Well-founded Induction and Program Synthesis Using Proof Plans

Colin Phillips*

University of Edinburgh

June 1991

Abstract

The Mathematical Reasoning Group (MRG) in the Department of Artificial Intelligence at Edinburgh University has developed an approach to the verification and synthesis of programs using proof plans - high level descriptions of proofs, which control the automatic search for a proof. Constructive mathematics is employed, enabling programs to be derived using the proofs as programs technique. The proofs in question employ various forms of induction and techniques have been developed for automating the search for the form of induction appropriate to a given proof.

In this paper I will discuss a modification of this methodology which involves using well-founded induction and the automated search for the appropriate well-founded relation. This is being implemented in a version of Martin-Löf's type theory extended by Nordström's Acc Type. I shall be comparing this approach with that previously employed in the Mathematical Reasoning Group.

1 Introduction

The Mathematical Reasoning Group (MRG) in the Department of Artificial Intelligence at Edinburgh University has developed an approach to the automatic construction of proofs by induction using proof plans. A plan has two constituents, a tactic and a method. A tactic is a computer program for applying rules of inference. A method is meta-logical specification of a tactic. Two key proof plans are those for *rippling* and for *middle-out reasoning*. The rippling plan involves the rewriting of the induction conclusion to make it resemble the induction hypothesis. The existential plan uses middle-out reasoning to postpone the selection of an existential witness. Using these ideas techniques have been developed to infer the form of induction that is needed to prove a given theorem.

In this paper I will discuss a modification of this methodology which involves using well-founded induction and the automated search for the appropriate well-founded relation. This has two advantages over existing techniques.

Firstly, at present Clam - the proof planner - simply adds schemes of induction to its repertoire as required while the modification I propose uses only a single schema, well-founded induction, thus obtaining greater generality.

Secondly, the MRG is concerned not only with the automated construction of proofs but also with the synthesis of programs using the proofs as programs technique. As a proof is constructed in Oyster an extract term is built up which contains the computational information in the proof. Unfortunately the extract term may contain computationally redundant information. If we use the Acc Type then the

*colinp@aipna.ed.ac.uk. This research was partly supported by ESPRIT Basic Research Action "Logical Framework" grant 3245.

extract term will not contain such information and will be the program we require. The reason for this is that a proof using that Acc type will consist of two parts; a proof that some relation is well-founded over some set and a proof of a goal using this as a lemma. The final extract term however will only contain computational information relating to the latter proof.

However Clam is a highly successful theorem proving system, so our first concern is to insure that the attaining of these benefits still allows us to employ existing methods - in particular rippling and middle-out reasoning. In fact there already exists a mechanism in Clam which can be used to enable rippling to occur with well-founded induction - rippling-across - which I will describe. I will also outline a form of middle-out-reasoning suitable for use with well-founded induction.

2 Proof Plans and Middle-Out Reasoning

The background to this paper is the work of the MRG at Edinburgh. I shall therefore begin by giving an outline of that work. For further information see [1].

The MRG at Edinburgh University is involved in program verification and synthesis by theorem proving. It employs an implementation of Martin-Lö f's type theory, the Oyster proof development system [4] which is essentially a Prolog version of NUPRL [3]. It is used in conjunction with Clam - a proof planner.

Clam was designed as a mechanism for automatically deriving proofs in Oyster. The emphasis has been on proofs by induction where the key problem is to identify the appropriate scheme of induction and the induction variable. The basic idea of a plan is that it seeks to capture a common structure identifiable in a set of proofs. A plan is a tree of methods. A method is a tactic and a meta-logical specification of that tactic.

2.1 Associativity of Plus

Consider, as an example, the proof of the associativity of plus. Putting aside the base case we have:

$$x + (y + z) = (x + y) + z$$

$$\begin{array}{lrcl} \vdash & s(x) + (y + z) & = & (s(x) + y) + z \\ \vdash & s(x + (y + z)) & = & s(x + y) + z \\ \vdash & s(x + (y + z)) & = & s((x + y) + z) \\ \vdash & x + (y + z) & = & (x + y) + z \end{array}$$

At each step an instance of the successor function is moved out from an initial position to one less nested in the equation. When both instances of the successor function are in the outermost position they can be deleted leaving a copy of the induction conclusion.

Each step can be justified by an appeal to a rewrite rule. Thus we use:

$$\begin{array}{rcl} \boxed{s(\underline{U})} + V & \Rightarrow & \boxed{s(\underline{U + V})} \\ \boxed{s(\underline{U})} = \boxed{s(\underline{V})} & \Rightarrow & U = V \end{array}$$

These rules are called wave rules. The process of successively rewriting the equation using such rules is called rippling. The final appeal to the induction hypothesis is called fertilization. The boxes designate wave fronts. The part of the argument inherited from the induction hypothesis is underlined. Rippling is a mechanism for overcoming the mismatch between induction hypothesis and conclusion to enable fertilization to take place.

So we can annotate our example as follows:

$$
\begin{array}{rrcl}
 & \multicolumn{3}{l}{x + (y + z) = (x + y) + z} \\
\vdash & \boxed{s(\underline{x})} + (y + z) & = & (\boxed{s(\underline{x})} + y) + z \\
\vdash & \boxed{s(\underline{x + (y + z)})} & = & \boxed{s(\underline{x + y})} + z \\
\vdash & \boxed{s(\underline{x + (y + z)})} & = & \boxed{s(\underline{(x + y) + z})} \\
\vdash & x + (y + z) & = & (x + y) + z
\end{array}
$$

The above wave rules we have used are of the form:

$$
\begin{array}{rcl}
F(\boxed{S(\underline{U})}) & \Rightarrow & \boxed{S(\underline{F(U)})} \\
F(\boxed{S(\underline{U_1})}, \boxed{S(\underline{U_2})}) & \Rightarrow & F(U_1, U_2)
\end{array}
$$

These are instances of a general form. Such rules rewrite terms so as to produce a closer match between the goal and the induction hypothesis.

Rippling and fertilization are implemented by rippling and fertilization methods. Clam operates on a meta-level representation of the proof sequent, and uses meta-level reasoning on the methods to select an appropriate combination of tactics.

2.2 Prime Factorisation Theorem

The meta-level representation of the proof can contain meta-variables which act as place holders for expressions whose identity has yet to be determined. Consider an example, the prime factorisation theorem. Suppose we want Clam to synthesise an appropriate induction scheme for this proof. The specification is:

$$\forall x{:}nat^{+}.\ \exists xl{:}list(prime).\ prod(xl) = x$$

For convenience I shall use familiar notation and not that of type theory or Oyster.

Setting aside the base case we can represent the step case with a metavariable.

$$
\begin{array}{c}
x' : nat^{+}, \\
\exists xl' : list(prime)\ prod(xl') = x' \\
\vdash \exists \boxed{xl} : list(prime) prod(\boxed{xl}) = \boxed{X}
\end{array}
$$

We use capitals Prolog fashion to represent meta-variables.

The meta-variable corresponds to a universally quantified variable and its instantiation will determine the form of induction needed. We annotate to indicate potential wave fronts.

We assume the following wave rules:

$$
\begin{array}{rcl}
(\boxed{HD :: \underline{TL}}){:}list(type) & \Rightarrow & \boxed{HD{:}type \wedge \underline{TL{:}list(type)}} \\
prod(\boxed{HD :: \underline{TL}}) & \Rightarrow & \boxed{HD \times \underline{prod(TL)}} \\
\boxed{U \times \underline{V}} = \boxed{U \times \underline{W}} & \Rightarrow & V = W
\end{array}
$$

We can use them in sequence to ripple out thus:

$$
\begin{aligned}
\ldots \quad \vdash \quad & \boxed{P : prime \wedge \underline{XL' {:} list(prime)}} \wedge prod(\boxed{P :: \underline{XL'}}) = X \\
\ldots \quad \vdash \quad & \boxed{P : prime \wedge \underline{XL' {:} list(prime)}} \wedge \boxed{P \times \underline{prod(XL')}} = X \\
\ldots \quad \vdash \quad & \boxed{P : prime \wedge \underline{XL' {:} list(prime)}} \wedge prod(XL') = X'
\end{aligned}
$$

where $P :: XL'/XL$ and $P \times X'/X$.

Fertilization succeeds with instantiations: xl'/XL' and x'/X' leaving subgoal $P : prime$. So XL is $P :: xl'$ and X is $P \times x'$.

The induction scheme suggested is:

$$
\frac{\vdash\ P(1) \qquad p{:}prime,\ x'{:}nat,\ P(x') \vdash P(p \times x')}{\vdash \forall x{:}nat.\ P(x)}
$$

The above is an example of *middle-out reasoning*. The use of meta-variables at the start of a proof enables us to look ahead and see how the rippling will precede and hence to determine the appropriate induction scheme. For further details see [1].

3 Computational Redundancy

The remainder of this paper will deal with extensions to the above work resulting from an implementation in Oyster of well-founded induction. Such an extension seems natural given the program of synthesising induction schemes. At present induction schemes are added to Clam, as they are shown to be required for the solution of a problem by middle-out reasoning. But why not simply have a single form of induction - well-founded induction - and use middle-out reasoning to determine the relation and the set needed for a particular example.

The precise choice of an implementation was guided however by another consideration. Clam is built on top of Oyster which is an implementation of constructive mathematics. As a consequence of this it can be used not only to automate the construction of proofs but also to synthesise programs. This was in fact a motive underlying the use of constructive mathematics. In Martin-Löf's system the notion of a proof is ambiguous - we can speak of a proof as a method for making the truth of a proposition evident or we can think of a proof as a proof object or program containing the computational content of the proofs (in the first sense) of the corresponding propositions. In type theory we can think of true propositions as types inhabited by appropriate proof-objects. In Oyster - as in NUPRL - proof objects are called extract terms and are automatically constructed with the corresponding proof.

One way of bringing out the connection between proofs and programs is to observe that in Martin-Löf's type theory the axiom of choice is provable. A version of the axiom of choice is as follows:

$$
\begin{aligned}
&(\forall n : nat)(\exists m : nat)(R(n, m)) \rightarrow \\
&\qquad (\exists f : nat \rightarrow nat)(\forall n : nat)(R(n, f(n)))
\end{aligned}
$$

One can observe that the antecedent has the general form of a program specification. A specification states that for any input there is an output related to the input in the manner specified. The consequent of the axiom states that there is a function which applied to the input gives the output. We can think of the function as the program we want. As we are using a constructive logic an existential claim must be backed by a witness. If we then prove the consequent from the antecedent then the term inhabiting the conclusion would be that program. In fact however the extract term is a pair, the first component of which is the program we want and the second component is the extract term of a proof that this program satisfies the specification. This second component is, from the programmers view, of no interest.

In practice it is the introduction rule for the existential quantifier that causes the trouble. $\exists x : A.F(x)$ has as its proof object a pair - the first is an object having the property F, while the second is a proof that it has that property. In the case of the axiom of choice only the first object is of interest to us. One way of dealing with this problem is to introduce into the theory sets which are not fully presented

- subsets. These are in fact available in both Oyster and Clam. We write $\{x : A|F(x)\}$ for the set of objects with the property F. This only has as inhabitants objects with the attribute F. However the use of subsets in its turn raises difficulties. Consider a well known problem, the root problem. We have a function which has the value 0 for some argument and we wish to compute such an argument. So we may write as specification:

$$(\forall f : \{g : nat \to nat | \exists x : nat.g(x) = 0\})$$
$$\{n : nat | f(n) = 0\}$$

Suppose we had for an input h a proof object for $\exists x : nat.g(x) = 0$. Then from this we would be able to extract both a root and a proof that it is a root. We would then be in a position to use subset introduction. But since we are using subsets this direct proof is unavailable.

4 The Acc Type

Nordström has proposed an extension to Martin-Löf's type theory which he calls the Acc type [5]. Following Nordström I use the notation $a : Acc(A, \prec)$ to mean that a is a member of the well-founded part of A with respect to the relation $\prec$. Note that there may be elements of A which are not in the well-founded part. We think of well-foundedness as involving the absence of infinite descending chains. The introduction rule is to the effect that an element is in the well-founded part of a set with respect to a relation if every element "less than" that element is in the well-founded part with respect to that relation. The elimination rule is the principle of well-founded induction.

These rules have been implemented in Oyster in sequent calculus form. In Oyster we prove goals by refinement. So the problem of proving a particular goal of the form $H_1, \ldots \vdash G$, where $H_1, \ldots$ is a list of assumptions and G is a theorem, is reduced to that of proving a list of subgoals $SG_1, \ldots$. Our goal must be derivable from these subgoals by use of one of the refinement rules.

Our introduction rule (simplified) states that in order to prove a goal of the form:

$$H \vdash X{:}acc(A, \prec)$$

We need to prove:

$$X{:}A$$
$$Y{:}A, Y \prec X \vdash Y{:}acc(A, R)$$

The intuition here is that if all the elements that precede X are accessible then so is X.

The elimination rule (again simplified) states that in order to prove a goal of the form:

$$H \vdash SPEC(X)$$

We need to prove:

$$W : A,$$
$$U : \forall V{:}\{V' : A | V' \prec W\} SPEC(V)$$
$$\vdash SPEC(W)$$
$$\text{(where X:acc(A,R) in H))}$$

The extract term for the goal will be $wo\_ind(X, [W, U, F])$ where F is the extract term for $SPEC(W)$.

We evaluate this as follows. Suppose that X evaluates to $X*$. Then the result of evaluating $wo\_ind(X, [W, U, F])$ is F with W and U replaced by $X*$ and $\lambda Y.(wo\_ind(Y, [W, U, F]))$ respectively.

The Acc type constitutes a possible solution to the two problems outlined earlier. Firstly, it constitutes a uniform mechanism for handling general recursion in type theory. Secondly, it provides a method for eliminating computationally redundant information. The reason for this is that a proof using the Acc type will consist of two parts, a proof that some relation is well-founded over some set and a proof of a goal using this as a lemma. The final extract term however will only contain computational information relating to the latter proof. The computational information relating to the former proof is 'thrown away' and the extract term is simply the program that we are attempting to construct. It is in order to effect this 'information hiding ' that we use a subset in the elimination rule above. We illustrate with a simple example - quicksort.

(There are a number of further interesting examples of proofs involving the Acc type in Saaman and Malcolm [7]).

5 Quicksort

To illustrate the way in which one can use the Acc Type to solve the problem of computational redundancy we will consider an example - quicksort.

The specification for the quicksort program is:

$$\forall k : list(nat)\{l : list(nat)|perm(k, l) \wedge ordered(l)\}$$

We define two auxiliary functions - $filter\_left$ and $filter\_right$ which partition the tail of a list $h :: t$ into two sublists, a list of elements of t less than h and a list of elements of t greater than or equal to t.

If we take the specification as a theorem to be proved in Oyster we must determine the appropriate relation - in this case we will take "$l \prec m$" to mean that that the length of l is less than the length of m. Next we invoke a tactic which given that relation generates two subgoals. One is:

$$\begin{array}{c} m : list(nat) \\ \vdash m{:}acc(list(nat), \prec) \end{array}$$

In effect the proof of this establishes that lists are well-founded under the $\prec$ relation, where we take the measure for this to be the length of a list. Such proofs employ the standard induction scheme for natural numbers. The second subgoal is:

$$\begin{array}{c} m : list(nat) \\ g : \forall n : \{n' : list(nat)|n' \prec m\}\{l : list(nat)|perm(n, l) \wedge ordered(l)\} \\ \vdash \{l' : list(nat)|perm(m, l') \wedge ordered(l')\} \end{array}$$

We do a case case split, according to whether or not m is nil.

We consider the case where m is nil. Clearly nil is an ordered permutation of nil so this case is trivial.

We take m to be of the form $(h :: t)$. We abbreviate:

$$\begin{array}{c} filter\_left(h, m)\ to\ LEFT\ m \\ filter\_right(h, m)\ to\ RIGHT\ m \end{array}$$

We do an elimination on n with $LEFT\ m$. We also do an elimination on n with $RIGHT\ m$. These steps require us to show that both introduced lists are less than m. We then use the induction hypothesis to obtain ordered permutations of both filters, say $g(LEFT\ m)$ and $g(RIGHT\ m)$. We then do an introduction with $g(LEFT\ m) <> h :: g(RIGHT\ m)$. We prove that this is an ordered permutation of m.

Consider now the extract term.

The termination proof establishes that $k{:}acc(list(nat), \prec)$. We abbreviate our goal to $SPEC(k)$.

The extract term for the goal will be:

$$\text{wo\_ind(k,[m,g,f])}$$

Where f is the extract term for the conclusion of the second subgoal, that is:

$$g(LEFT\ m) <> h :: g(RIGHT\ m)$$

The result of evaluating $wo\_ind(k, [m, g, f])$ is f with m replaced with k and g with:

$$\lambda y.(wo\_ind(y,[m,g,f]))$$

That is:

$$wo\_ind(LEFT\ k,[m,g,f]) <> (hd\ k) :: (wo\_ind(RIGHT\ k,[m.g,f]))$$

I have used familiar notation in place of Oyster terminology. The extract term obtained from the proof of the above in Oyster is precisely the program we want; and if we evaluate it as applied to a list of natural numbers, it will output a sorted list of those numbers.

6 The Root Problem revisited

We recall the specification:

$$(\forall f : \{g : nat \rightarrow nat | \exists x : nat.g(x) = 0\})\{n : nat | f(n) = 0\}$$

To solve this we need to use a set I defined as:

$$I = \{y : nat | \forall t : nat.t < y \rightarrow \neg f(t) = 0\}$$

Our induction step is:

$$\begin{array}{c} p : I \\ g : \forall q : \{q' : I | q' = succ(p)\}\{n : nat | f(n) = 0\} \\ \vdash \{n : nat | f(n) = 0\} \end{array}$$

We do a case split with $f(p) = 0$. If this is true then p is our witness. Otherwise we have $\neg(f(p) = 0)$. But p is in I and, since p is not a root, $succ(p)$ is also in I and we have our goal. We need to find an item which is in I. 0 will do. We must prove that $0 : Acc(I, pred)$. This is trivial as I is finite.

So have as the extract term for the induction step:

if f(p) = 0 then p else g(succ p)

The extract term for the goal will be:

wo_ind(0,[p,g,if f(p) = 0 then p else g(succ p)])

The evaluation of this is:

If f(0) = 0 then 0 else wo_ind(succ(0),[p,g,h]).

Again I have used familiar notation instead of Oyster's.

7 Rippling-across

If we extend Oyster with the Acc Type we have provided ourselves with the means for modifying Clam so that we employ a single form of induction - well-founded induction. Using the Acc Type will also guarantee that the extract term from a proof will contain no computationally redundant information.However Clam is a highly successful theorem proving system, so our first concern is to insure that the attaining of these benefits still allows us to employ existing methods - in particular rippling and middle-out reasoning.

In this connection the first point to observe is that the principle of well-founded induction gives a schema for induction which does not contain wave-fronts so how is the rippling to begin? In fact however there is an experimental mechanism in Clam which is able to deal with this problem - that of rippling-across. The basic idea here is due to A.Steven's, see [2].

This was introduced to deal with proofs involving destructor style induction. One's first thought here is that one will obtain wave fronts in the induction hypothesis so one can ripple forwards in the induction hypothesis rather than backwards in the induction conclusion. However for reasons which need not concern us here a somewhat more complicated method is needed. In rippling-across we create in the induction conclusion wave-fronts and anti-wave-fronts. The latter neutralise the wave fronts in the induction hypothesis.

Consider an example - the associativity of append. The specification is:

$$\forall l, m, n : list(nat). l <> (m <> n) = (l <> m) <> n$$

I will assume that the induction will be on the variable l. We replace l with a metavariable in the induction hypothesis to indicate that it is a potential wave front. So we have:

$$\begin{gathered} l, m, n : listnat, \\ L <> (m <> n) = (L <> m) <> n, \\ \vdash l <> (m <> n) = (l <> m) <> n \end{gathered}$$

Where we need to show that any substitution for L is less than l by some appropriate measure.

We now use a destructor style definition of *append*.

$$\neg(X = nil) \rightarrow X <> Y \quad = \quad \boxed{hd(X) :: \underline{\underline{\boxed{(tl(\underline{X}))} <> Y}}}$$

This definition suggests rewriting the induction step in the following manner:

$$\begin{array}{rcl} \boxed{tl(\underline{l})} <> (m <> n) & = & (\boxed{tl(\underline{l})} <> m) <> n \\ \vdash \boxed{hd(l) :: \underline{\underline{\boxed{(tl(\underline{l})} <> (m <> n)}}} & = & (\boxed{hd(l) :: \underline{\underline{\boxed{(tl(\underline{l}))} <> m}}}) <> n \end{array}$$

This leads to the choice of the length of the list as the appropriate well-founded relation, since the length of the tail of non-empty list is less than that of the entire list. The wave fronts now introduced in the induction conclusion "cancel" those in the assumption.

$$\begin{array}{rcl} tl(l) <> (m <> n) & = & (tl(l) <> m) <> n \\ \vdash \boxed{hd(l) :: \underline{(tl(l) <> (m <> n))}} & = & (\boxed{hd(l) :: \underline{(tl(l) <> m)}}) <> n \end{array}$$

We ripple out one step using:

$$(\boxed{l :: \underline{m}}) <> n \rightarrow \boxed{l :: \underline{(m <> n)}}$$

To obtain:

$$\begin{array}{rcl} tl(l) <> (m <> n) & = & (tl(l) <> m) <> n \\ \vdash \boxed{hd(l) :: \underline{(tl(l) <> (m <> n))}} & = & \boxed{hd(l) :: \underline{((tl(l) <> m) <> n)}} \end{array}$$

We can now fertilize to complete the proof.

So we see that it is possible to use the rippling method with well-founded induction.

8 Prime Factorisation

In general we are interested in using middle-out reasoning to find a program satisfying a specification of the form:

$$\forall x : A\{y : B|R(x,y)\}$$

So we are looking for a program $PROG$ such that $PROG(x) : \{y : B|R(x,y)\}$. In order to use the induction hypothesis in the case of a particular x of type A it is necessary that x should be "reducible" in some, yet to be determined, sense. If x is such a reducible object then there will be an object of the form $g(x_1, \ldots x_n, y_1, \ldots y_n)$, call it $x*$ such that $x = x*$. (In fact there are cases where the relation between x and $x*$ is not identity but I shall not consider this complication.) The objects x_i must be of type A and less than x by some well founded measure. We can then assume the existence of $PROG(x_i)$ for any such x_i and use this construction in obtaining $PROG(x)$.

In general we will need, in connection with a particular example, a lemma to the effect that $x*$ exists for any reducible x, and that the constituents of $x*$ have appropriate properties. In the case of an object x_i this will be to the effect that $x_i \prec x$, so we can then use the objects x_i to eliminate on the induction hypothesis. The key problem is to identify the relation in which the x_i stand to x.

To see how existing methods in Clam can be adapted to deduce this function let us look again at the prime factorisation theorem. The induction hypothesis is:

$$\begin{gathered} p : nat^+ \\ \forall x : \{x' : nat^+ | x' < p\}\{xl : list(prime)|product(xl) = x'\} \\ \vdash \{pl : list(prime)|product(pl) = p\} \end{gathered}$$

We assume as before (but with base cases):

$$\begin{aligned}
nil{:}list(type) & & & \quad (1)\\
(\boxed{HD :: \underline{TL}}){:}list(type) &\Rightarrow& \boxed{HD{:}type \wedge \underline{TL{:}list(type)}} & \quad (2)\\
prod(nil) &=& s(0) & \quad (3)\\
prod(\boxed{HD :: \underline{TL}}) &\Rightarrow& \boxed{HD \times \underline{prod(TL)}} & \quad (4)\\
\boxed{U \times \underline{V}} = \boxed{U \times \underline{W}} &\Rightarrow& V = W & \quad (5)
\end{aligned}$$

We can rewrite our goal as:

$$\vdash PL : A \wedge product(PL) = P$$

The meta-variable PL indicates that we are looking for an appropriate witness. The meta-variable P indicates that we may substitute for p expressions containing wave-fronts. We now look at possible instantiations for P. If rippling is to take place we need a match with the left hand side of a wave rule. (1),(2) and (4) are of the wrong form. (3) gives us a solution for $p = s(0)$. (5) is a possible match. So we try rewriting P as $U \times W$. But now we need a rule to rewrite $prod(PL)$ as $U \times V$. (2) will do this provided PL is of the form $HD :: TL$.Then we can ripple out (just as before):

$$\begin{aligned}
\ldots \vdash & \boxed{HD : prime \wedge \underline{TL{:}list(prime)}} \wedge prod(\boxed{HD :: \underline{TL}}) = U \times W \\
\ldots \vdash & \boxed{HD : prime \wedge \underline{TL{:}list(prime)}} \wedge \boxed{HD \times \underline{prod(TL)}} = U \times W \\
\ldots \vdash & \boxed{HD : prime \wedge \underline{TL{:}list(prime)}} \wedge prod(TL) = W
\end{aligned}$$

As a result of the second step we can impose on U the condition that it be prime. Consider the third step. As a general strategy we should leave invoking the induction hypothesis for as long as possible since this will only work with reducible objects and we have yet to determine what these are. We try (3) again for a match. This gives us one with $TL = nil$ and $W = s(0)$. This gives a solution if p is prime. We now need to invoke the induction hypothesis. We instantiate TL with $PROG(V)$. This requires us to prove that $W \prec P$. Our solution is then $U :: PROG(V)$ with U prime. So we deduce that a reducible number in this example to be a composite number - a number which is neither $s(0)$ nor prime.

What is the relation we are looking for? The above reasoning suggests a necessary condition:

$$x \prec y \rightarrow \exists p{:}prime.(y = p \times x)$$

But we also need a relation with respect to which the set we are interested in is well-founded. This means that the relation must not be reflexive. So we have:

$$x \prec y \leftrightarrow \exists p{:}prime.(y = p \times x) \wedge \neg(x = y)$$

We now have the lemma we require. It is to the effect that any composite number y is the product of a prime number p and a number x such that $x \prec y$, where $\prec$ is defined as above. We have to prove this lemma and show that the positive natural numbers are well-founded with respect to this relation.

We conclude that it is possible to perform middle-out reasoning with well-founded induction using methods which are similar to those already in existence in Clam.

9 Conclusion

We have looked at a way of extending Oyster with the Acc Type and Clam with appropriate methods for well-founded induction. The extension is a natural one, enabling rippling and middle-out-reasoning to be used in a wider context. In this sense it confirms the soundness of those basic ideas. The implementation of the Acc Type in Oyster provides empirical verification that it can be used to eliminate computational redundant information, a result which is of some significance as the elimination of such information is a problem which many researchers engaged in synthesising programs confront. For example see [6]

One of the advantages of using the Acc Type in Clam is that it gives one a slightly different perspective on how to deal with the various kinds of eureka steps involved in a proof. A perspective that one might consider more natural for a programmer trying to synthesis a program from a specification, rather than a mathematician trying to automate the proof of a theorem. So one looks for a proof of termination rather than the proof of the correctness of a scheme of induction. This connects up the work of the MRG with that of researchers automating proofs of well-foundedness. Possibly this will be a source of new ideas enabling further extensions to the basic ideas already employed.

References

[1] A.Bundy, A.Smaill,J.Hesketh. “Turning Eureka Steps into Calculations in Automatic Program Synthesis” In proceedings of UK IT, 1990

[2] Bundy, A. et al. “Rippling: A Heuristic for Guiding Inductive Proofs.” Forthcoming

[3] R.L.Constable *et al.* “Implementing Mathematics with the Nuprl Proof Development System.” Prentice Hall, 1986.

[4] C.Horn. “The Nuprl Proof Development System.” Working paper 214, Dept. of Artificial Intelligence,Edinburgh,1988. The Edinburgh version of Nuprl has been renamed Oyster.

[5] B. Nordström. “Terminating General Recursion.” In preparation (1987).

[6] C. Paulin & B. Werner. “Extracting and Executing Programs in the Inductive Constructions System: a Progress Report.” Proceedings of the First Annual Workshop on Logical Frameworks.

[7] E.H.Saaman and G.R. Malcolm "Well-founded Recursion in Type Theory." Computing Science Notes 8710 Department of Mathematics and Computing Science of Groningen University