The Use of Abduction and Recursion-Editor Techniques for the Correction of Faulty Conjectures*

Raúl Monroy
Computer Science Department, ITESM-CEM
Carretera al Lago de Guadalupe, Km. 3.5, Atizapán, 52926, México
raulm@campus.cem.itesm.mx

Abstract

The synthesis of programs as well as other synthetic tasks often end up with an unprovable, partially false conjecture. A successful subsequent synthesis attempt depends on determining why the conjecture is faulty and how it can be corrected. Hence, it is highly desirable to have an automated means for detecting and correcting faulty conjectures.

We introduce a method for patching faulty conjectures. The method is based on abduction and performs its task during an attempt to prove a given conjecture. On input $\forall X.\, G(X)$*, the method builds a definition for a* corrective predicate*,* $P(X)$*, such that* $\forall X.\, P(X) \to G(X)$ *is a theorem. The synthesis of a corrective predicate is guided by the constructive principle of formulae as types, relating inference with computation.*

We take the construction of a corrective predicate as a program transformation task. The method consists of a collection of construction commands. A construction command is a small program that makes use of one or more program editing commands, geared towards building recursive, equational procedures.

A synthesised corrective predicate is guaranteed to be correct, turning a faulty conjecture into a theorem. If conditional, it will be well-defined. If recursive, it will also be terminating.

1. Introduction

False assumptions or mal-formulations often originate in concept formation, software development and other error prone tasks. The ability of quickly recovering from one such a fault is the key for success in a subsequent refinement. It is therefore very important to have a means for detecting whether a theory is faulty and suggesting how it could be corrected.

In formal methods, a mal-formulation often takes the form of a partially false conjecture. A *faulty conjecture* is a statement, $\forall X.\ G(X)$, which is not provable within a theory, Γ, written $\Gamma \not\vdash \forall X.\ G(X)$, but it would be if enough conditions were assumed to hold.

We introduce a novel, fully mechanical method for the correction of faulty conjectures. Given Γ and $\forall X.\ G(X)$ such that $\Gamma \not\vdash \forall X.\ G(X)$, the method builds a definition for a *corrective predicate*, $P(X)$, such that $\Gamma \vdash \forall X.\ P(X) \to G(X)$. The construction of a corrective predicate takes place during a proof attempt of the corresponding conjecture. It is guided by the principle known as proofs-as-programs [6], relating inference with computation. So, a recursive predicate corresponds to an inductive proof attempt. Similarly, a conditional predicate corresponds to the use of case analysis.

The definition of a corrective predicate is built taking into account the entire deduction tree. This includes subgoals that could not be established. For example, consider the following faulty conjecture [8]:

$$\forall N{:}\text{nat}.\ \text{double}(\text{half}(N)) = N \qquad (1)$$

where the functions double and half have their natural interpretation, returning twice and half their inputs:[1]

$$\begin{aligned}
\text{double}(0) &= 0\\
\text{double}(s(N)) &= s(s(\text{double}(N)))\\
\text{half}(0) &= 0\\
\text{half}(s(0)) &= 0\\
\text{half}(s(s(N))) &= s(\text{half}(N))
\end{aligned}$$

An attempt to prove (1) using two step structural induction would yield one step case and two base cases, one in which

*The author is grateful to Louise Dennis, Alan Bundy and the anonymous referees for their useful comments on an earlier draft of this paper. This research has been supported by CONACyT-REDII grant.

[1] The function s stands for the successor constructor function on natural numbers.

Case	Result after deduction	Corrective predicate
$\vdash \text{double}(\text{half}(0)) = 0$	Established using symbolic reasoning	$P(0) = \text{true}$
$\vdash \text{double}(\text{half}(s(0))) = s(0)$	Unsolved	$P(s(0)) = \text{false}$
$\text{double}(\text{half}(x)) = x \vdash \text{double}(\text{half}(s(s(x)))) = s(s(x))$	Established using symbolic reasoning and the induction hypothesis	$P(s(s(N))) = P(N)$

Table 1. Outline of an attempt to prove (1) using induction

$N = 0$ and another in which $N = s(0)$. While the first base case, $N = 0$, is readily provable, the second base case, $N = s(0)$, is not. The latter case gives rise to a goal contradicting an axiom of the natural numbers theory, namely: $0 = s(0)$. This suggests that (1) holds if $N = 0$ but does not if $N = s(0)$. Also the step case is readily provable, via symbolic evaluation and the application of the induction hypothesis. Similarly, this result suggests that (1) holds when $N = s(s(x))$, as long as it does when $N = x$. These observations are all summarised in Table 1. The corrective condition, $P(N)$, can be verified to be the predicate *even*. The method therefore suggests that (1) holds only if N is divisible by two.

We take the construction of a corrective predicate as a program transformation task. The method hence consists of a set of high-level construction commands, each associated with an inference rule, or a low-level tactic, and automatically executed on tactic invocation. While a tactic manipulates the current goal, the associated construction command refines upon the definition of the corrective predicate. A construction command is a small program, expressed in a meta-language and makes use of a handful of editing commands geared towards the construction of equational procedures.

Correcting a faulty conjecture, via building a predicate, is not new. The first attempt at automatic fault correction was reported by Moore [9]. In a similar vein, Franova and Kodratoff [5] first pioneered the use of proofs-as-programs for the construction of a corrective condition. Franova and Kodratoff's approach, called *predicate synthesis*, has never been formalised, being described only by example, hence, making it difficult for readers to reproduce their results.

This paper makes two key contributions: First, it gives a formal means of mechanisation of predicate synthesis and other fault correction techniques found in the literature [8, 10]. Second, it shows that an abduction framework, together with a methodology based on editing and construction commands, is helpful in automatically constructing patching conditions for faulty conjectures. Our results cover all those found in the literature, surpassing each existing method individually.

We shall demonstrate that, by construction, a synthesised corrective predicate is correct, turning a faulty conjecture into a theorem. If conditional, the procedure will be well-defined and, if recursive, it will also be terminating.

The rest of this paper is organised as follows: In §2 we describe the editing commands used in the construction or refinement of a corrective predicate, while in §3 we show how to use them giving rise to a small collection of construction commands. Also in §3 we illustrate general aspects of the method with a running example. In §4 we describe implementation aspects. In §5 we illustrate the strength of our method by showing a small but representative set of results. We discuss related work in §6. Finally, future work and drawn conclusions appear in §7

2. Editing Commands

Predicate construction does not require all of the editing commands that one would find in a typical recursion-techniques, program editor — see for example [2]. This is because we refine upon each predicate incrementally. For example, the arity of a predicate may increase but may not decrease. Similarly, the order of the arguments in a predicate may not change. This section provides a brief description of each editing command.

In the sequel, we assume an infinite, countable set of fresh predicate names, $\mathcal{P}$, and use $P, Q, R, \ldots$ to stand for meta-variables ranging over $\mathcal{P}$. Also we assume a set of variables, $\mathcal{V}$, and let $X, Y, \ldots$ range over $\mathcal{V}$. $\vec{X}$ denotes a tuple of distinct variables. We shall use B (respectively S) to represent meta-variables ranging over base (respectively step) recursive constructor functions.

A corrective predicate fits into this primitive recursion schema:

$$\begin{aligned} P(B, Y) &\leftrightarrow Q(Y) \\ P(S(X, Z), Y) &\leftrightarrow R(X, Y, Z, P(X, Y)) \end{aligned} \tag{2}$$

where P is the predicate being defined; Q and R are the predicates P is defined in terms of; X is the recursion variable; Y is the *regular parameter*; and Z is the *constructor parameter*. A predicate schemata for P is a set of case schemata for P, each of which is of the form:

$$\mathit{Condition} \rightarrow \mathit{Head} \leftrightarrow \mathit{Body}$$

where Head is of the form $\mathsf{P}(\ldots)$ and where $\mathit{Condition}$ and Body are formulae. A term is either a constant, a variable, or a function applied to some terms. A formula is either a predicate applied to some terms or some formulae closed under logical operators and (universal and existential) quantifiers.

2.1. Increasing the Arity of Predicates

There are two kinds of commands for increasing the arity of predicates: One, called `addParameter`, is not recursive and the other one, called `addArgument`, is. They are respectively restricted to the addition of parameters and of recursion arguments.

Adding Parameters `addParameter` is used whenever the whole schema requires an extra parameter. Upon selection of this command, we must indicate three things: i) the name of the predicate to be modified, ii) the number of new parameters, and iii) the type of each — regular or constructor. New parameters are inserted always at the end of a predicate, which may currently be parameterless. For example, if the number of parameters were 1, with the type being regular, in (2) the result would be:

$$\begin{array}{rcl} \mathsf{P}(\mathsf{B}, \mathsf{Y}, W) & \leftrightarrow & \mathsf{Q}(\mathsf{Y}, W) \\ \mathsf{P}(\mathsf{S}(\mathsf{X}, \mathsf{Z}), \mathsf{Y}, W) & \leftrightarrow & \mathsf{R}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, W, \mathsf{P}(\mathsf{X}, \mathsf{Y}, W)) \end{array}$$

Note that we have placed a fresh variable in the position of the new parameter.

Adding Arguments Adding a new recursion argument to a predicate not only amounts to giving the predicate an extra argument but also to multiplying the number of cases. Each old case is replaced with two cases: one in which the new argument takes the base value, and one in which it takes the step value. To use this command, we must indicate the name of the predicate under construction and the sort of recursion schema with which is to be augmented. For instance, if we applied `addArgument` to (2), using one step induction over natural numbers, the second case would be:

$$\begin{array}{rcl} \mathsf{P}(\mathsf{S}(\mathsf{X}, \mathsf{Z}), \mathsf{Y}, 0)) & \leftrightarrow & \mathsf{R}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, 0, \mathsf{P}(\mathsf{X}, \mathsf{Y}, 0)) \\ \mathsf{P}(\mathsf{S}(\mathsf{X}, \mathsf{Z}), \mathsf{Y}, \mathrm{s}(W)) & \leftrightarrow & \mathsf{R}(\mathsf{X}, \mathsf{Y}, \mathsf{Z}, W, \mathsf{P}(\mathsf{X}, \mathsf{Y}, W)) \end{array}$$

This transformation ensures that P is still both well-defined and terminating. `addArgument` introduces a new recursion argument at the end of the predicate in the same way as its twin.

2.2. Instantiating the Schema

The command `solve` allows us to instantiate a meta-variable, ranging over predicates, variables, etc., with an actual predicate, variable, etc. The scope of predicate instantiation is global, while that of variable instantiation is local to each case. To perform an instantiation, we have to invoke this command indicating the symbol to be renamed, together with the new symbol value. For example, we may instantiate R, the predicate that appears in the second case of (2), to the proposition true, yielding:

$$\mathsf{P}(\mathsf{S}(\mathsf{X}, \mathsf{Z}), \mathsf{Y}) \leftrightarrow \text{true}$$

2.3. Introducing Conditional Cases

The `multiplyCases` command is used whenever an existing case needs to be expanded into several sub-cases. So `multiplyCases` is complementary to `addArgument`, which, according to the sort of recursion schema, provides a fixed number of cases to a predicate procedure. `multiplyCases` takes three arguments: i) the predicate under construction; ii) the case; and iii) a case split, $\{\mathsf{C}_1, \ldots, \mathsf{C}_n\}$, of size n, a positive integer indicating the number of cases with which the original case should be replaced. The system uses C_i as the i-th condition. Each C_i $(1 \leq i \leq n)$ is a predicate applied to the free variables of the current predicate. For example, if the case split were $\{\mathsf{C}, \neg\mathsf{C}\}$ and `multiplyCases` were applied to the first case of (2), the result would be:

$$\begin{array}{rcl} \mathsf{C}(\mathsf{Y}) & \rightarrow & \mathsf{P}(\mathsf{B}, \mathsf{Y}) \leftrightarrow \mathsf{Q}(\mathsf{Y}) \\ \neg\mathsf{C}(\mathsf{Y}) & \rightarrow & \mathsf{P}(\mathsf{B}, \mathsf{Y}) \leftrightarrow \mathsf{Q}(\mathsf{Y}) \end{array}$$

2.4. Predicate Simplification

We use a small collection of simplification commands. The `reduce` command, for example, eliminates a very simple form of unnecessary recursion. For example, given a predicate of the schematic form:

$$\begin{array}{rcl} \mathsf{P}(\mathsf{B}, \mathsf{Y}) & \leftrightarrow & \mathsf{Q}(\mathsf{Y}) \\ \mathsf{P}(\mathsf{S}(\mathsf{X}), \mathsf{Y}) & \leftrightarrow & \mathsf{P}(\mathsf{X}, \mathsf{Y}) \end{array}$$

`reduce` returns just $\mathsf{Q}(\mathsf{Y})$. `simplify` is another example simplification command. It performs a simple check to eliminate unnecessary cases. For example, given a predicate of the form:

$$\begin{array}{rcl} \mathsf{P}(0) & \leftrightarrow & \text{true} \\ \mathsf{P}(\mathrm{s}(0)) & \leftrightarrow & \text{false} \\ \mathsf{P}(\mathrm{s}(\mathrm{s}(\mathsf{X}))) & \leftrightarrow & \text{true} \end{array}$$

`simplify` returns $X \neq s(0)$ instead.[2] Other forms of simplification are also considered. They are all simple, and trivially justifiable. We shall not discuss the issue further here.

2.5. Updating Predicate Dependencies

Since the arity of a procedure may change upon edition, care is to be taken to keep all the procedures well-defined. To this aim, the system maintains a dependency graph relating procedures.

In a *dependency graph*, each procedure, P, is connected by an arrow to each of the procedures in terms of which it is defined, $Q, R, \ldots$. We write $P \gg Q, P \gg R, \ldots$ to denote dependency relations. A procedure should label a leaf iff it is a constructor function or a procedure in the working theory. The system ensures that the dependency graph never becomes cyclic.

`updateDependencies` updates the structure of all the procedures, ensuring well-definedness. For example, if $P \gg Q$ and if the arity of Q has changed, then `updateDependencies` will update the arity of P, increasing it or leaving it unmodified as required (N.B. This command makes use of `addParameter`.) `updateDependencies` is guaranteed to terminate, both because the dependency graph is finite and because the graph is acyclic. Note that after `updateDependencies`, the dependency graph is still acyclic.

Editing commands are combined into a unit to perform predicate construction. Such a unit is called a construction command and is discussed next.

3. Construction Commands

This section describes the commands with which we build corrective predicates. Each construction command is associated with a low-level tactic, and automatically executed on tactic invocation. Thus, the correction task is performed upon a proof method.

The proof method is assumed to operate on sequents of the form $\Gamma \vdash G$. G is a formula and Γ is a set of formulae. Each formula in Γ is considered a hypothesis. Upon application, the proof method is assumed to build a typical deduction tree. Each node of a deduction tree should contain at least the following information: i) the result of the deduction process so far, called the *goal*; ii) markers to indicate whether the node is root, leaf, or intermediate; iii) the tactic or inference rule used to explore the node, if any; iv) a means to reach the parent and child nodes; and v) the current definition of the abducted predicate. Also each node, or more precisely the expression at each node, is assumed to be labelled by a predicate schema, or a case schema for a predicate schema. This piece of information, manipulated by our abduction mechanism, will not necessarily change from one node to the other.

At the beginning of a deduction process, the input goal, $\forall \vec{X}.\ G(\vec{X})$, is labelled with P_0. Initially P_0 is parameterless, non-recursive and undefined. The aim of the method is to instantiate P_0, possibly adding extra conditions $P_0 \wedge C_1 \wedge \ldots$, where each C_i should have been abducted during a proof attempt. As will become clear, $\forall \vec{X}.\ P_0(\vec{X}) \to G(\vec{X})$ is a theorem by construction. If $G(\vec{X})$ is false for all $\vec{X}$, P_0 will be set to false.[3] Conversely, if $\forall \vec{X}.\ G(\vec{X})$ is a theorem, P_0 will be trivially set to true. If $G(\vec{X})$ is faulty, P_0 will be other than true or false and an algorithm for P_0 will be constructed or identified.

Now we introduce each construction command, illustrating its workings by an example application. We shall use the following running example:

$$\forall L\!:\alpha\ \text{list}.\ \text{sort}(L) = L \tag{3}$$

Although awkward at first glance, (3) 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.[4] Let us assume the availability of the following rewrite rules:

$$\begin{array}{rcl}
\text{insert}(X, \text{nil}) & :\Rightarrow & X :: \text{nil} \\
X \leq Y \to \text{insert}(X, Y :: L) & :\Rightarrow & X :: Y :: L \\
X \not\leq Y \to \text{insert}(X, Y :: L) & :\Rightarrow & Y :: \text{insert}(X, L) \\
\text{sort}(\text{nil}) & :\Rightarrow & \text{nil} \\
\text{sort}(X :: L) & :\Rightarrow & \text{insert}(X, \text{sort}(L))
\end{array}$$

Structural Induction The first construction command we examine is `induction`. `induction` is invoked upon the application of an induction principle suggesting that the corrective predicate might be recursive. Each induction principle used by the proof method is assumed to be indexed (and identified) by two parameters: i) the sort of (structural) induction and ii) the induction variable. Accordingly, `induction` makes use of such information in order to create a predicate with the associated dual recursive structure.

The new predicate will have as many cases as the sum of the base cases and the step cases of the induction principle. Each base or step case is pairwise associated with a case, which it instantiates. If there were b base cases, then the i-th base case, $1 \leq i \leq b$, is associated with the i-th case. Similarly, if there were s step cases, then the i-th step case, $1 \leq i \leq s$, is associated with the $i + b$-th case.

[2]Do not confuse the predicate P with even. Even is recursive while P is not (c.f. the third case).

[3]Notice that, however, if P_0 is set to false, then $\forall \vec{X}.\ G(\vec{X})$ is not necessarily false. This means that P_0 does not characterise the set of *all* elements for which $G(\vec{X})$ holds.

[4]An *over-generalisation* is an operation that transforms a true statement into a false one.

`induction`		
Inputs:	*Current corrective predicate:*	$\mathsf{P_i}$
	Type of induction principle:	`IndType`
	Induction variable:	`IndVar`
Action:	`solve`$(\mathsf{P_i}/\mathsf{P_i})$, $\mathsf{P_i}$ is a fresh predicate	
	`addArgument`$(\mathsf{P_i}, \mathtt{IndType}:\mathtt{IndVar})$	
	`updateDependencies`	

To illustrate the workings of `induction`, consider an attempt to prove (3) using list induction on L. The base case and step case are:

$$\vdash \text{sort}(\text{nil}) = \text{nil} \tag{4}$$

$$l{:}\ldots, h{:}\ldots, ih(i) : \text{sort}(l) = l \vdash \text{sort}(h :: l) = h :: l \tag{5}$$

`induction` would take an actual predicate, P_0, yielding the following refinement:

$$\begin{aligned} P_0(\text{nil}) &\leftrightarrow P_1 \\ P_0(H :: L) &\leftrightarrow P_2(H, P_0(L)) \end{aligned}$$

P_1 and P_2 will be respectively solved by the deduction process originated by goals (4) and (5).

Closing Deduction Branches The `elementary` construction command is invoked whenever a deduction branch is closed. It may be associated with a low-level tactic responsible for performing simple reasoning, including arithmetic, type checking, propositional-tautology checking, etc. The tactic might be able to apply one or more hypotheses, but none of them should be an induction hypothesis.

`elementary`	
Inputs:	*Current corrective predicate:* $\mathsf{P_i}$
Action:	`solve`$(\text{true}/\mathsf{P_i})$

Going back to our running example, the base case, (4), is easily established using symbolic reasoning. So P_1 is set to true, yielding the following predicate transformation:

$$\begin{aligned} P_0(\text{nil}) &\leftrightarrow \text{true} \\ P_0(H :: L) &\leftrightarrow P_2(H, P_0(L)) \end{aligned}$$

The Use of an Induction Hypothesis The use of induction is necessary but not sufficient for a predicate to be recursive. If recursion is present then one or more induction hypotheses are used. This use of an induction hypothesis, called *fertilisation*, is a crucial part of inductive theorem proving. There are two kinds of fertilisation, strong and weak. The use of strong fertilisation may close the current deduction branch while the use of weak fertilisation may not.

`fertilise`	
Inputs:	*Current corrective predicate:* $\mathsf{P_i}$
Condition:	$\exists PP.\ PP \gg P_i$; i.e., $PP(S(X)) \leftrightarrow P_i$
Action:	`solve`$(PP(X) \wedge P_j/P_i)$
	N.B. i) P_j is a fresh variable
	ii) new goal labelled with P_j
	iii) if new goal empty P_j=true

If recursive, a corrective predicate is guaranteed to be terminating, because each induction principle we use is well-founded. The induction order becomes the recursion order.

For our running example, the step case, (5), via symbolic reasoning followed by weak fertilisation, gives rise to this goal formula:

$$\ldots \vdash \text{insert}(h, l) = h :: l \tag{6}$$

Accordingly, $P_0(H :: L) \leftrightarrow P_0(L) \wedge P_3$. P_3 is to be refined by the deduction process involved in a proof attempt of (6).

Since l is a free variable, an attempt to prove (6) also proceeds by list induction, which expands it into two new goals:

$$\vdash \text{insert}(h, \text{nil}) = h :: \text{nil} \tag{7}$$

$$\ldots, ih(j) : \text{insert}(h, l) = h :: l \vdash \text{insert}(h, h' :: l) = h :: h' :: l \tag{8}$$

So, `induction` instantiates P_3, leaving:

$$\begin{aligned} P_0(\text{nil}) &\leftrightarrow \text{true} \\ P_0(H :: L) &\leftrightarrow P_0(L) \wedge P_3(L) \\ P_3(\text{nil}) &\leftrightarrow P_4 \\ P_3(H' :: L) &\leftrightarrow P_5(H', L, P_3(L)) \end{aligned}$$

Again, P_4 and P_5 will be respectively solved using the deduction process originated by a proof attempt of (7) and (8). Notice that (7) can be trivially established and so $P_3(\text{nil}) \leftrightarrow \text{true}$. By contrast, a proof of (8) is not immediate.

Case Analysis The use of a hypothesis of the form $C_1 \vee \ldots \vee C_n$ gives rise to a proof attempt by case analysis. Case analysis in turn gives rise to an expansion in the number of cases of a corrective predicate. The `caseSplit` construction command is invoked whenever a proof attempt is split into cases. Unlike previous commands, `caseSplit` does not modify the current corrective predicate, P_i, but it modifies the predicate P_i is defining, $PP \leftrightarrow P_i$.

`caseSplit`		
Inputs:	*Current corrective predicate:* $\mathsf{P_i}$	
	Case split:	$\{C_1, \ldots, C_n\}$
Condition:	$\exists PP.\ PP \gg P_i$; that is, $PP(\overrightarrow{X}) \leftrightarrow P_i$	
Action:	`multiplyCases`$(PP, P_i, \{C_1, \ldots, C_n\})$	
	`updateDependencies`	

We assume a finite set of case splits, Ξ, each of which is complete. A case split, $\{C_1, \ldots, C_n\}$, is said to be *complete* if $C_1 \vee \ldots \vee C_n$ holds but $C_i \wedge C_j$, for $1 \le i < j \le n$, does not. A conditional, corrective predicate is guaranteed to be well-defined, because each case split we use is complete.

The workings of `caseSplit` are now illustrated. Handling (8) requires an application of case analysis, with case split $\{h \le h', h \not\le h'\}$:

$$\ldots, h \le h' \vdash \text{insert}(h, h' :: l) = h :: h' :: l$$
$$\ldots, h \not\le h' \vdash \text{insert}(h, h' :: l) = h :: h' :: l$$

The second case of $P_3(H :: L)$ is hence expanded into two sub-cases. After updating dependencies, the partial definition of P_0 is as follows:

$$\begin{array}{rcl}
P_0(\text{nil}) & \leftrightarrow & \text{true} \\
P_0(H :: L) & \leftrightarrow & P_0(L) \wedge P_3(L, H) \\
P_3(\text{nil}, H) & \leftrightarrow & \text{true} \\
H \le H' \rightarrow P_3(H' :: L, H) & \leftrightarrow & P_5(H', L, H, P_3(L, H)) \\
H \not\le H' \rightarrow P_3(H' :: L, H) & \leftrightarrow & P_5(H', L, H, P_3(L, H))
\end{array}$$

The first case, $h \le h'$, is easily established without using fertilisation. So the body of the first step equation, $X \le H$, is set to true. The second case, $h \not\le h'$, however, cannot be proved, for $h :: h' :: l = h' :: h :: l$ is not a valid formula and, thus, is rejected. These are the sorts of goal formulae handled by the following and last construction command.

Dead End Goals The `deadEnd` construction command is invoked whenever a dead end goal is detected. A goal is said to be a *dead end* if it cannot be manipulated further but it has not yet been proved. The application of `deadEnd` disables the use of backtracking and the proof attempt is continued. The action of `deadEnd` is to check whether the current goal is either invalid or abducible.

A formula, P, is said to be *invalid* if it represents an obvious contradiction of some axiom. It is said to be *abducible* if there exists some substitution, ϕ, such that $P\phi$ is variable-free and such that $\vdash P\phi$. We assume two finite sets, one of invalid formulae, ∇, and the other one of abducible predicates, Δ. E.g., for the natural numbers of Peano $\nabla = \{X \ne X, X = s(X), X \ne 0 \wedge X = p(X), p(X) = s(X)\}$, where p is the predecessor function on that type, and $\Delta = \{X = Y, X \ne Y, \ldots\}$. Often, ∇ can be built fully automatically.

If the goal is invalid, the system sets the current case to false. Otherwise, if the goal is abducible, it sets the current case to true and then adds the goal as an abducted condition, provided that the goal does not clash with the current abducted conditions. Ξ is used to determine whether two or more (abducted) conditions are contradictory. Users may change ∇, Δ or Ξ, possibly leaving them empty but — as we shall see in §5 — the quality of a corrective predicate may improve when they are not.

`deadEnd`

Inputs:	*Current corrective predicate:* P_i *Current goal:* $\Gamma \vdash G$
Action:	if $\exists G' \in \nabla.\ G'\psi = G$, for some term substitution ψ, then `solve`(false/P_i), or if $\exists G' \in \Delta.\ G'\psi = G$ and if $\{G, P_0\} \cup \{\text{abducted predicates}\} \subseteq \xi$, for some $\xi \in \Xi$, then `solve`(false/P_i) otherwise `solve`(true/P_i) and abduct(G)

Going back to our example, since the second case goal, $h \not\le h'$, yields a dead end goal, the mechanism sets the body of the second step equation, $X \not\le H$, to false. It turns out that P_0 constitutes the predicate *ordered* on lists:

$$\begin{array}{rcl}
P_0(\text{nil}) & \leftrightarrow & \text{true} \\
P_0(H :: L) & \leftrightarrow & P_0(L) \wedge P_3(L, H) \\
P_3(\text{nil}, H) & \leftrightarrow & \text{true} \\
H \le H' \rightarrow P_3(H' :: L, H) & \leftrightarrow & \text{true} \\
H \not\le H' \rightarrow P_3(H' :: L, H) & \leftrightarrow & \text{false}
\end{array}$$

This completes our description of the method. We now discuss implementation aspects as a preparation for illustrating the experimental results.

4. Implementation Aspects: Proof Planning

The abduction mechanism presented in this paper has been implemented within inductive proof planning [1], built in the *Clam* system [4]. The rationale behind this implementation decision was twofold: i) inductive proof planning provides a significant level of automation in inductive theorem proving; and ii) it works at the meta-level, hence, facilitating the modification of the search control engine. The careful search guidance provided by inductive proof planning made our experimental results possible.

Proof planning is a meta-level reasoning technique, developed especially to automate theorem proving. It works in the context of a tactical style of reasoning, using AI planning techniques to build large complex tactics from simpler ones.

Methods are the building-blocks of proof planning. A *method* is a high-level description of a tactic. It specifies the preconditions under which the tactic is applicable and the effects of its application. The application of a method to a given goal consists in checking the preconditions against the goal and then determining the output new goals by computing the effects.

Proof planning returns a *proof plan* (i.e., a customised tactic together with its justification), whose execution, in the normal case of success, guarantees correctness of the final proof. Proof planning is cheaper than searching for a proof in the underlying object theory. This is both because each plan step covers a lot of ground steps, and because the method preconditions dramatically restrict the search space.

Inductive proof planning is the application of proof planning to inductive theorem proving. It is characterised by the following methods: The *induction* method selects the most promising induction scheme via a process called *rippling analysis*. 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 *ripple* and *fertilise* methods. Ripple applies *rippling* [3], a heuristic that guides transformations in the induction conclusion to enable fertilisation.

The construction commands are named according to the methods of inductive proof planning. They are all associated in the expected manner — induction and `induction`, elementary and `elementary`, etc. — except for `deadEnd`, which has been implemented via an exception handler mechanism.

The incorporation of an exception handler within proof planning is called *proof planning with critics* [7]. A *critic* is a small program and is invoked if both the goal at hand is blocked and 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 with the partially applicable method, if any, one at a time, in the order of appearance. We have partially implemented proof planning with critics within *Clam* version 2.7.

5. Experimental Results

Our abduction mechanism is capable of patching various classes of faulty conjectures fully automatically. Conjectures that overlook boundary case conditions [8], for example, can be tackled using `elementary`, `deadEnd`, `caseSplit` and `induction` only. Some example conjectures belonging to this class are shown in Table 2.

Another class of faulty conjectures refers to overlooking properties of some of the operators involved in the specification. By adding `fertilise` to the construction commands above mentioned, we have dealt with conjectures belonging to this class; some of them are shown in Table 3.

The information displayed in these tables is as follows. For each faulty conjecture, we provide the patch proposed by our system and, if necessary, the definition of the patch. Examples #1–6, 8, 11–12 and 14 all involve corrective predicates with straightforward interpretation. A semantic of the corrective predicates in the remaining examples is given below.

Example #7: The patch specifies that ordering insertion amounts to list construction as long as the element to be inserted is lesser than or equal to the element that is in the head of the list, if any. Example #9: Sorting the concatenation of two lists amounts to concatenating the result of sorting each list separately, only if each element in the first list is lesser than or equal to every element in the second one. Example #10: List concatenation is commutative, as long as the lists in question are equal. This patch does not take into account that the conjecture is valid even in cases where the lists in question are not equal, e.g., $A = x :: y :: L$ and $B = x :: y :: x :: y :: L$]. Example #13: The length of a list is divisible by two provided that the list is empty or that it grows in chunks of size two. Example #15: In the context of natural numbers, $(X - Y) + Z = (X + Z) - Y$ holds if $X > Y$. Again, this patch does not take into account that the conjecture also holds when $X = Y$.

Before concluding this section, we illustrate the way in which the quality of a patch can be improved by making use of the set of abducibles, ∇, and the set of case splits, Ξ. Consider the example conjecture #12. If both ∇ and Ξ were empty, the method would yield a corrective predicate with the following structure:

$$\begin{aligned} P_0(0, Y) &\leftrightarrow \text{even}(Y) \\ P_0(s(0), Y) &\leftrightarrow \text{odd}(Y) \\ P_0(s(s(X)), Y) &\leftrightarrow P_0(X) \end{aligned}$$

P_0 is assumed to have been built out of an attempt to prove conjecture #12 using two-step induction. The patch shows that to prove both the first base case, $X = 0$, and the second one, $X = s(0)$, we need to prove that the natural numbers are all even and that, if different from 0, they are all odd. If we let $\nabla = \{\text{even}, \text{odd}\}$, then the fact that Y should be even would be abducted as a top-condition: $P_0(X) \wedge \text{even}(Y)$; P_0 is as before except that the body of the first case is true. Furthermore, if we let $\Xi = \{\{\text{even}, \text{odd}\}\}$, then the method can detect that the second case of P_0 actually evaluates to false. In that case, P_0 can easily be verified to constitute the predicate *even*.

Now we discuss related work. We shall see that our results cover all those found in the literature, surpassing each existing method individually.

6. Related Work

Moore introduced a method that tries to deal with a faulty conjecture that has developed from an over-generalisation.

#	Faulty Conjecture	Patch	Definition
1	$\text{length}(\text{app}(A, B)) > \text{length}(A)$	$B \neq \text{nil}$	
2	$\text{length}(\text{app}(A, B)) > \text{length}(B)$	$A \neq \text{nil}$	
3	$X + Y > X$	$Y \neq 0$	
4	$X + Y > Y$	$X \neq 0$	
5	$\text{double}(X) > X$	$X \neq 0$	
6	$X > \text{half}(X)$	$X \neq 0$	
7	$\text{insert}(X, L) = X :: L$	$P_0(L, X)$	$P_0(\text{nil}, X) \leftrightarrow \text{true}$ $X \leq H \rightarrow P_0(H :: T, X) \leftrightarrow \text{true}$ $X \not\leq H \rightarrow P_0(H :: T, X) \leftrightarrow \text{false}$

Table 2. Faulty conjectures overlooking boundary case conditions

Roughly, the method intercepts any attempt of replacing a compound term by a single variable and then constructs the type information of that term. If the constructed type and the type of any of the variables that appear in it do not correspond, the method will try to build a recursive definition of the constructed type. Given that it needs both the original formula and the over-generalised formula, Moore's method does not deal with the problem of correcting faulty conjectures in general.

Franova and Kodratoff have investigated the problem of mending faulty conjectures [5]. They introduced a method, called *PreS*, which, as opposed to Moore's method, does not look into type information. Franova and Kodratoff tried to tackle the problem from a general perspective and have pioneered the use of proofs-as-programs to approach it. However, they have not described PreS formally, introducing it by example only. By contrast, we provide a formal means of mechanisation. Franova and Kodratoff do not consider the immediate abduction of dead end goals. Hence, they cannot handle all of the conjectures shown in Table 2. To reproduce the results shown in [5], we have used the full set of construction commands.

Protzen has also investigated the application of proofs-as-programs for patching faulty conjectures [10]. He investigated the use of PreS in several interesting faulty conjectures, yielding conditional, recursive corrective predicates. However, like Franova and Kodratoff, Protzen did not provide a procedural account of PreS, introducing it by example only.

In a similar vein, Monroy, Bundy and Ireland have introduced a method for correcting faulty conjectures [8]. They provide a complete characterisation of their method; readers can easily reproduce their results. However, they only partially deal with the problem of correcting faults. For example, they cannot build a corrective predicate, only identify it — as long as it is present in the working theory. Similarly, they cannot correct false statements if the failure can only be detected in the step case of an inductive proof attempt. Using `elementary`, `deadEnd`, `induction` and `fertilise` only, we have reproduced the results of Monroy, Bundy and Ireland.

Theorem proving is not the only formulation to program synthesis to have been proposed. Several researchers have explored the use of knowledge-based systems to synthesise computer programs. Related to ours is the work of Smith [11], who introduced a system, called *Rainbow*, that attempts to mend faulty conjectures.[5] Rainbow is knowledge based, and does not take into account the notion of formulae as types. Some of the examples with which we have tested our system have their source in Smith's papers.

7. Conclusions and Future Work

We have described a mechanism for correcting faulty conjectures via constructing a predicate. The mechanism is able to identify or synthesise, if necessary, the definition of the corrective predicate. Predicate construction is approached using low-level commands, typical of a program editor. Editing commands are geared towards the construction of a recursive, possibly conditional, equational procedure. Combining editing commands, we form high-level construction commands, used to build or refine the definition of a procedure. The application of construction commands is guided by the constructive principle known as proof-as-programs.

The procedural aspects of the mechanism are fully given. We have successfully tested our abduction mechanism using a number of faulty conjectures drawn from the literature. The mechanism is implemented within inductive proof planning, built in *Clam*. We have demonstrated that the mechanism equals the coverage of existing techniques, surpassing each of them individually.

Ongoing and future work concerns the application of this technique to larger, industrial strength faulty conjectures, determining what sorts of extensions are required. In particular, we envision the need of sophisticated simplification

[5] Smith calls the problem of mending a faulty formula via a corrective predicate the *antecedent-derivation problem*.

#	Faulty Conjecture	Patch	Definition
8	$\mathrm{sort}(L) = L$	$\mathrm{ordered}(L)$	
9	$\mathrm{sort}(\mathrm{app}(A, B)) = \mathrm{app}(\mathrm{sort}(A), \mathrm{sort}(B))$	$P_0(A, B)$	$P_0(\mathrm{nil}, B) \leftrightarrow \mathrm{true}$ $P_0(H :: A, B) \leftrightarrow P_0(A, B) \wedge P_1(B, H)$ $P_1(\mathrm{nil}, X) \leftrightarrow \mathrm{true}$ $X \leq H \rightarrow P_1(H :: B, X) \leftrightarrow P_1(B, X)$ $X \not\leq H \rightarrow P_1(H :: B, X) \leftrightarrow \mathrm{false}$
10	$\mathrm{rev}(\mathrm{rev}(\mathrm{app}(A, B))) = \mathrm{app}(B, A)$	$P_0(A, B)$	$P_0(\mathrm{nil}, B) \leftrightarrow \mathrm{true}$ $P_0(H :: A, B) \leftrightarrow P_0(A, B) \wedge P_1(B, H)$ $P_1(\mathrm{nil}, X) \leftrightarrow \mathrm{true}$ $X = H \rightarrow P_1(H :: B, X) \leftrightarrow P_1(B, X)$ $X \neq H \rightarrow P_1(H :: B, X) \leftrightarrow \mathrm{false}$
11	$\mathrm{double}(\mathrm{half}(X)) \neq X$	$\mathrm{odd}(X)$	
12	$\mathrm{even}(X + Y)$	$\mathrm{even}(X) \wedge \mathrm{even}(Y)$	
13	$\mathrm{even}(\mathrm{length}(A))$	$P_0(A)$	$P_0(\mathrm{nil}) \leftrightarrow \mathrm{true}$ $P_0(H :: \mathrm{nil}) \leftrightarrow \mathrm{false}$ $P_0(H_1 :: H_2 :: A) \leftrightarrow P_0(A)$
14	$\prod(P, L) = P(I) \Big\vert \prod(P, \mathrm{del}(I, L))$	$\mathrm{member}(I, L)$	
15	$(X - Y) + Z = (X + Z) - Y$	$P_0(Y, X)$	$P_0(0, X) \leftrightarrow \mathrm{true}$ $P_0(s(Y), X) \leftrightarrow P_0(Y, X) \wedge X - Y \neq 0$

Table 3. Faulty conjectures overlooking properties of operators

commands. This would yield simple corrective predicates, a necessary condition for understanding the root of malformulations.

The abduction mechanism sometimes yields corrective predicates that we consider “unnatural”. Ongoing work is thus related to trying to characterise families of corrective predicates. This information would then be used for improving the quality of a patch.

References

[1] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *Proceedings of the 9th Conference on Automated Deduction*, Lecture Notes in Computer Science, Vol. 310, pages 111–120, Argonne, Illinois, USA, 1988. Springer-Verlag.

[2] A. Bundy, G. Grosse, and P. Brna. A recursive techniques editor for Prolog. *Instructional Science*, 20:135–72, 1991.

[3] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[4] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 449, pages 647–648. Springer-Verlag, 1990.

[5] M. Franova and Y. Kodratoff. Predicate synthesis from formal specifications. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI’92*, pages 87–91, Chichester, England, 1992. John Wiley and Sons.

[6] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[7] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[8] R. Monroy, A. Bundy, and A. Ireland. Proof Plans for the Correction of False Conjectures. In F. Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning, LPAR’94*, Lecture Notes in Artificial Intelligence, Vol. 822, pages 54–68, Kiev, Ukraine, 1994. Springer-Verlag.

[9] J. S. Moore. *Computational Logic: Structure sharing and proof of program properties, part II.* PhD thesis, University of Edinburgh, 1974.

[10] M. Protzen. Patching faulty conjectures. In M. McRobbie and J. Slaney, editors, *13th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 1104, pages 77–91, New Brunswick, NJ, USA, 1996. Springer-Verlag.

[11] D. R. Smith. The top-down synthesis of divide and conquer algorithms. *Artificial Intelligence*, 27(1), 1985.