Division of Informatics, University of Edinburgh

Institute for Representation and Reasoning

Multi-Predicate Induction Schemes for Mutual Recursion

by

Richard Boulton

Informatics Research Report EDI-INF-RR-0014

Division of Informatics **April 2000**
http://www.informatics.ed.ac.uk/

Multi-Predicate Induction Schemes for Mutual Recursion

Richard Boulton

Informatics Research Report EDI-INF-RR-0014

DIVISION *of* INFORMATICS
Institute for Representation and Reasoning

April 2000

Abstract :

Where mutually recursive data types are used in programming languages, etc., mutually recursive functions are usually required. Mutually recursive functions are also quite common for non-mutually recursive types. Reasoning about recursive functions requires some form of mathematical induction but there have been difficulties in adapting induction methods for simple recursion to the mutually recursive case. This paper proposes the use of multi-predicate induction schemes in the context of explicit induction and presents a proof method for their use within a proof planning system. An implementation in Clam has successfully planned proofs for a number of mutually recursive examples.

Keywords : Automated Reasoning, Theorem Proving, Mathematical Induction, Mutual Recursion

Multi-Predicate Induction Schemes for Mutual Recursion*

Richard J. Boulton

DIVISION *of* INFORMATICS
Institute for Representation and Reasoning

April 6, 2000

Abstract: Where mutually recursive data types are used in programming languages, etc., mutually recursive functions are usually required. Mutually recursive functions are also quite common for non-mutually recursive types. Reasoning about recursive functions requires some form of mathematical induction but there have been difficulties in adapting induction methods for simple recursion to the mutually recursive case. This paper proposes the use of multi-predicate induction schemes in the context of explicit induction and presents a proof method for their use within a proof planning system. An implementation in *Clam* has successfully planned proofs for a number of mutually recursive examples.

Keywords: Automated Reasoning, Theorem Proving, Mathematical Induction, Mutual Recursion

1 Introduction

The abstract syntax of programming languages (and other formal languages) is typically represented as recursive types, with one type for each syntactic category. For example, a syntactic category of simple arithmetic expressions might be represented by the following type (expressed in the syntax of the Standard ML programming language):

```
datatype exp = num of int | var of string | plus of exp * exp
             | minus of exp * exp | times of exp * exp
```

*Research supported by the Engineering and Physical Sciences Research Council of Great Britain under grant GR/L14381. The author would like to thank Alan Bundy, Ian Green, Konrad Slind and Christoph Walther for their feedback on this work.

Thus an arithmetic expression is a numeric constant, a variable, or an arithmetic operator applied to two expressions. The type `exp` is recursive and has two base constructors `num` and `var`, and three step constructors `plus`, `minus`, and `times`.

Many languages have syntactic categories that are mutually dependent, e.g., ML has declarations that involve expressions while expressions may themselves contain local declarations. In such situations the types used to represent the abstract syntax are mutually recursive. Another common form has the recursive type appearing as an argument to a type constructor, e.g.:

```
datatype command = Block of (command)list | ...
```

This type represents the abstract syntax of commands. One of the possible forms is a block (a sequence) of commands represented by `(command)list`. Here the recursion of the type is via the `list` type constructor. This form of recursion is often referred to as *nested* recursion.

When the abstract syntax is represented by mutually recursive and/or nested recursive types, functions defined over the abstract syntax are usually mutually recursive. Such functions include compilers, type checkers, etc. Reasoning about recursive functions often requires mathematical induction but special problems arise when the functions are mutually recursive. In particular, the induction hypothesis may involve a different function to the induction conclusion.

In the next section, some of the difficulties of using induction with mutually recursive functions are discussed. This leads to the proposal that induction schemes with more than one induction predicate should be used. Section 3 illustrates the use of such multi-predicate induction schemes by a simple example and in Sect. 4 a proof method for their use is presented. Section 5 describes how the method also applies to mutual recursion over a single type and Sect. 6 outlines how it works on a more difficult example. A procedure for generating multi-predicate schemes from function definitions is described in Sect. 7. Related work is discussed in Sect. 8 and an implementation and results are presented in Sect. 9, with conclusions and future prospects in Sect. 10.

2 Induction for Mutually Recursive Functions

Consider mutually recursive functions *even* and *odd* defined over natural numbers (`suc` is the successor function):

$$\begin{array}{rcl@{\qquad}rcl} even(\mathtt{0}) & = & true & odd(\mathtt{0}) & = & false \\ even(\mathtt{suc}(n)) & = & odd(n) & odd(\mathtt{suc}(n)) & = & even(n) \end{array}$$

Now suppose we wish to prove the following property:

$$\forall n.\ even(n) \supset \neg odd(n)$$

An ordinary natural number induction on n yields a base case (where `0` is substituted for n) that is easily proved. The step case is:

$$even(\texttt{suc}(m)) \supset \neg odd(\texttt{suc}(m))$$

with $even(m) \supset \neg odd(m)$ as the induction hypothesis. Rewriting the induction conclusion using the definitions produces $odd(m) \supset \neg even(m)$ but this does not correspond to the induction hypothesis.

The problem is that the induction has gone only one step through the mutually recursive cycle. A nested induction on the reduced conclusion $odd(m) \supset \neg even(m)$ does not help because the new induction conclusion that it gives rise to, namely $even(m') \supset \neg odd(m')$, involves a different value to the original induction hypothesis (m' instead of m). Using a two-step induction scheme for the original induction does, however, work. The scheme looks like this:

$$\forall P.\ (P(\texttt{0}) \ \wedge\ P(\texttt{suc}(\texttt{0})) \ \wedge\ (\forall n.\ P(n) \supset P(\texttt{suc}(\texttt{suc}(n))))) \ \supset\ \forall n.\ P(n)$$

Another (related) approach is to unwind the mutually recursive functions so that the recursive calls are direct:

$$\begin{array}{rcl@{\qquad}rcl} even(\texttt{0}) & = & true & odd(\texttt{0}) & = & false \\ even(\texttt{suc}(\texttt{0})) & = & false & odd(\texttt{suc}(\texttt{0})) & = & true \\ even(\texttt{suc}(\texttt{suc}(n))) & = & even(n) & odd(\texttt{suc}(\texttt{suc}(n))) & = & odd(n) \end{array}$$

However, for two functions f and g that are each defined in terms of both f and g, unwinding like this is not possible, and in any case, unwinding tends to cause a quadratic increase in the number of equations. There are also the well-known techniques for encoding mutually recursive functions into a single function but in some contexts these approaches might be considered to be cheating because they involve making new definitions. It is preferable to be able to prove properties of existing functions without making new definitions.

The nature of the problem becomes more apparent when considering mutually recursive or nested recursive types, as illustrated by the following types and functions:

```
datatype alpha = z of int | a of beta   and beta = b of alpha
```

$$f(\texttt{z}(n)) = n \qquad f(\texttt{a}(b)) = 1 + g(b) \qquad g(\texttt{b}(a)) = f(a)$$

What does an induction scheme look like for this problem? Here is an incorrect first attempt:

$$\begin{array}{l} \forall P.\ \ ((\forall n : \texttt{int}.\ P(\texttt{z}(n))) \ \wedge\ (\forall b : \texttt{beta}.\ P(b) \supset P(\texttt{a}(b))) \ \wedge \\ \qquad (\forall a : \texttt{alpha}.\ P(a) \supset P(\texttt{b}(a)))) \ \supset \\ \qquad \forall x.\ P(x) \end{array}$$

Some problems with this scheme should be immediately apparent. In a typed setting this scheme is badly typed. The variable P (which will be referred to as an *induction predicate*)

is being applied to two different types, `alpha` and `beta`, and it is not clear what type x has in the conclusion.

One solution is to form the disjoint union type of all the mutually recursive types. The induction predicate can then have this disjoint union type as its argument type. However, to be done properly in a formal framework, the scheme has to include injections into the union type:

$$\begin{array}{l}\forall P.\ ((\forall n.\ P(inl(\mathtt{z}(n)))) \ \wedge\ (\forall b.\ P(inr(b)) \supset P(inl(\mathtt{a}(b)))) \ \wedge \\ \quad (\forall a.\ P(inl(a)) \supset P(inr(\mathtt{b}(a))))) \ \supset \\ \quad \forall (x : \mathtt{alpha} + \mathtt{beta}).\ P(x)\end{array}$$

The presence of the injections makes use of this scheme messy. A much more natural approach is to have more than one induction predicate, in this case one for each mutually recursive type:

$$\begin{array}{l}\forall Q\ R.\ ((\forall n.\ Q(\mathtt{z}(n))) \ \wedge\ (\forall b.\ R(b) \supset Q(\mathtt{a}(b))) \ \wedge\ (\forall a.\ Q(a) \supset R(\mathtt{b}(a)))) \ \supset \\ \quad (\forall a.\ Q(a)) \ \wedge\ (\forall b.\ R(b))\end{array}$$

In fact, Q and R can be seen as the composition of P with *inl* and *inr* respectively. Multi-predicate induction schemes like this are not a new idea. For example, the mutually recursive type definition package of the HOL theorem prover automatically generates structural induction schemes of this form [VG93, Appendix A].

The remainder of this paper considers how induction using multi-predicate schemes can be automated.

3 A Simple Example

The example in this section differs from the examples presented in Sect. 2 in that it involves a nested recursive type. It is a type of arbitrarily branching trees:

```
datatype tree = leaf of int | node of (tree)list
```

A structural induction scheme for the type is:

$$\begin{array}{l}\forall P\ Q.\ ((\forall n.\ P(\mathtt{leaf}(n))) \ \wedge\ (\forall ts.\ Q(ts) \supset P(\mathtt{node}(ts))) \ \wedge \\ \quad Q(\mathtt{nil}) \ \wedge\ (\forall t\ ts.\ P(t) \wedge Q(ts) \supset Q(t\mathtt{::}ts))) \ \supset \\ \quad (\forall t.\ P(t)) \ \wedge\ (\forall ts.\ Q(ts))\end{array}$$

Here the second induction predicate is for the type constructor (`list`) under which the recursive type is nested. The identifier `nil` denotes the empty list and the infix function `::` is the list constructor.

Now consider the following definitions for two functions that construct a list of the leaf nodes of a tree:

$$flatten(\mathtt{nil}) \ = \ \mathtt{nil}$$

$$
\begin{array}{rcl}
flatten(l::ls) & = & app(l, flatten(ls)) \\
\\
flatten\_tree(\texttt{leaf}(n)) & = & n::\texttt{nil} \\
flatten\_tree(\texttt{node}(ts)) & = & flatten(map(flatten\_tree, ts)) \\
\\
fringes(\texttt{nil}) & = & \texttt{nil} \\
fringes(t::ts) & = & app(fringe(t), fringes(ts)) \\
\\
fringe(\texttt{leaf}(n)) & = & n::\texttt{nil} \\
fringe(\texttt{node}(ts)) & = & fringes(ts)
\end{array}
$$

The function *app* appends two lists, and *map* applies a function to every element of a list. They have the usual recursive definitions. The function *map* is second-order and $flatten\_tree$ uses it to avoid mutual recursion. The function $fringe$, on the other hand, has a mutually recursive counterpart, $fringes$, for dealing with the list of subtrees.

The goal is to prove that the two definitions are equivalent, i.e.:

$$\forall t.\ flatten\_tree(t) = fringe(t)$$

The first step is to match the goal with one of the conjuncts of the conclusion of the induction scheme. Type constraints mean it has to be the first conjunct. Thus, the induction predicate Q remains uninstantiated and P is bound to $\lambda t.\ flatten\_tree(t) = fringe(t)$. Beta-reduction yields the following goal:

$$(\forall t.\ flatten\_tree(t) = fringe(t)) \ \wedge \ (\forall ts.\ Q(ts))$$

A proof procedure for induction would now normally attempt to prove all the hypotheses of the instantiated induction scheme. However, since Q has not yet been instantiated only the hypotheses whose consequent involves P should be attempted.

Base Case $P(\texttt{leaf}(n))$. Using the definitions of the functions the first case reduces as follows:

$$
\begin{array}{rcl}
flatten\_tree(\texttt{leaf}(n)) & = & fringe(\texttt{leaf}(n)) \\
n::\texttt{nil} & = & n::\texttt{nil}
\end{array}
$$

Step Case $Q(ts) \supset P(\texttt{node}(ts))$. This case also proceeds by reduction but the two sides of the equation do not become equal (The $\vdash_?$ symbol is used to separate the hypotheses and conclusion of the conjecture. The question mark indicates that we do not yet know that the conjecture is a theorem.):

$$
\begin{array}{lll}
Q(ts) & \vdash_? & flatten\_tree(\texttt{node}(ts)) = fringe(\texttt{node}(ts)) \\
Q(ts) & \vdash_? & flatten(map(flatten\_tree, ts)) = fringes(ts)
\end{array}
$$

At this point one would normally expect to be able to make use of the induction hypothesis. In fact, we do just that, by (second-order) matching the hypothesis to the whole of the residual goal. So, Q is instantiated to:

$$\lambda ts.\ flatten(map(flatten\_tree, ts)) = fringes(ts)$$

Now that Q has been instantiated the remaining induction cases can be attempted.

Base Case Q(`nil`).

$$\begin{aligned} flatten(map(flatten\_tree, \texttt{nil})) &= fringes(\texttt{nil}) \\ flatten(\texttt{nil}) &= \texttt{nil} \\ \texttt{nil} &= \texttt{nil} \end{aligned}$$

Step Case $P(t) \wedge Q(ts) \supset Q(t::ts)$. The goal in this case is:

$$\begin{array}{l} (flatten\_tree(t) = fringe(t)) \wedge \\ (flatten(map(flatten\_tree, ts)) = fringes(ts)) \\ \quad \vdash_? \quad flatten(map(flatten\_tree, t::ts)) = fringes(t::ts) \end{array}$$

The conclusion can be rewritten as follows:

$$\begin{aligned} flatten(map(flatten\_tree, t::ts)) &= fringes(t::ts) \\ flatten(flatten\_tree(t)::map(flatten\_tree, ts)) & \\ &= app(fringe(t), fringes(ts)) \\ app(flatten\_tree(t), flatten(map(flatten\_tree, ts))) & \\ &= app(fringe(t), fringes(ts)) \end{aligned}$$

Then using the implication $(x_1 = x_2) \wedge (y_1 = y_2) \supset app(x_1, y_1) = app(x_2, y_2)$, the goal can be converted to a form in which the induction hypotheses are immediately applicable:

$$flatten\_tree(t) = fringe(t) \quad \wedge \quad flatten(map(flatten\_tree, ts)) = fringes(ts)$$

4 A Proof Method for Using Multi-Predicate Schemes

The example proof in Sect. 3 motivates a proof method `induction_mutual` for multi-predicate induction schemes. The method takes a scheme S, a goal term t, and a matching induction predicate P_k as arguments.

Definition 1 *The scheme S has the general form:*

$$\frac{\begin{array}{c} P_{1,1}(\vec{v}_{1,1}) \ \wedge \ \ldots \ \wedge \ P_{1,n_1}(\vec{v}_{1,n_1}) \ \wedge \ C_1 \ \supset \ P_{1,0}(\vec{f}_1[\vec{v}_{1,1}, \ldots, \vec{v}_{1,n_1}]) \\ \vdots \\ P_{m,1}(\vec{v}_{m,1}) \ \wedge \ \ldots \ \wedge \ P_{m,n_m}(\vec{v}_{m,n_m}) \ \wedge \ C_m \ \supset \ P_{m,0}(\vec{f}_m[\vec{v}_{m,1}, \ldots, \vec{v}_{m,n_m}]) \end{array}}{(\forall \vec{v}_1.\ P_1(\vec{v}_1)) \ \wedge \ \ldots \ \wedge \ (\forall \vec{v}_r.\ P_r(\vec{v}_r))}$$

with the following properties (where $[x]$ denotes the set $\{1, \ldots, x\}$):

1. $m > 0$, $\forall i \in [m].\ n_i \geq 0$, $r > 0$.

2. *The P's are induction predicates.*

3. *The $\vec{v}$'s are vectors of one or more variables.*

4. *The C's are additional conditions.*

5. $\forall i \in [m].\ \vec{f_i}[\vec{v}_{i,1}, \ldots, \vec{v}_{i,n_i}]$ *denotes a vector of terms involving variables in* $\vec{v}_{i,1}, \ldots, \vec{v}_{i,n_i}$.

6. $\forall i \in [r].\ \exists j \in [m].\ P_i = P_{j,0}$.

7. $\forall i \in [m].\ \forall j \in [n_i].\ \exists k \in [m].\ P_{i,j} = P_{k,0}$.

8. *The free variables appearing in* $\{\vec{v}_{i,1}, \ldots, \vec{v}_{i,n_i}, C_i, \vec{f_i}[\vec{v}_{i,1}, \ldots, \vec{v}_{i,n_i}]\}$ *are assumed to be universally quantified in hypothesis* i.

The consequent *of hypothesis i of the scheme is* $P_{i,0}(\vec{f_i}[\vec{v}_{i,1}, \ldots, \vec{v}_{i,n_i}])$ *and the* antecedants *are* $\{P_{i,1}(\vec{v}_{i,1}), \ldots, P_{i,n_i}(\vec{v}_{i,n_i}), C_i\}$. *If h denotes hypothesis i then let $pred(h)$ denote* $P_{i,0}$.

It is not necessary for all the $P_{i,j}$ predicates ($1 \leq i \leq m$, $0 \leq j \leq n_i$) to appear in the conclusion of the scheme (i.e., be equal to one of the P_k ($1 \leq k \leq r$)).

Property 6 of Definition 1 says that each of the induction predicates in the conclusion must appear as the consequent of at least one of the hypotheses. Property 7 says that each of the predicates in the antecedants of the hypotheses must be the consequent of one of the hypotheses.

Definition 2 *A predicate is* native *in the antecedants of a hypothesis if it is equal to the predicate that appears in the consequent. Otherwise it is* foreign.

For the proof method it is useful to distinguish cases of the induction that involve foreign predicates in the antecedants. This motivates the following definition.

Definition 3 *A hypothesis*

$$P_{i,1}(\vec{v}_{i,1}) \ \wedge \ \ldots \ \wedge \ P_{i,n_i}(\vec{v}_{i,n_i}) \ \wedge \ C_i \ \supset \ P_{i,0}(\vec{f_i}[\vec{v}_{i,1}, \ldots, \vec{v}_{i,n_i}])$$

of a scheme is said to be a base case *if $n_i = 0$. Otherwise it is a* step case *if there is a $j \in [n_i]$ such that $P_{i,j} = P_{i,0}$, and a* cycle case *if there is a $j \in [n_i]$ such that $P_{i,j} \neq P_{i,0}$. So, a hypothesis may be both a step case and a cycle case.*

Let us now assume that P_k has been selected as the induction predicate for the goal term t and that variables $\{x_1, \ldots, x_{l_k}\}$ in t have been matched up to $\vec{v}_k$. Without loss of generality, t can be assumed to have the form $\forall x_1 \ldots x_{l_k}\ y_1 \ldots y_u.\ F[x_1, \ldots, x_{l_k}, y_1, \ldots, y_u]$ (universal quantifiers can be commuted). Heuristics for selecting induction variables are described elsewhere in the literature, e.g. [BM79].

The `induction_mutual` procedure is shown in Fig. 1. It is non-deterministic and may fail at various points. The intention is that it should backtrack at points of failure and try alternative execution paths (see below). The submethod `use_hypotheses` assumes that the goal has been rewritten to a conjunction of formulas and tries to match hypotheses to the conjuncts. It goes beyond the procedures typically found in inductive provers in that it can instantiate predicate variables in the hypotheses during the matching process.

Some remarks about the procedure:

- The $\vec{x}$'s and $\vec{y}$'s are vectors of variables.
- The function $frees$ computes the free variables of a term.
- ϕ is a substitution and $(p\phi)_\beta$ denotes the result of applying ϕ to p and beta-reducing (including redexes from previous calls of `use_hypotheses`).
- `match(`p`,`t`)` is true if and only if p can be made syntactically equal to t by instantiating the universal quantifiers in p.
- The submethods `base_case` and `step_case` are as they might be in a method for induction using a single-predicate scheme. Specifically, `base_case` rewrites using the definitions of the functions in the goal and performs standard logical simplifications, while `step_case` manipulates the induction conclusion using the definitions and lemmas to get it into a form in which (some of) the hypotheses can be used. It then uses the hypotheses and simplifies. Both submethods are allowed to leave a residual goal. For simple examples it would suffice for these submethods to do exhaustive rewriting with the function definitions but for more complex examples it is beneficial to use heuristic procedures such as those in *Clam* [BvHHS90].
- The syntactic form '`(base_case then use_hypotheses)(`h`)`' means "first apply the `base_case` submethod to h and then apply the `use_hypotheses` submethod to any residual goals".

The choice points in the procedure are:

1. the selection of an instantiated predicate;
2. within the `base_case` and `step_case` submethods;
3. the selection and ordering of the antecedants Γ' in `use_hypotheses`.

Failure of a conjunction of antecedants (the induction hypotheses) to match the residual term causes backtracking which drives a search through the different combinations and permutations of antecedants at choice point 3. The search terminates if a match is found. If no combination of antecedants produces a match then `use_hypotheses` fails. This may cause backtracking at point 2 to yield a different residual term. If that is not successful the induction predicates may be processed in a different order (point 1) but this is unlikely to

```
procedure induction_mutual(S,t,P_k);

   procedure matches(p,t);
      p_1 ∧ ... ∧ p_u := p;   t_1 ∧ ... ∧ t_u := t;
      I := {i ∈ [u] | p_i has the form P_i(x⃗_i)};
      for i ∈ I do begin y⃗_i := frees(t_i) − x⃗_i; Λ_i := λx⃗_i. ∀y⃗_i. t_i end;
      φ := {Λ_i/P_i | i ∈ I};
      if ∀i ∈ [u]. match((p_i φ)_β,t_i) then
         begin
            for i ∈ I do instantiate P_i to Λ_i;
            return true
         end
      else return false
   end procedure;

   procedure use_hypotheses(Γ ⊃ t);
      if (∃Γ' ⊆ Γ. (Γ' = {a_1,...,a_u}) ∧ u > 0 ∧ matches(a_1 ∧ ... ∧ a_u,t))
      then return
      else fail
   end procedure;

   H := the hypotheses of S;
   instantiate P_k to λx_1 ... x_{l_k}. ∀y_1 ... y_u. F[x_1,...,x_{l_k},y_1,...,y_u];
   while H ≠ {} do
      begin
         P := any pred(h) (for h ∈ H) that has been instantiated;
         Cases := {h ∈ H | pred(h) = P};
         beta-reduce applications of instantiated induction predicates in Cases;
         H := H − Cases;
         Base := {h ∈ Cases | h is a base case};
         Cycle := {h ∈ Cases | h is a cycle case and not a step case};
         Step := {h ∈ Cases | h is a step case and not a cycle case};
         StepAndCycle := {h ∈ Cases | h is both a step case and a cycle case};
         for h ∈ Base do base_case(h);
         for h ∈ Cycle do (base_case then use_hypotheses)(h);
         for h ∈ Step do step_case(h);
         for h ∈ StepAndCycle do (step_case then use_hypotheses)(h)
      end

end procedure
```

Figure 1: The induction method for multi-predicate induction schemes

help the proof because the cases for the originally chosen predicate must be processed eventually. Further backtracking might cause earlier instantiations of predicates to be undone and a different choice to be made at point 3 for an earlier invocation of `use_hypotheses`, but undoing instantiations may be difficult to implement.

Definition 4 *The* dependency graph $G(S)$ *of the scheme* S *is the graph with vertices* $\{P_{i,0} \mid i \in [m]\}$ *and edges* $\{(P_{i,0}, P_{i,j}) \mid i \in [m] \ \wedge \ j \in [n_i]\}$.

Lemma 1 *If* $G(S)$ *is strongly connected (any vertex can be reached from any other vertex) and the* `base_case` *and* `step_case` *submethods cannot complete a cycle case or step case without using the induction hypotheses, then on each entry to the body of the* `while` *loop in the* `induction_mutual` *procedure, there exists some* $h \in H$ *such that* $pred(h)$ *has been instantiated.*

Proof. On the first time in the body of the loop the result follows because P_k has been explicitly instantiated in the code prior to the loop. On subsequent iterations the theorem relies on the instantiation of predicates in the `use_hypotheses` submethod causing some $pred(h)$ (for $h \in H$) to become instantiated.

First, observe that if P is the current predicate being processed and there is an edge (P, Q) in $G(S)$ then Q will become instantiated on this iteration of the loop. From the presence of the edge it follows that Q must be an antecedant in one of the cycle or step cases. Under the assumption that `base_case` and `step_case` cannot complete a cycle case or step case without using the induction hypotheses, `use_hypotheses` must be called on at least one goal involving Q as a hypothesis and the Γ' chosen will also involve Q. Hence if Q is not already instantiated it will become so in `matches`.

So, for there to be hypotheses remaining but no instantiated predicate, the remaining uninstantiated predicates could not have been reachable from any of the previously processed predicates. But this contradicts the assumption that $G(S)$ is strongly connected.

Theorem 2 (Termination) *If* $G(S)$ *is strongly connected and if the* `base_case` *and* `step_case` *submethods terminate and cannot complete an inductive case without using the induction hypotheses, the* `induction_mutual` *procedure terminates (or fails).*

Proof. From Lemma 1, an instantiated predicate P exists and so $Cases$ is non-empty. Hence the cardinality of H strictly decreases each time round the `while` loop and so the loop condition will eventually become false.

If the procedure is to be used on a scheme and goal for which the conditions of Theorem 2 are not satisfied, the procedure can easily be modified to check that H has strictly decreased between successive iterations of the `while` loop and fail if it has not. Of course, this will only make the whole procedure terminating if the submethods are terminating.

5 Mutual Recursion Over a Single Type

The `induction_mutual` method presented in Sect. 4 is not restricted to mutually recursive functions where each function is defined over a different type. To see this, consider again the *even*/*odd* example from Sect. 2. We use the following multi-predicate induction scheme:

$$\begin{array}{l}\forall P\ Q.\ \ (P(\mathtt{0})\ \wedge\ (\forall n.\ Q(n) \supset P(\mathtt{suc}(n))) \wedge \\ \qquad\quad Q(\mathtt{0})\ \wedge\ (\forall n.\ P(n) \supset Q(\mathtt{suc}(n))))\ \supset \\ \qquad\quad (\forall n.\ P(n))\ \wedge\ (\forall n.\ Q(n))\end{array}$$

The proof of the property $\forall n.\ even(n) \supset \neg odd(n)$ proceeds as follows.

Base Case $P(\mathtt{0})$. This case proceeds as usual by symbolic evaluation:

$$\begin{array}{c} even(\mathtt{0}) \supset \neg odd(\mathtt{0}) \\ true \supset \neg false \\ true \end{array}$$

Cycle Case $\forall n.\ Q(n) \supset P(\mathtt{suc}(n))$.

$$\begin{array}{lll} Q(n) & \vdash_? & even(\mathtt{suc}(n)) \supset \neg odd(\mathtt{suc}(n)) \\ Q(n) & \vdash_? & odd(n) \supset \neg even(n) \end{array}$$

Instantiate Q to $\lambda n.\ odd(n) \supset \neg even(n)$.

Base Case $Q(\mathtt{0})$. By symbolic evaluation.

Cycle Case $\forall n.\ P(n) \supset Q(\mathtt{suc}(n))$.

$$\begin{array}{lll} even(n) \supset \neg odd(n) & \vdash_? & odd(\mathtt{suc}(n)) \supset \neg even(\mathtt{suc}(n)) \\ even(n) \supset \neg odd(n) & \vdash_? & even(n) \supset \neg odd(n) \end{array}$$

Using the multi-predicate induction scheme, the example behaves as if *even* and *odd* were defined over distinct (mutually recursive) types:

```
datatype num_e = 0_e | suc_e of num_o   and num_o = 0_o | suc_o of num_e
```

Viewing the example in this way is natural, for if it were not appropriate to make some distinction between the entities over which the mutually recursive functions are defined, then mutually recursive functions would not have been used. A single function would have been used instead.

6 A More Complex Example

We now consider an example presented by Kapur and Subramaniam [KS96] and credited by them to Bidoit and Garland. It concerns the equivalence of two forms of expression evaluation. The definitions of the functions involved are as follows (where the free variables are implicitly universally quantified):

$$
\begin{array}{rcl}
exp(\mathtt{nat}(x), y, z) & = & \mathtt{nat}(x) \\
(x = y) \supset exp(\mathtt{id}(x), y, z) & = & exhelp(z) \\
\neg(x = y) \supset exp(\mathtt{id}(x), y, z) & = & \mathtt{id}(x) \\
exp(\mathtt{plus}(x, y), z, w) & = & \mathtt{plus}(exp(x, z, w), exp(y, z, w)) \\
exp(\mathtt{apply}(x, y, z), z_1, w) & = & exp(exp(x, y, z), z_1, w) \\
 & & \\
exhelp(\mathtt{nat}(x)) & = & \mathtt{nat}(x) \\
exhelp(\mathtt{id}(x)) & = & \mathtt{id}(x) \\
exhelp(\mathtt{plus}(x, y)) & = & \mathtt{plus}(exhelp(x), exhelp(y)) \\
exhelp(\mathtt{apply}(x, y, z)) & = & exp(x, y, z) \\
 & & \\
lookup(x, \mathtt{start}) & = & \mathtt{0} \\
lookup(x, \mathtt{update}(y, z, w)) & = & cond((x = y), z, lookup(x, w)) \\
 & & \\
eval(\mathtt{nat}(x), w) & = & x \\
eval(\mathtt{id}(x), w) & = & lookup(x, w) \\
eval(\mathtt{plus}(x, y), w) & = & eval(x, w) + eval(y, w) \\
eval(\mathtt{apply}(x, y, z), w) & = & eval(x, \mathtt{update}(y, eval(z, w), w))
\end{array}
$$

For an explanation see Kapur and Subramaniam's paper. We use the following multi-predicate induction scheme:

$$
\begin{array}{l}
\forall P\ Q.\ ((\forall x\ z.\ P(\mathtt{nat}(x), z))\ \wedge\ (\forall x\ z.\ Q(z) \supset P(\mathtt{id}(x), z)) \wedge \\
\quad (\forall x\ y\ w.\ P(x, w) \wedge P(y, w) \supset P(\mathtt{plus}(x, y), w)) \wedge \\
\quad (\forall x\ y\ z\ w.\ P(exp(x, y, z), w) \wedge P(x, z) \supset P(\mathtt{apply}(x, y, z), w)) \wedge \\
\quad (\forall x.\ Q(\mathtt{nat}(x)))\ \wedge\ (\forall x.\ Q(\mathtt{id}(x))) \wedge \\
\quad (\forall x\ y.\ Q(x) \wedge Q(y) \supset Q(\mathtt{plus}(x, y))) \wedge \\
\quad (\forall x\ y\ z.\ P(x, z) \supset Q(\mathtt{apply}(x, y, z))))\ \supset \\
\quad (\forall x\ z.\ P(x, z))\ \wedge\ (\forall z.\ Q(z))
\end{array}
$$

This scheme is interesting because of the nested recursion in the *exp* function as well as for having two induction predicates. The nested recursion gives rise to the $P(exp(x, y, z), w)$ term in the antecedant of the induction case for $P(\mathtt{apply}(x, y, z), w)$. For a discussion of why this antecedant is present and how the scheme can be shown to be sound, see the work of Slind [Sli97, Sect. 4.3]. The scheme can be generated automatically from the definitions as described in Sect. 7.

The conjecture to be proved is:

$$\forall u\ v\ t\ s.\ eval(exp(u, v, t), s) = eval(u, \texttt{update}(v, eval(t, s), s))$$

The proof uses the above induction scheme with P matched to the conjecture and u and t matched to its arguments (x and z respectively). Using the definitions of the functions the $P(\texttt{nat}(x), z)$ case is almost trivial. The $Q(z) \supset P(\texttt{id}(x), z)$ case is the most important because it derives a value for Q:

$$\begin{array}{lll} Q(z) & \vdash_? & \forall v\ s.\ eval(exp(\texttt{id}(x), v, z), s) = eval(\texttt{id}(x), \texttt{update}(v, eval(z, s), s)) \\ Q(z) & \vdash_? & \forall v\ s.\ eval(exp(\texttt{id}(x), v, z), s) = lookup(x, \texttt{update}(v, eval(z, s), s)) \\ Q(z) & \vdash_? & \forall v\ s.\ eval(exp(\texttt{id}(x), v, z), s) = cond((x = v), eval(z, s), lookup(x, s)) \end{array}$$

The conditional rules for $exp(\texttt{id}(x), v, z)$ motivate a case-split. In the case $x = v$ the goal becomes:

$$Q(z) \quad \vdash_? \quad \forall s.\ eval(exhelp(z), s) = eval(z, s)$$

Since no further rewriting is possible, the `induction_mutual` method would at this point attempt to instantiate the hypothesis in order to satisfy the goal. Thus, Q becomes instantiated to $\lambda z.\ \forall s.\ eval(exhelp(z), s) = eval(z, s)$. In the $\neg(x = v)$ case the conclusion reduces to $\forall s.\ lookup(x, s) = lookup(x, s)$ which is true by reflexivity. The induction hypothesis is not required. The fact that only one case requires the hypothesis is rather convenient and in general would not be so. In general, for n cases, new variables $Q_1, \ldots, Q_n$ would have to be introduced in place of Q and then Q would be instantiated to a term formed from the instances of the Q_i's (ignoring any uninstantiated ones).

The fourth case is also interesting because of the nested recursion but since that is not the subject of this paper the details are omitted. In fact, on paper the proof is straightforward but is unusual in that the induction hypotheses have to be used in sequence; they cannot be used simultaneously. All the other cases are straightforward.

7 Generating Multi-Predicate Schemes

It can be seen from the examples in Sects. 5 and 6 that the induction scheme should not simply follow the recursion of the type(s) but rather it should follow the recursion of the mutually recursive functions involved in the conjecture. There should be one induction predicate for each of the mutually recursive functions. An outline of a procedure to generate such a scheme from constructor-style function definitions follows.

1. Find the functions $f_1, \ldots, f_n$ defined by the mutually recursive definition.
2. Form corresponding induction predicates (variables) $P_1, \ldots, P_n$ such that for f_i of type $t_1 \times \ldots \times t_k \to t$, P_i has type $t_1 \times \ldots \times t_k \to \texttt{bool}$.

3. For each clause in the definition compute a case for the induction scheme. Assuming a clause has the general form $\forall v_1 \ldots v_m.\ \ldots \supset (l = r)$, the conclusion c of the induction case is formed from l by replacing the f at its head with the corresponding P. Then find every application of one of the f's in r, including those nested inside arguments of other applications. These applications (with the f replaced by the corresponding P) form the hypotheses $\{h_1, \ldots, h_j\}$ of the induction case. If $j = 0$ the induction case is $\forall v_1 \ldots v_m.\ c$, otherwise it is $\forall v_1 \ldots v_m.\ h_1 \wedge \ldots \wedge h_j \supset c$.

4. Some of the induction cases generated in Step 3 may have the same conclusion (up to renaming of the v's). Such cases should be merged by renaming the v's and merging the hypotheses.

5. For each P_i form a term $\forall x_1 \ldots x_k.\ P_i(x_1, \ldots, x_k)$ where the x's have unique names and the types of P_i's arguments.

6. Form a conjunction H of the cases generated in Step 4 and another conjunction C of the terms created in Step 5. The induction scheme is then $\forall P_1 \ldots P_n.\ H \supset C$.

For the example in Sect. 6 this algorithm will generate a scheme in which P has three arguments rather than two. A more sophisticated algorithm could eliminate the extra argument because it plays no role in the induction. Its presence does not interfere with the proof. See also Walther's work on generating induction schemes [Wal92].

It is possible to *formally* derive a multi-predicate induction scheme from the definitions of mutually recursive functions, as Slind shows in his PhD thesis [Sli99]. The soundness of the induction scheme follows from the termination of the functions. For an approach to showing termination, see for example the work of Giesl [Gie97].

8 Related Work

The method presented here for induction using multi-predicate schemes is in some respects similar to so-called *middle-out reasoning* [Hes91]. The proof of the original conjecture proceeds simultaneously with finding suitable instances for the induction predicates, with each assisting the other. The formulas used to instantiate the predicates can be seen as intermediate lemmas or as extra conjuncts that generalise the original conjecture. Like middle-out reasoning, the method uses meta-variables to stand for some initially unknown term structure.

Kapur and Subramaniam describe a technique for automating induction over mutually recursive functions using cover sets [KS96]. Their approach is to unroll the mutual recursion to obtain a cover set that captures the recursive dependencies. For example, for mutually recursive functions f and g, recursive calls to g in the body of f are expanded using the rules for g so that calls to f appear in the body instead. If g has no recursive calls to itself then it is eliminated completely. It is not clear from Kapur and Subramaniam's paper how their algorithm generalises to more than two mutually recursive functions or that it

works in general for two functions that are each defined in terms of both functions. The multi-predicate approach can handle both of these situations.

Kapur and Subramaniam discuss the relationship of their work to handling of mutual recursion in Nqthm and Spike. They discuss the need for the user to provide hints or intermediate lemmas in both systems. In our work, these intermediate lemmas are precisely the formulas to which the induction predicates become instantiated, i.e., they are generated automatically. The intermediate lemma required by Spike for the example described in Sect. 6 is precisely the formula to which the Q induction predicate is instantiated. Thus, the `induction_mutual` method is computing lemmas that had to be supplied manually in Nqthm and Spike. In Kapur and Subramaniam's approach these lemmas are bypassed by unrolling the recursion. It might be argued that extra information is being supplied manually in my approach via the more exotic induction scheme but, as has been shown in Sect. 7, the scheme can be generated automatically from the function definitions.

Liu and Chang [LC87] deal with mutual recursion by generating strong induction schemes, i.e., ones in which the property is assumed to hold for a chain of smaller values rather than just the next smallest. The hypotheses for the smaller values are captured by defining auxiliary recursive functions. These functions follow the recursion pattern of the mutually recursive functions about which the property is to be proved but with predicates in place of the original expressions. For example, the following function would be generated from the $fringes$ function of Sect. 3:

$$f(\texttt{nil}) \;=\; true \qquad f(t::ts) \;=\; P(t) \;\wedge\; f(ts)$$

where P is the induction predicate. By this means, all the necessary hypotheses for the proof are obtained. Liu and Chang present an algorithm for generating the auxiliary functions. It is similar to the procedure described in Sect. 7. Their paper is mainly about generating the schemes, saying little about how they are used in proofs, and is in an untyped destructor-style setting, but their work appears to transfer to typed and constructor-style settings. Finally, they claim that strong induction hypotheses should be used for mutual recursion but, as has been shown in this paper, multi-predicate weak schemes also work.

9 Implementation and Results

The `induction_mutual` method of Sect. 4 has been implemented in the *Clam* proof planner [BvHHS90]. For step cases (using the terminology of Definition 3) the rippling method [BSvH$^+$93] is used to guide the proof. Rippling takes place with respect to the hypotheses that correspond to the native induction predicate (Definition 2). For non-step cases symbolic evaluation and simplification are used. Currently, the instantiation of induction predicates works for cases in which either symbolic evaluation or "rippling out" are used. Another form of rippling, called "rippling in", guides the proof to a point at which a universally quantified variable in the hypothesis can be instantiated in a non-trivial way. This form of rippling is typically required where one of the functions has an "accumulator" argument. It is not yet catered for by the `induction_mutual` method.

Table 1: Results for *Clam* implementation

	Conjecture	Source
1	$\forall n.\ even(n) \supset \neg odd(n)$	Sect. 5
2	$\forall n.\ even(n) \vee even(suc(n))$	
3	$\forall t.\ flatten\_tree(t) = fringe(t)$	Sect. 3
4	$\forall t.\ flatten\_tree(reverse\_tree(t)) = reverse(flatten\_tree(t))$	
5	$\forall x\ y.\ fringes(app(x, y)) = app(fringes(x), fringes(y))$	
6	$\forall t.\ fringe(reverse\_tree(t)) = reverse(fringe(t))$	
7	$\forall s.\ s = foo(s)$	Reference [LC87]

It is important that the goal is fully reduced before the `induction_mutual` method is applied. If not, there is a tendency for the induction hypotheses and conclusion to mismatch. For example, the *even*/*odd* conjecture in Sects. 2 and 5 could have been expressed as:

$$\forall n.\ even(n) \supset \neg even(\texttt{suc}(n))$$

If this is not reduced initially, then in the final case the goal is:

$$even(n) \supset \neg even(\texttt{suc}(n)) \quad \vdash_? \quad odd(\texttt{suc}(n)) \supset \neg even(\texttt{suc}(n))$$

This requires delicate control over rewriting if the hypothesis is to be used successfully, but if the goal is fully reduced before applying induction this problem does not arise; the definitions of the functions can be used freely.

The procedure for generating schemes in Sect. 7 has been implemented in an object-level theorem prover, from where the scheme is passed to the (meta-level) proof planner automatically.

The `induction_mutual` method was developed using the examples in Sects. 3 and 5. The implementation in *Clam* has been used to automatically plan proofs for both of the development examples and the other formulas listed in Table 1.

The function *reverse_tree* reverses the order of the leaf nodes and is defined in a similar way to *flatten_tree* using *map*. Since *reverse_tree* is not mutually recursive, Example 4 does not involve mutually recursive functions but a multi-predicate scheme is still used because the `tree` type is nested recursive. The proof requires lemmas for the distributivity of *map*, *reverse*, and *flatten* over *app*, and the lemma $app(x, \texttt{nil}) = x$. In addition, Example 5 requires the associativity of *app*, and Example 6 requires Example 5 as a lemma. The function *foo*, taken from [LC87], manipulates Lisp-like expressions and is mutually recursive with a function *foolist*.

The example in Sect. 6 has not yet been fully automated due to the nested recursive function call. This kind of nested recursion is currently outside of the scope of *Clam*. However, the proof method does get as far as synthesizing the instantiation for the second induction predicate, and all but one (the fourth) of the cases of the induction have been automated.

Since the proof method maintains second-order variables (the uninstantiated induction predicates) for part of the proof, the implementation language of *Clam*, namely Prolog, is not ideally suited. The second-order variables have to be coded as first-order variables and beta-reduction must be handled explicitly. A more appropriate platform for the method would be *λClam* [RSG98] which is implemented in the higher-order logic programming language λProlog. *λClam* was not used because, when implementation began, some aspects of it were not sufficiently mature to support this research.

10 Conclusions and Future Work

This paper has shown how induction theorems (schemes) with multiple induction predicates can be used to prove properties of mutually recursive functions in a natural way. A proof method for using such schemes has been proposed and illustrated on several examples. The method has been implemented and tested in the *Clam* proof planner. The implemented method makes use of *Clam*'s infrastructure but it is essentially independent of the details of *Clam*, and hence could be re-used in other systems.

The induction schemes have one predicate for each of the mutually recursive functions. This avoids the need for techniques such as unwinding the functions into a single function (which tends to cause a quadratic increase in size and is not always possible anyway). One induction predicate is matched against the initial goal and the instantiations for the other predicates are synthesized as part of the proof method. An important point is that the induction scheme follows the recursion pattern of the functions rather than of the types over which they are defined (though these do coincide in many cases).

A strength of the method is that it is able to speculate intermediate lemmas that in some other approaches have to be provided by the user. This speculation of lemmas corresponds to synthesizing values for the initially uninstantiated induction predicates. As can be seen from the examples, this is often quite straightforward. However, there are examples where in order to get into a position in which a predicate can be instantiated, an implicative (i.e., non-equality preserving) step has to be used. Such a step generalises the goal and can result in the predicate being instantiated to a formula that is not a theorem. This seems to be a particular problem when the case involves more than one occurrence of the predicate in the antecedants, e.g.:

$$\forall x\ y.\ Q(x) \wedge Q(y) \supset P(\mathtt{C}(x, y))$$

A possible solution might be to only partially instantiate Q when the hypotheses are used (by introducing new higher-order meta-variables to stand for unknown term structure) and use the base cases to fill in the gaps. This would require some modification to the proof method and would be quite intricate unless higher-order unification was built in to the implementation language.

Another item for future work is to investigate examples where there are more than two induction predicates. In such examples, there may be cases of the form

$$\forall x\ y.\ Q(x) \wedge R(y) \supset P(\mathtt{C}(x, y))$$

where it may not be clear how the two predicates Q and R should be instantiated. Also, if there is more than one mutually recursive function involved in a conjecture it may be necessary to combine induction schemes as is done, for example, by Boyer and Moore [BM79], Liu and Chang [LC87], and by Walther [Wal93].

References

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

[BSvH+93] 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.

[BvHHS90] 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*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 647–648, Kaiserslautern, FRG, July 1990. Springer-Verlag.

[Gie97] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–29, August 1997.

[Hes91] J. T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving*. PhD thesis, University of Edinburgh, 1991.

[KS96] D. Kapur and M. Subramaniam. Automating induction over mutually recursive functions. In M. Wirsing and M. Nivat, editors, *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, Munich, Germany, July 1996. Springer-Verlag.

[LC87] P. Liu and R.-J. Chang. A new structural induction scheme for proving properties of mutually recursive concepts. In *Proceedings of the 6th National Conference on Artificial Intelligence*, volume 1, pages 144–148. AAAI, July 1987.

[RSG98] J. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with $\lambda Clam$. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 129–133, Lindau, Germany, July 1998. Springer-Verlag.

[Sli97] K. Slind. Derivation and use of induction schemes in higher-order logic. In E. L. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 275–290, Murray Hill, NJ, USA, August 1997. Springer-Verlag.

[Sli99] K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Institut für Informatik, Technische Universität München, 1999. Accessible at `http://www.cl.cam.ac.uk/users/kxs/papers/phd.ps.gz`.

[VG93] M. VanInwegen and E. Gunter. HOL-ML. In J. J. Joyce and C.-J. H. Seger, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, pages 61–74, Vancouver, B.C., Canada, August 1993. Springer-Verlag, 1994.

[Wal92] C. Walther. Computing induction axioms. In A. Voronkov, editor, *Proceedings of LPAR'92 (Logic Programming and Automated Reasoning)*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 381–392, St Petersburg, Russia, July 1992. Springer-Verlag.

[Wal93] C. Walther. Combining induction axioms by machine. In R. Bajcsy, editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, volume 1, pages 95–100, Chambéry, France, August/September 1993. Morgan Kaufmann Publishers, Inc.