wave-holes and, then, the annotations completely, we get the *skeleton*. The skeleton is a copy of the induction hypothesis and, so, should be kept unchanged.

A *wave-rule* is an annotated rewrite rule, $L :\Rightarrow R$, such that it is skeleton preserving and measure decreasing, under a suitable ordering, $\succ$, on annotated terms (Basin & Walsh, 1996). This is characterised by $skel(L) = skel(R)$ and $m(L) \succ m(R)$. The well-founded measure is the inverted list of the depths, with respect to the skeleton, of the wave-fronts, at each level, ordered lexicographically. The well-founded measure can be shown to be stable and monotonic under a non-standard term replacement. Hence, rippling can be shown to be terminating.[4]

With this, we complete our revision of both abduction and proof planning. We are ready to introduce the abduction mechanism, the subject of the next section.

4 Correcting Faulty Formulae

Like Monroy et al. (1994), our abduction mechanism is built within inductive proof planning with critics. It is given by a collection of proof critics, each of which captures heuristics to detect, isolate and correct some sorts of mal-formulations.

Roughly, the mechanism works as follows: Given conjecture G, then:

1. Use inductive proof planning to search for a proof of G. If G is a theorem, this process will often terminate yielding a proof plan. Otherwise, it will terminate failing and pointing at an unprovable sub-goal. Call this sub-goal G' and call the method at which proof plan formation stopped M.

2. Use M's proof critics to perform a syntactic analysis on G'. If the failure pattern is captured, this

[2]Wave annotations can also be introduced by applying *difference unification* (Basin & Walsh, 1993) to the hypotheses and the conclusion. Difference unification extends unification so that differential structures between the terms to be unified can also be hidden, while computing a substitution.

[3]s is the successor function defined over natural numbers.

[4]For further details, readers are referred to the Mathematical Reasoning Group home page, `http://dream.dai.ed.ac.uk`.

analysis often identifies or builds a condition, P, which will (hopefully) remove the bug from G.

3. Use this procedure recursively, considering $P \rightarrow G$, until a proof plan is found.

Corrective conditions take the form of the following recursive scheme:

$$\begin{aligned} P(B_0) &= P_{B_0} \\ P(B_1) &= P_{B_1} \\ &\vdots \\ P_{S_0} \rightarrow P(C_0(N)) &= P_0(N) \\ P_{S_1} \rightarrow P(C_1(N)) &= P_1(N) \\ &\vdots \end{aligned}$$

where $B_0, B_1, \ldots$ and $C_0, C_1, \ldots$ denote base case elements and constructor functions. These terms are fixed by the inductive rule used to attempt to prove G.

Building P amounts to fixing P_{B_i}, P_{S_i} and P_i, for $i \geq 0$. Predicate construction takes place upon proof planning failure. P_{B_i} ($i \geq 0$) is fixed to false if the base case i could not be established, or fixed to true otherwise. It might be equally fixed to other, possibly recursive, proposition, if induction was further applied. So, nested recursion is yielded.

P_i ($i \geq 0$) is fixed to P if fertilisation was used to complete the proof of step case i. Otherwise it is set to true and, so, P might not be recursive. P_{S_i} ($i \geq 0$) is fixed to true if neither nested induction nor extra assumptions were required to complete the proof. Extra assumptions are often introduced by case analysis. We insist that neither P_i nor P_{S_i} are ever set to false. If no critic captures any one step case failure, proof planning will fail.

P is not necessarily unary. Extra arguments are added to its structure if either case analysis or nested induction is used. We will come back to this issue later on in the text. First, though, we will look into fault correction using non-recursive conditions, which are interesting on their own. As we shall see, they mend specification formulae that overlook boundary case conditions, a common mistake in software development.

4.1 Non-Recursive Conditions

Specifications which overlook boundary case conditions develop in either of two sorts of blocked goals: contradictory or contingent.

4.1.1 Contradictory Blocked Goals

A goal is *contradictory blocked* if it is blocked, ground, and invalid with respect to the working theory. Contradictory blocked goals may originate when we try to solve either a base case of an inductive proof, or a branch of a proof by case analysis. To recover from failure, the abduction mechanism will suggest to introduce the negation of the case as a condition to the original conjecture. If it is a base case and nested induction was used, the mechanism will select the case for the most recent induction. By omitting this case, the contradictory goal will not be experienced again.

To give an impression as to how the mechanism works, consider the following faulty formula:

$$\forall L{:}\alpha \text{ list. } \text{length}(L) \neq 0 \tag{2}$$

Proof planning attempts to address (2) using primitive list induction on L, henceforth abbreviated as $H{::}L$ induction. The base case, $L = \text{nil}$, gives rise to a contradictory blocked goal, namely: $0 \neq 0$. With this failure pattern, the mechanism will postulate $L \neq \text{nil}$ as the required condition, yielding the new specification:

$$\forall L{:}\alpha \text{ list. } L \neq \text{nil} \rightarrow \text{length}(L) \neq 0 \tag{3}$$

This is a theorem. It can be proved by induction or case analysis. If induction is used, however, the step case can be established without fertilisation. This explains why the condition is not recursive. We will have more to say about this in §4.2.1.

To give the reader a flavour of the implementation, Figure 1 provides the definition of the critic following the *Clam* convention.

Now, let us consider the alternate case, where the blocked goal is not contradictory.

4.1.2 Contingently Blocked Goals

A goal is *contingently blocked* if it is blocked, open and contingent with respect to the working theory. A formula is *open* if it is not ground and the status of all the variables that occur in it has not been announced with a quantifier. Recall that a formula is *contingent* if it is a proposition and if the proposition holds in some interpretation. Put in a logical way, a formula is contingent if either the formula or the negation of its closure is a theorem. The *closure* of a formula is as the formula except that there is a universal quantifier attached to each of its free variables. Contingently blocked goals may appear at any proof step. Often they serve as conditions to patch the buggy specifications from which they originated. Here is one such a specification:

$$\forall X, Y{:}\text{nat. } X + Y > X \tag{4}$$

Proof planning would attempt to address (4) using single-step induction on X, henceforth abbreviated as $s(X)$ induction. The base case, $X = 0$, would yield a contingently blocked goal, namely: $Y \neq 0$. With this

```
CRITIC elementary
 Input:          $Plan, Address,$                              ; If current goal, $G'$, is
                 $\text{node}(Plan, Address, \Gamma' \vdash G')$   ; contradictory blocked,
 Precondition:                                                 ; negate the condition
                 $\exists T \in \{t : t \text{ is invalid}\},$      ; for the most recent
                 $G' = T\phi$—$\phi$ is a substitution             ; induction or case and
 Patch:                                                        ; add it as a condition to
                 $\text{failed\_at}(Plan, Address, Var, Case),$   ; the original goal, $G$
                 $\text{node}(Plan, [], \Gamma \vdash \forall \overrightarrow{Vars}.\, G),$
                 $\neg(Case(Var))$
 Output:
                 $\forall \overrightarrow{Vars}.\, P \rightarrow G$

Meanings of the meta-logic terms:
 • node($Plan, Address, Seq$) means that node at position $Address$ of proof plan
   tree $Plan$ contains the sequent $Seq$. N.B. [] is the address of the root node.
 • failed_at($Plan, Address, Var, Case$) $Case$ denotes the case at which failure
   has occurred with respect to the most recent induction or casesplit
   application. $Var$ is the current induction variable or the variable involved in
   the case analysis.
```

Figure 1: Critic: contradictory blocked goals

failure pattern, the mechanism would postulate this new conjecture:

$$\forall X, Y{:}\text{nat}.\, Y \neq 0 \rightarrow X + Y > X$$

This abductive strategy resembles resolution based abduction mechanisms, where the residue is used as a patching condition if it belongs to a set of abducibles. Figure 2 provides the definition of the critic handling this situation.

When contingently blocked goals cannot be used as patching conditions, equation solving and symbolic evaluation are required. Fortunately, these formulae often contain no recursive symbols and, hence, it is easy to write the corresponding procedure.

4.1.3 Contingently Blocked Goals—Case Analysis

It only remains to illustrate abduction in case analysis. We shall work by example. Consider this faulty formula:

$$\forall X{:}\alpha, L{:}\alpha\ \text{list}.\ \text{insert}(X, L) = X :: L \qquad (5)$$

together with the rewrites given by the usual definition of in-order insertion:

$$\begin{aligned} \text{insert}(X, \text{nil}) &\Rightarrow X :: \text{nil} \\ X \leq Y \rightarrow \text{insert}(X, \boxed{Y :: \underline{L}}) &\Rightarrow X :: Y :: L \\ X \not\leq Y \rightarrow \text{insert}(X, \boxed{Y :: \underline{L}}) &\Rightarrow \boxed{Y :: \underline{\text{insert}(X, L)}} \end{aligned}$$

Proof planning would try to deal with (5) using $H{::}L$ induction. It would establish the base case, but get stuck in the step case, after an application of case analysis. The cases are $X \leq H$ and $X \not\leq H$. The second case, $X \not\leq H$, cannot be proved, for $X :: H :: L = H :: X :: L$ is not valid but contingent. Thus, it will be rejected. The mechanism will produce:

$$\begin{aligned} \text{P}_{\text{ins}}(X, \text{nil}) &= \text{true} \\ X \leq H \rightarrow \text{P}_{\text{ins}}(X, H :: T) &= \text{true} \\ X \not\leq H \rightarrow \text{P}_{\text{ins}}(X, H :: T) &= \text{false} \end{aligned}$$

Three aspects deserve observation:

1. Inductive proof search involving mal-formulated boundary case specifications does not necessarily break in base cases.

2. Neither the second defining equation nor the third one are recursive, for the induction hypothesis was not used in the step case.

3. P_{ins} is binary; the extra argument was introduced by the use of case analysis.

Figure 3 formulates this patching strategy as a proof critic.

The failure patterns yielded by blocked goals, both contingent and contradictory, are represented in three

```
CRITIC elementary
  Input:          Plan,Address,                        ; If current goal, $G'$, is
                  node(Plan, Address, $\Gamma' \vdash G'$)      ; an instance of an
  Precondition:                                        ; abducible predicate,
                  contingent($G'$),                      ; then use it as a
                  $\exists T \in \{t : t \text{ is abducible}\}$,          ; patching condition to
                  $G' = T\phi$—$\phi$ is a substitution        ; the original goal
  Patch:
                  node(Plan, [], $\Gamma \vdash \forall \vec{X}. G$),
                  $P = G'$                                 Output:
                  $\forall \vec{X}. P \to G$
```

Figure 2: Critic: contingently blocked goals

Table 2: Some formulae corrected using non-recursive patches

Faulty Conjecture	Patching Condition
$\text{length}(\text{app}(A,B)) > \text{length}(A)$	$B \neq \text{nil}$
$\text{length}(\text{app}(A,B)) > \text{length}(B)$	$A \neq \text{nil}$
$X + Y > X$	$Y \neq 0$
$X + Y > Y$	$X \neq 0$
$\text{double}(X) > \text{half}(X)$	$X \neq 0$
$\text{double}(X) > X$	$X \neq 0$
$X > \text{half}(X)$	$X \neq 0$
$\text{insert}(X, L) = X :: L$	$\text{P}_{\text{ins}}(X, \text{nil}) = \text{true}$ $X \leq H \to \text{P}_{\text{ins}}(X, H :: T) = \text{true}$ $X \not\leq H \to \text{P}_{\text{ins}}(X, H :: T) = \text{false}$

proof critics. These critics are associated to the proof methods elementary and casesplit. Combined, they deal with a number of faulty formulae, some are shown in Table 2.

We are now ready to consider the construction of recursive conditions, the subject of the next section.

4.2 Recursive Conditions

Overlooking boundary case conditions is but one sort of error people make when writing specifications. We shall consider one more: failing to identify properties of the operators involved in the conjecture. Here is one example of such faulty formulae:

$$\forall L{:}\alpha \text{ list}.\ \text{sort}(L) = L \tag{6}$$

Though it looks unusual, conjecture (6) is appealing. It can arise as a result of an over generalisation of the conjecture $\text{sort}(\text{sort}(M)) = \text{sort}(M)$, which states sort is idempotent.[5] In the following, we show that our abduction mechanism tackles it.

We assume the availability of the insert rewrites, as well as the following:

$$\begin{aligned} \text{sort}(\text{nil}) &:\Rightarrow \text{nil} \\ \text{sort}(\boxed{X :: \underline{L}}) &:\Rightarrow \boxed{\text{insert}(X, \underline{\text{sort}(L)})} \end{aligned}$$

To attempt to prove (6), rippling analysis suggests the use of $H{::}L$ induction. The base case is easily solved, so attention will be only given to the step case. The induction hypothesis and the induction conclusion respectively are:[6] $\text{sort}(l) = l \vdash \text{sort}(\boxed{h :: \underline{l}}) = \boxed{h :: \underline{l}}$. As usual, we work on the conclusion. Rippling and then fertilisation yield:

$$\ldots \vdash \text{insert}(h, l) = h :: l$$

This sub-goal is then transformed into:

$$\forall X{:}\alpha, B{:}\alpha \text{ list}.\ \text{insert}(X, B) = X :: B$$

[5] An *over generalisation* is an operation that transforms a theorem into a faulty formula.

[6] l and h are Skolem constants introduced by the induction rule. They respectively replace L and H. Strictly speaking, we should add type information to formulae. However, for the sake of readability we have preferred to omit it.

```
CRITIC casesplit
Input:          $Plan, Address,$                                 ; If current goal, $G'$, is contingent,
                node($Plan, Address, \Gamma' \vdash G'$)         ; then set current equation case to
Precondition:                                                    ; false and build $P$ according to the
                contingent($G'$)                                 ; structure of the partial proof tree
Patch:
                failed_at($Plan, Address, Var, Case_i$),
                freevars($Case_i, \overrightarrow{Vars}$),
                $Case_i \rightarrow P(\overrightarrow{Vars}) =$ false,
                $\forall j \neq i.$ closed($Case_j$) $\rightarrow Case_j \rightarrow P(\overrightarrow{Vars}) =$ true,
                build_condition($Plan, Structure, P$)
                node($Plan, [], \Gamma \vdash \forall \overrightarrow{X}. G$)
Output:
                $\forall \overrightarrow{X}. P \rightarrow G$

Meanings of the meta-logic terms:
  • freevars($Case, \overrightarrow{Vars}$) $\overrightarrow{Vars}$ contains the variables that appear free in $Case$
  • build_condition($Plan, Structure, P$) $Structure$ is the structure associated with the
    partial proof plan, $Plan$, which is used in order to construct the corresponding
    corrective condition
```

Figure 3: Critic: contingently blocked goals in case analysis

which is (5), the example conjecture reviewed in §4.1.3.

So, to correct (6), the abduction mechanism first repairs the new instance of (5), yielding $\mathrm{P_{ins}}$. Next it performs an analysis upon the entire proof search of (6), gathering the following information:

1. The proof of the initial base case is complete. So the patching condition evaluates to true when the input list is empty, c.f. (7).

2. The proof of the initial step case is incomplete. Fertilisation and then nested induction were used, outputting a sub-goal that can be patched via $\mathrm{P_{ins}}$, c.f. (8).

So the patching predicate is recursive, the side condition being $\mathrm{P_{ins}}$. Thus, the mechanism synthesises $\mathrm{P_{st}}$:

$$\begin{aligned} \mathrm{P_{st}}(\mathrm{nil}) &= \mathrm{true} \quad (7)\\ \mathrm{P_{ins}}(H,T) \rightarrow \mathrm{P_{st}}(H :: T) &= \mathrm{P_{st}}(T) \quad (8)\end{aligned}$$

With which, it finally patches (6):

$$\forall L{:}\alpha\ \mathrm{list}.\ \mathrm{P_{st}}(L) \rightarrow \mathrm{sort}(L) = L$$

Observe that $\mathrm{P_{st}}$ is recursive, because the inductive hypothesis was used. Furthermore, observe that $\mathrm{P_{ins}}$ is added as an extra condition in the second equation, because it was suggested to repair the new sub-goal.

By unfolding and symbolic evaluation, we can compose both definitions to yield a simpler one:

$$\begin{aligned} P_{\mathrm{st}}(\mathrm{nil}) &= \mathrm{true}\\ P_{\mathrm{st}}(H :: \mathrm{nil}) &= \mathrm{true}\\ H_1 \leq H_2 \rightarrow P_{\mathrm{st}}(H_1 :: H_2 :: T) &= P_{\mathrm{st}}(H_2 :: T)\end{aligned}$$

which defines the predicate *ordered*.

4.2.1 A Word on Recursion

Conditions are recursive only if fertilisation is applied. In §4.1.1, we emphasised that fertilisation is not required for demonstrating the step case of (3), namely:

$$\begin{aligned} & l \neq \mathrm{nil} \rightarrow \mathrm{length}(l) \neq 0\\ & \qquad \vdash \boxed{h :: l} \neq \mathrm{nil} \rightarrow \mathrm{length}(\boxed{h :: l}) \neq 0\end{aligned}$$

This is because the antecedent and the consequent of the implicational conclusion are both readily provable using arithmetical reasoning. So if no notice of this is taken, we could then carelessly extract P′:

$$\begin{aligned} \mathrm{P}'(\mathrm{nil}) &= \mathrm{false}\\ \mathrm{P}'(H :: L) &= \mathrm{P}'(L)\end{aligned}$$

which is non interesting as it evaluates to false. Thus, care is to be taken in order to gain a profit from proof failure.

If the abduction mechanism yields the proposition false, the conjecture at hand is considered a non-theorem. If the current conjecture is the top goal, then the abduction mechanism suggests to consider the negation of the closure. Otherwise, the abduction mechanism treats it as a contradictory blocked goal and proceeds accordingly.

The strategy shown in this section was implemented as an extension of the casesplit critic. It extends the range of faulty formulae that the mechanism can patch automatically. Some are shown in Table 3.

4.3 Working by Refinement

Working by refinement was designed to refine previous attempts at patching faulty formulae. This technique is invoked any time proof planning is stuck and there is evidence which suggests that old patching conditions are necessary but insufficient to transform the conjecture into a theorem. It combines the strategies seen so far.

To illustrate, consider the following buggy specification (Monroy et al., 1994):

$$\forall X:\text{nat}.\ \text{double}(\text{half}(X)) = X \tag{9}$$

where double and half have their natural interpretation returning twice and half the integer part of their inputs:

$$\begin{array}{rcl}
\text{double}(0) & \Rightarrow & 0\\
\text{double}(\boxed{\text{s}(\underline{N})}) & \Rightarrow & \boxed{\text{s}(\text{s}(\underline{\text{double}(N)}))}\\
\text{half}(0) & \Rightarrow & 0\\
\text{half}(\text{s}(0)) & \Rightarrow & 0\\
\text{half}(\boxed{\text{s}(\text{s}(\underline{N}))}) & \Rightarrow & \boxed{\text{s}(\underline{\text{half}(N)})}
\end{array}$$

Proof planning suggests to attempt to prove (9) using $\text{s}(\text{s}(X))$ induction. The first base case goes through but the second one does not. The abduction mechanism therefore suggests to omit the second case, yielding the following formulation:

$$\forall X:\text{nat}.\ X \neq \text{s}(0) \rightarrow \text{double}(\text{half}(X)) = X$$

With the refined conjecture, the use of $\text{s}(\text{s}(X))$ induction is again suggested. Now both basis cases go through. In the step case, the induction hypothesis is:

$$x \neq \text{s}(0) \rightarrow \text{double}(\text{half}(x)) = x \tag{10}$$

and the annotated induction conclusion is:

$$\vdash \boxed{\text{s}(\text{s}(\underline{x}))} \neq \text{s}(0) \rightarrow \\ \text{double}(\text{half}(\boxed{\text{s}(\text{s}(\underline{x}))})) = \boxed{\text{s}(\text{s}(\underline{x}))}$$

Rippling followed by fertilisation transforms the induction conclusion outputting $\vdash \text{s}(\text{s}(x)) \neq \text{s}(0) \rightarrow x \neq \text{s}(0)$. This goal constitutes a blockage for plan formation. It is not a theorem or a contradiction, but a contingency. So the abduction mechanism is called.

As before, the entire proof plan is inspected to recover from failure. We can then make these observations:

1. The base cases are complete but the step case is not.
2. At the beginning, the step case is of the following schematic form:
$$P(X) \rightarrow G(X) \vdash P(\boxed{C(\underline{X})}) \rightarrow G(\boxed{C(\underline{X})})$$
while the residue is of the form $P(C(X)) \rightarrow P(X)$.
3. The (implicational) hypothesis was used (right to left).

Together, these features suggest that the condition $P(X)$ is necessary but not enough to mend (9). A refined condition must be built so that it is consistent with the current definition of $P(X)$, and prevents the blockage $P(C(X)) \rightarrow P(X)$. Thus, the abduction mechanism will synthesise the following recursive definition:

$$\begin{array}{rcl}
\text{P}_{\text{dh}}(0) & = & \text{true}\\
\text{P}_{\text{dh}}(\text{s}(0)) & = & \text{false}\\
\text{P}_{\text{dh}}(\text{s}(\text{s}(X))) & = & \text{P}_{\text{dh}}(X)
\end{array}$$

The cases associated to P_{dh}, the corrective condition, can be checked to constitute the predicate *even*. The method has therefore corrected (9) suggesting it holds only if N is divisible by two. The proof critic representing the strategy of working by refinement is shown in Figure 4. Table 4 shows some example faulty formulae that were successfully corrected working by refinement.

In total, we tested our mechanism on a set of 35 faulty formulae. It proved to be capable of correcting 93% of them. Most failures had to do with the method used to construct an equational procedure, c.f. build_condition/3. Since it works backwards, from the tip of the partial proof tree to the root of it, this method is complex and hard to formalise. So it is largely ad-hoc (We will have more to say about this issue in §5.)

Here are the failures:

$$\text{odd}(\text{half}(N)) \tag{11}$$
$$\text{ord}(L) \wedge \text{ord}(M) \rightarrow \text{ord}(L <> M) \tag{12}$$
$$L <> (M <> N) = (L <> N) <> M \tag{13}$$

Patching (11) seems to require a mutually recursive condition, a form of recursion that both the design and the development stage of our method missed to identify.

Table 3: Some formulae corrected using recursive patches

Faulty Conjecture	Patch
$\mathrm{sort}(\mathrm{app}(A,B)) = \mathrm{app}(\mathrm{sort}(A),\mathrm{sort}(B))$	$\mathrm{P_{st}}(\mathrm{nil},B) = \mathrm{true}$ $\mathrm{P_s}(H,B) \rightarrow \mathrm{P_{st}}(H :: A,B) = \mathrm{P_{st}}(A,B)$ $\mathrm{P_s}(X,\mathrm{nil}) = \mathrm{true}$ $X \leq H \rightarrow \mathrm{P_s}(X,H :: B) = P_s(X,B)$
$\mathrm{app}(A,B) = \mathrm{app}(B,A)$	$\mathrm{P_{com}}(\mathrm{nil},B) = \mathrm{true}$ $\mathrm{P_s}(H_A,B) \rightarrow \mathrm{P_{com}}(H_A :: A,B) = \mathrm{P_{com}}(A,B)$ $\mathrm{P_s}(H_A,\mathrm{nil}) = \mathrm{true}$ $H_A = H_B \rightarrow \mathrm{P_s}(H_A,H_B :: B) = \mathrm{P_s}(H_A,B)$
$\mathrm{rev}(\mathrm{rev}(\mathrm{app}(A,B))) = \mathrm{app}(B,A)$	$\mathrm{P_{rev}}(\mathrm{nil},B) = \mathrm{true}$ $\mathrm{P_s}(H_A,B) \rightarrow \mathrm{P_{rev}}(H_A :: A,B) = P_{\mathrm{rev}}(A,B)$ $\mathrm{P_s}(H_A,\mathrm{nil}) = \mathrm{true}$ $H_A = H_B \rightarrow \mathrm{P_s}(H_A,H_B :: B) = \mathrm{P_s}(H_A,B)$
$(X-Y)+Z = (X+Z)-Y$	$\mathrm{P_0}(0,X) = \mathrm{true}$ $X - Y \neq 0 \rightarrow \mathrm{P_0}(s(Y),X) = \mathrm{P_0}(Y,X)$

CRITIC fertilise

Input: *Plan*, *Address*, node(*Plan*, *Address*, $\Gamma' \vdash G'$) ; If current goal, G', is

Precondition: ; an instance of the ; refinement pattern,

G' is of the form $P(C(X)) \rightarrow P(X)$, ; then build and then add

node(*Plan*, [], $\Gamma \vdash G$), ; the associated patching

G is of the form $\forall X.\, P(X) \rightarrow G(X)$; condition to the

Patch: ; original goal

Build P'; base cases are built according to the current definition of P, while the step case is set to $P'(C(X)) = P'(X)$

Output:

$\forall X.\, P'(X) \rightarrow G(X)$

Figure 4: Critic: Working by Refinement

Patching (12) is really complex. It requires the method to build a condition asserting that the last element of L ought to be lesser than or equal to the first element of M, unless L or M is empty. The build_condition/3 predicate cannot handle this level of nested recursion, so the whole abduction mechanism halts failing. A similar situation arises when the abduction mechanism is faced with (13).

Tables 2, 3 and 4 aim to justify the strength of our mechanism. They convey empirical evidence in the form of results obtained from testing the abduction mechanism on a set of examples. This evidence cannot be taken as conclusive, though. For example, more research is to be done in order to characterise the class of faulty formulae our method is able to deal with. Further, we should test the method on larger example problems, or try the incursion of it into other domains. So far, the test set involves problems that we gathered from the literature and, hence, is not representative with respect to any sort of fault.

5 Comparison to Related Work

In his thesis, Moore (1974) introduced a method that attempts to avoid over generalisation. Roughly, the method intercepts any attempt at replacing a compound term by a single variable and then guesses the type information of that term. If the guessed type and the type of any of the variables that appear in the developed formula do not correspond, the method marks the formula as faulty. Then, the method will try to build a, possibly recursive, definition of the guessed type so as to patch the faulty formula. Given that it needs both the original formula and the over generalised one, Moore's method

Table 4: Specifications corrected by refinement

Faulty Conjecture	Patch
$\neg\text{even}(X)$	$\text{odd}(X)$
$\text{double}(\text{half}(X)) \neq X$	$\text{odd}(X)$
$\text{even}(X+Y)$	$\text{even}(X) \wedge \text{even}(Y)$
$\text{odd}(X+Y)$	$\text{even}(X) \wedge \text{odd}(Y)$
$\text{even}(\text{length}(A))$	$p(\text{nil}) = \text{true}$ $p(H :: \text{nil}) = \text{false}$ $p(H_1 :: H_2 :: A) = p(A)$
$\text{odd}(\text{length}(A))$	$p(\text{nil}) = \text{false}$ $p(H :: \text{nil}) = \text{true}$ $p(H_1 :: H_2 :: A) = p(A)$

does not deal with the problem of correcting faulty formulae in general. It is nevertheless a first attempt at fault correction.

Franova and Kodratoff (1992) investigated the problem of mending faulty formulae from a general perspective. They introduced a method, called *PreS*, which builds a corrective condition using the constructive principle of proofs-as-programs (Howard, 1980). Predicate construction takes place along the search for a proof. So a use of induction gives rise to a recursive condition and a use of case analysis to conditional cases. PreS, however, does not consider the immediate abduction of dead end goals and, hence, it cannot handle all of the conjectures shown in Table 2.

Also Protzen (1996) has investigated the use of proofs-as-programs for building corrective conditions. He applied PreS in several interesting faulty formulae, yielding conditional, recursive corrective conditions. However, like Franova and Kodratoff, Protzen does not consider abducible goals. Neither Protzen (1996) nor Franova and Kodratoff (1992) have provided a precise specification of the abduction mechanism, introducing it only by example.

Previously, we introduced a method for correcting faulty formulae (Monroy et al., 1994). The method comprises three proof critics: i) contradictory blocked goals, ii) working by refinement and iii) lochs and dikes. Lochs and dikes aims to complement the job done by working by refinement and, so, is special-purpose. The method is amenable to mechanisation: readers can easily reproduce the reported results. However, the method deals with the problem of correcting faults but partially. For example, it cannot build a corrective condition, only identify it, provided that it is present in the working theory. Similarly, the method cannot correct false statements if failure shows up in the step case of an inductive proof attempt.

In an earlier version of this paper, we pointed out that it would be worthwhile to strive towards the incorporation of proofs-as-programs to our approach. We have successfully developed this observation in (Monroy, 2000), yielding an implementation that is much cleaner and more efficient than the one presented here. In particular, the revised mechanism is able to cope with fairly complex faulty formulae. By comparison, the successful cases reported in this paper are all simple: dealing with each faulty formula requires no more than 3 nested applications of induction.

6 Conclusions

We have introduced an abductive mechanism which works within inductive theorem proving. The mechanism has been implemented using inductive proof planning with critics, built within *Clam*. It is given as a collection of proof critics. Each critic captures the restricted way in which the search for a proof of a faulty formula can fail, as well as providing general knowledge to patch the associated fault. The mechanism avoids a combinatorial explosion. This is both because proof planning carefully guides the search for an inductive proof, and because it assists fault analysis.

Acknowledgements

We are grateful to the anonymous referees for providing invaluable, useful feedback on an earlier version of this paper.

This paper is an extended version of other that appeared in the proceedings of two conferences: i) ENC'99 (Monroy, 1999b), as a conference paper, and ii) TAINA'99 (Monroy, 1999a), as a keynote paper.

This research was partially supported by CONACYT grants SEP-REDII and 33337-A, and by EPSRC grant GR/M/45030.

References

Armando, A., Smaill, A., & Green, I., "Automatic synthesis of recursive programs: The proof-planning paradigm". *Automated Software Engineering, 6*(4), 1999, pp. 329–356.

Basin, D., & Walsh, T., "Difference unification". In *Proceedings of the 13th IJCAI.* International Joint Conference on Artificial Intelligence, 1993. Also available as Technical Report MPI-I-92-247, Max-Planck-Institute für Informatik.

Basin, D., & Walsh, T., "Annotated rewriting in inductive theorem proving". *Journal of Automated Reasoning, 16*(1-2), 1996, pp. 147–80.

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

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

Bundy, A., van Harmelen, F., Hesketh, J., Smaill, A., & Stevens, A. , "A rational reconstruction and extension of recursion analysis". In Sridharan, N. S. (Ed.), *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1989 pp. 359–365. . Also available from Edinburgh as DAI Research Paper No. 419.

Bundy, A., van Harmelen, F., Horn, C., & Smaill, A., "The Oyster-Clam system". In Stickel, M. E. (Ed.), *Proceedings of the 10th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 449. Springer-Verlag, 1990, pp. 647–648. Also available from Edinburgh as DAI Research Paper No. 507.

Bundy, A., "The use of explicit plans to guide inductive proofs". In Lusk, R., & Overbeek, R. (Eds.), *Proceedings of the 9th Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 310, Argonne, Illinois, USA. Springer-Verlag, 1988, pp. 111–120. Also available from Edinburgh as DAI Research Paper No. 349.

Cantu, F., Bundy, A., Smaill, A., & Basin, D., "Experiments in automating hardware verification using inductive proof planning". In Srivas, M., & Camilleri, A. (Eds.), *Proceedings of the Formal Methods for Computer-Aided Design Conference*, Lecture Notes in Computer Science, Vol. 1166, Springer-Verlag, 1996, pp. 94–108. Lecture Notes in Computer Science, Vol. 1166. Also available from Edinburgh as DAI Research Paper No. 828.

De Kleer, J., "Problem solving with the ATMS". *Artificial Intelligence, 28 (2)*, 1986, pp. 197–224.

Flach, P. A., & Kakas, A. C. (Eds.). *Abduction and Induction: Essays on their relation and integration*, Vol. 18 of *Applied Logic Series.* Kluwer Academic Publishers, 2000.

Franova, M., & Kodratoff, Y.,. "Predicate synthesis from formal specifications". In Neumann, B. (Ed.), *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI'92*, Chichester, England. John Wiley and Sons, 1992, pp. 87–91.

Gordon, M. J., Milner, A. J., & Wadsworth, C. P., *Edinburgh LCF—A mechanised logic of computation*, Vol. 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

Howard, W. A., "The formulae-as-types notion of construction". In Seldin, J. P., & Hindley, J. R. (Eds.), *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490. Academic Press, 1980.

Ireland, A., "The Use of Planning Critics in Mechanizing Inductive Proofs". In Voronkov, A. (Ed.), *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, Springer-Verlag, 1992, pp. 178–89. . Lecture Notes in Artificial Intelligence Vol. 624. Also available from Edinburgh as DAI Research Paper No. 592.

Ireland, A., & Bundy, A., "Productive use of failure in inductive proof". *Journal of Automated Reasoning, 16*(1–2), 1996, pp. 79–111. Also available from Edinburgh as DAI Research Paper No 716.

Kakas, A. C., Kowalski, R., & Toni, F., "The role of abduction in logic programming". In Gabbay, D. M., Hogger, C. J., & Robinson, J. (Eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5, Oxford University Press, 1998, pp. 235–324.

Levesque, H. J., "A knowledge-level account of abduction". In Sridharan, N. S. (Ed.), *Proceedings of IJCAI-89*, International Joint Conference on Artificial Intelligence, Morgan Kaufmann, Inc., 1989, pp. 1061–1067.

Monroy, R. "On the correction of faulty formulae". In de Albornoz, A., & Alvarado, M. (Eds.), *Taller Internacional de Inteligencia Artificial, TAINA'99*, D.F., México. Instituto Politécnico Nacional, 1999, pp. 56–68.

Monroy, R., "The use of proof planning failure for correcting faulty conjectures". In Ahuactzin, J. (Ed.), *Encuentro Nacional de Computación, Taller de Lógica*, pp. 1–6 Pachuca. Sociedad Mexicana de Ciencia de la Computación, 1999. ISBN 968-6254-46-3.

Monroy, R., Bundy, A., & Green, I., "Planning proofs of equations in CCS". *Automated Software Engineering*, 7(3), 2000, pp. 263–304.

Monroy, R., Bundy, A., & Ireland, A., "Proof plans for the correction of false conjectures". In Pfenning, F. (Ed.), *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning, LPAR'94*, Lecture Notes in Artificial Intelligence, Vol. 822, Kiev, Ukraine. Springer-Verlag, 1994, pp. 54–68. Also available from Edinburgh as DAI Research Paper No. 681.

Monroy, R., "The use of abduction and recursion-editor techniques for the correction of faulty conjectures". In Flenner, P., & Alexander, P. (Eds.), *Proceedings of the 15th Conference on Automated Software Engineering*, Grenoble, France. IEEE Computer Society Press, 2000, pp. 91–99. Celebrated on September 11-15, 2000.

Moore, J. S., *Computational Logic: Structure Sharing and Proof of Program Properties, Part II*. Ph.D. thesis, University of Edinburgh, 1974. Available from Edinburgh as DCL memo No. 68 and from Xerox PARC, Palo Alto as CSL 75-2.

Peirce, C. S., *Collected papers of Charles Sanders Peirce*, Vol. 2. Harvard University Press, 1959. edited by Harston, C. and Weiss, P.

Protzen, M., "Patching faulty conjectures". In McRobbie, M., & Slaney, J. (Eds.), *13th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 1104, New Brunswick, NJ, USA. Springer-Verlag, 1996, pp. 77–91. Lecture Notes in Artificial Intelligence, Vol. 1104.

Selman, B., & Levesque, H. J., "Abductive and Default Reasoning: A Computational Core". In *AAAI-90, Proceedings of the Eighth National Conference on Artificial Intelligence*, AAAI Press/MIT Press, 1989, pp. 343–348.

__Raúl Monroy Borja__ obtained his PhD in Artificial Intelligence from Edinburgh University, UK, under the supervision of Prof. Alan Bundy. Since 1985 he has been in computing at Tecnológico de Monterrey, Campus Estado de México. From 1992-1998 he was given a leave of absence to conduct graduate studies in UK. After that, he returned to his original lecturer position and in 2000 he was promoted associate professor in Computer Science. Dr. Monroy's research centres on automated (inductive) theorem proving to formal methods of software and hardware development, and the productive use of proof failure to automatically uncover errors in system specifications.

__Alan Bundy__ was educated as a Mathematician, obtaining a 1st class honours degree in Mathematics in 1968 from Leicester University and a PhD in Mathematical Logic in 1971, also from Leicester, under the supervision of Prof. R.L. Goodstein. Since 1971 he has been at the University of Edinburgh. From 1971-73, he was a research fellow on Prof. B. Meltzer's SERC grant 'Theorem Proving by Computer'; in 1973 he became a university lecturer; in 1984 he was promoted to reader; in 1987 he was promoted to professorial fellow; and in 1990 he was promoted to professor. From 1987-92 he held an SERC Senior Fellowship. In 1998 he became Head of the Division of Informatics.