TURNING EUREKA STEPS INTO CALCULATIONS IN AUTOMATIC PROGRAM SYNTHESIS *

Alan Bundy, Alan Smaill and Jane Hesketh

University of Edinburgh, Scotland

Abstract

We describe a technique called *middle-out reasoning* for the control of search in automatic theorem proving. We illustrate its use in the domain of automatic program synthesis. Programs can be synthesised from proofs that their logical specifications are satisfiable. Each proof step is also a program construction step. Unfortunately, a naive use of this technique requires a human or computer to produce proof steps which provide the essential structure of the desired program. It is hard to see the justification for these steps at the time that they are made; the reason for them emerges only later in the proof. Such proof steps are often call 'eureka' steps. Middle-out reasoning enables these eureka steps to be produced, automatically, as a side effect of non-eureka steps.

1 INTRODUCTION

We describe a technique called *middle-out reasoning* for the control of search during automatic program synthesis. Computer program synthesis is conducted by our Oyster program, Horn [1], which was based on the Cornell University, Nuprl system, Constable *et al* [2]. To synthesise a program one provides a logical specification describing a relation, $spec(inputs, output)$, between the inputs and outputs of the proposed program. Oyster is then used as an interactive theorem prover to prove a conjecture of the form:

$$\forall inputs\ \exists output\ spec(inputs, output) \qquad (1)$$

in a logic based on Martin-Löf Intuitionist Type Theory[1], Martin-Löf [3]. This logic is *constructive*, *i.e.* the proof that an *output* exists for any combination of *inputs* must also show how to *construct* the *output* from the *inputs*. This construction recipe can be extracted from the proof and turned into a computer program.

Oyster proceeds by a process of backwards reasoning from the goal. Associated with each of its backwards proof steps is a program construction step. As the proof proceeds a functional/logic program, $prog(inputs)$, which is also expressed in the Type theory logic, is constructed. This program meets the specification:

$$\forall inputs.\ spec(inputs, prog(inputs))$$

There is a duality between proof steps and program construction steps, *e.g.* an inductive proof produces a recursive program. Since recursion is pervasive in functional/logic programs, we are particularly interested in proofs by mathematical induction. Such proofs present especially difficult problems of search control, namely in the choice of induction rule and induction variable(s).

To automate this process of program synthesis we have built a program Clam, van Harmelen [4], which guides Oyster in its search for a proof. Clam contains a collection of *tactics*. These are computer programs which apply Oyster rules, and hence direct its search. Clam analyses the conjecture to be proved and uses AI planning techniques to construct a *proof plan*, which is a tactic especially designed for the current conjecture. Further details of the Oyster-Clam system can be found in Bundy *et al* [5].

In Oyster's logic, a conjecture of form (1) is usually proved by eliminating its quantifiers at some stage, to form a goal of the form:

$$spec(inputs, prog(inputs)) \qquad (2)$$

where $prog(inputs)$ is called the *witness* of the existentially quantified variable *output*. This rather defeats the object of the exercise. As can be seen in goal (2), it is necessary for the Oyster user to *provide* the very program which Oyster is supposed to be synthesising. Even if the proof is first divided into cases and the quantifiers eliminated separately in each case, as in §3, the combination of the separate witnesses still defines the program.

The rest of the proof constitutes a mere *verification* that this program meets the specification, rather than a *synthesis* of the program. This elimination of an existential quantifier is an example of a *eureka* step, *i.e.* a step whose justification is not apparent at the time it is made. It becomes evident later in the proof, of course, when the verification succeeds. Eureka steps present a problem for both human and computer theorem proving. In either case it is hard to see how they might be thought of, *e.g.* what tactic might be implemented to make eureka steps.

*The research reported in this paper was supported by SERC grant GR/E/44598 and an SERC Senior Fellowship to the first author. We are grateful for discussions with the other members of the mathematical reasoning group at Edinburgh.

[1] However, in the interests of wider readability we have translated the Martin-Löf language into a more conventional notation in the examples given below.

Another example of a eureka step is the choice of appropriate induction variables and induction rule of inference. These will determine the forms of recursion to be used by the program.

2 PROGRAM SYNTHESIS BY THEOREM PROVING

To illustrate the technique of program synthesis and the eureka steps it requires, we will show how a program, $factors(x)$, for factorizing a positive integer, x, into its prime factors, can be synthesised from a proof of the prime factorization theorem. The prime factorization theorem can be expressed in English as:

> "For all positive integers, x, there is a list of prime numbers, xl, such that x equals the product of the numbers in xl."

It can be represented in our Type Theory as the formula:

$$\forall x{:}posint\ \exists xl{:}list(prime)\ prod(xl) = x \tag{3}$$

where $X : T$ means X is an object of type T, *posint* is the type of positive natural numbers, *prime* is the type of prime numbers, and $list(T)$ is the type of lists of objects of type T, *e.g.* $xl : list(prime)$ means xl is a list of primes. The function $prod : list(posint) \mapsto posint$ takes a list of numbers and returns their product, *i.e.*

$$\begin{aligned} prod(nil) &= 1 \\ prod(hd :: tl) &= hd \times prod(tl) \end{aligned}$$

where *nil* is the empty list and :: is the infix list constructor.

We can prove this theorem easily by using the *prime*$\times$ induction rule:

$$\frac{\vdash_\alpha P(1) \qquad p{:}primes, x'{:}posint, P(x') \vdash_\beta P(p \times x')}{\vdash_\gamma \forall x{:}posint.\ P(x)}$$

i.e. we prove the theorem for $x = 1$, then we assume it for $x = x'$ and prove it for $x = p \times x'$, where p is a prime number. The greek letters labelling the $\vdash$ symbols are the program fragments associated with the proofs of these sequents. The program construction step associated with this induction rule defines the γ program in terms of the α and β program fragments, as follows:

$$\begin{aligned} \gamma(x) \equiv\ & \text{if} \quad x = 1 \quad \text{then} \quad \alpha \\ & \text{else}\ \beta(p, x')\ \text{where}\ p = first(x) \wedge x' = rest(x) \end{aligned}$$

where $first(x)$ is the smallest prime number that divides x and $rest(x)$ is the quotient when x is divided by $first(x)$. In this case $factors(x) = \gamma(x)$. So the decision to use *prime*$\times$ induction ensures that the program will use a dual form of recursion. Deciding to use this esoteric form of induction is our first eureka step.

The base case of the induction is:

$$\vdash_\alpha \quad \exists xl{:}list(prime)\ prod(xl) = 1 \tag{4}$$

and the step case is:

$$\begin{aligned} & p{:}prime, \\ & x'{:}posint, \\ & \exists xl{:}list(prime)\ prod(xl) = x' \quad \vdash_\beta \\ & \qquad \exists xl{:}list(prime)\ prod(xl) = p \times x' \end{aligned} \tag{5}$$

The base case is readily proved by eliminating the existential quantifier in (4) and introducing the witness *nil* for xl. This also instantiates α to *nil*. Similarly, the step case is proved by first eliminating the existential quantifier in the induction hypothesis part of (5) with witness xl' and then the one in the induction conclusion with witness $p :: xl'$. Oyster is able to work out that xl' is $factors(x')$, so it instantiates β to $p :: factors(x')$. The program is now:

$$\begin{aligned} & factors(x) \ = \\ & \quad \text{if} \quad x = 1 \ \text{then}\ nil \\ & \quad \text{else}\ p :: factors(x') \\ & \qquad \text{where}\ p = first(x) \wedge x' = rest(x) \end{aligned}$$

The program is now fully synthesised.

The witness introduced by the elimination of the existential quantifier in the induction hypothesis is standard, but the witnesses introduced by the elimination of the other two quantifiers constitute eureka steps. Note how these two witnesses are the values of the function $factors(x)$ in the base and step cases of the recursion.

3 THE *RIPPLE_OUT* TACTIC

The remaining parts of the proof will provide a verification that our choices of induction rule and existential witnesses yield a program that meets the specification. The remaining part of the step case is to prove:

$$\begin{aligned} p{:}prime, & \\ x'{:}posint, & \\ xl'{:}list(prime) & \\ prod(xl') = x' & \quad \vdash_\beta \quad \boxed{p ::} xl'{:}list(prime) \wedge \\ & \qquad prod(\boxed{p ::} xl') = \boxed{p \times} x' \end{aligned} \tag{6}$$

Note that the existential quantifiers have all been eliminated and the existential variables replaced by their witnesses. The witness of the existential variable in the induction conclusion, $p :: xl'$ must be shown to have the right type, $list(prime)$, so the subgoal, $p :: xl' : list(prime)$, becomes part of the induction conclusion.

Three sub-expressions of the induction conclusion have been placed in boxes. These are examples of *wave fronts*. *Wave fronts* are those sub-expressions of the induction conclusion in which it differs from the induction hypothesis. The Clam system contains a tactic called *ripple_out* whose task is to make the induction hypothesis appear as a sub-expression of the induction conclusion. It works by rewriting the induction conclusion to move the wave fronts outwards from their original deeply nested positions. The rules used by *ripple_out* are called *wave rules*. Wave rules are rewrite rules of the form[2]:

$$F(\boxed{S_1(\,\underline{U_1}\,)}, \ldots, \boxed{S_n(\,\underline{U_n}\,)}) \Rightarrow \boxed{T(\,\underline{F(U_1,\ldots,U_n)}\,)}$$

where F, T and the S_i are terms with distinguished arguments and T may be empty, but F and the S_i must not be. The S_i are old wave fronts and T is the new wave front. Application of a wave rule ripples some wave fronts out by one stage. Repeated application ripples them all to the outside of the induction conclusion.

The wave rules required for this proof are:

$$\boxed{hd :: \underline{tl}}{:}list(type) \Rightarrow \boxed{hd{:}type \wedge \underline{tl{:}list(type)}} \quad (7)$$

$$prod(\boxed{hd :: \underline{tl}}) \Rightarrow \boxed{hd \times \underline{prod(tl)}} \quad (8)$$

$$\boxed{u \times \underline{v}} = \boxed{u \times \underline{w}} \Rightarrow v = w \quad (9)$$

Applying these wave rules to the induction conclusion of (6) rewrites the step case as follows. After application of rule 7 to the first wave front, Clam derives:

$$\begin{aligned} &p{:}prime,\\ &x'{:}posint,\\ &xl'{:}list(prime)\\ &prod(xl') = x' \quad \vdash_\beta \quad \boxed{p : prime \wedge \underline{xl'{:}list(prime)}} \wedge\\ &\qquad prod(\boxed{p :: \underline{xl'}}) = \boxed{p \times \underline{x'}} \end{aligned}$$

After application of rule 8 to the second wave front, it derives:

$$\begin{aligned} &p{:}prime,\\ &x'{:}posint,\\ &xl'{:}list(prime)\\ &prod(xl') = x' \quad \vdash_\beta \quad \boxed{p : prime \wedge \underline{xl'{:}list(prime)}} \wedge\\ &\qquad \boxed{p \times \underline{prod(xl')}} = \boxed{p \times \underline{x'}} \end{aligned}$$

And after application of rule 9, simultaneously, to the second and third wave fronts, it derives:

$$\begin{aligned} &p{:}prime,\\ &x'{:}posint,\\ &xl'{:}list(prime)\\ &prod(xl') = x' \quad \vdash_\beta \quad \boxed{p : prime \wedge \underline{xl'{:}list(prime)}} \wedge\\ &\qquad prod(xl') = x' \end{aligned}$$

Figures 1 and 2 illustrate this rippling-out process graphically by showing two of the stages that the induction conclusion goes through.

After these three rule applications each of the three conjuncts of the induction conclusion is identical to one of the induction hypotheses. The induction hypothesis can then be used to prove the induction conclusion. The Clam system has a tactic called *fertilization* whose task is to match the induction hypothesis against sub-expressions of the induction conclusion and replace any such sub-expressions by *true*. Doing this completes the proof of the step case in this example.

A fuller explanation of *ripple_out* can be found in [6].

[2] This is not the most general form that wave rules can take, but is sufficient for the examples in this paper. A more general form is given in Bundy *et al* [6].

4 MIDDLE-OUT REASONING

We can think about the eureka steps described in §2 in the following way. The choices of induction scheme and existential witnesses are actually made in such a way as to allow the rest of the proof to proceed successfully. In particular, the eureka choices will enable the subsequent *ripple_out* tactic to succeed. However, since the eureka steps are made before *ripple_out* is applied, this is not immediately obvious.

This observation suggests a way to automate the eureka decisions, namely: postpone making the eureka steps and apply *ripple_out first*; then integrate the eureka steps into the rippling as required to enable it to continue. Effectively we do the middle of the proof first. In the process we calculate what form the beginning of the proof should take to make the middle of it succeed. We call this strategy *middle-out reasoning*.

4.1 Rippling Under Existential Quantifiers

We consider first the problem of choosing witnesses for existentially quantified variables. Our solution is not to eliminate the existential quantifiers at all, but to modify the rewriting procedure so that rippling can be carried out under existential quantifiers. To illustrate this process, consider again the proof of the prime factorization theorem. We return to the step case of the proof, just after the application of induction.

$$\begin{aligned} &p{:}prime, \qquad (10)\\ &x'{:}posint,\\ &\exists xl'{:}list(prime)\ prod(xl') = x' \quad \vdash_\beta\\ &\qquad \exists \boxed{xl}{:}list(prime)\ prod(\boxed{xl}) = \boxed{p \times \underline{x'}} \end{aligned}$$

This time we have marked in the wave fronts *before* eliminating the existential quantifier. Recall that the witness for xl in the induction conclusion, $p :: xl'$, contained a wave front, $\boxed{p ::}$, (see (6)). This justifies our marking this existential variable as a wave front.

This remark generalises to all existentially quantified variables in induction conclusion. Their witnesses will be the value of the synthesised program in the step case of the recursion, *i.e.* some function of the value of the program when it is called recursively. When this function is not the identity function, then it will be a wave front. So all existential variables in induction conclusions are potential wave fronts. Note that the occurrence of the existential variable in the quantifier declaration is marked as a wave front, as well as all occurrences in the body of the formula. This is because the quantifier declaration also contains a type declaration, which will require rippling with wave rules for types, like (7) above.

We now proceed to apply *ripple_out*, using the same wave rules as in §3, but without first removing the $\exists xl : list(prime)$. Our modified rewriting procedure has the freedom to instantiate existential variables to compound terms of the same type. This is because in the unmodified procedure these compound terms could have been introduced as witnesses before *ripple_out* was applied. To take advantage of this possibility the modified rewriting procedure matches the left hand side of the rewrite rule with expressions in the goal, treating existential variables as free variables that can be instantiated. When an existential variable is instantiated to a witness of the same type, the existential quantifier that governs it will no longer be required, and should be deleted. However, a new existential quantifier will be required for each variable contained in the witness. To assist with checking that the witness has the required type and with the calculation of the types of the new existential variables, it helps if the wave fronts in the quantifier/type declarations are rippled before those in the body of the formulae.

Consider, for instance, the application of wave rule (7) to the first wave front in (10).

$$\boxed{\exists hd\!:\!prime}\ \exists tl\!:\!list(prime)\ prod(\boxed{hd ::}\,tl) \quad = \quad \boxed{p\times}\,x'$$

This instantiates xl, and hence β, to $hd :: tl$. The existential quantifier governing xl is replaced by two existential quantifiers: one for hd and one for tl. Note how the types of these new existential variables are provided by the right hand side of rule (7). This use of a type rule also guarantees that the witness has the right type. Note also how the second occurrence of xl, in the body of the formula, has been instantiated to $hd :: tl$.

The witness to the existential variable, xl, has now been determined. It was not necessary to do this as a eureka step. The appropriate witness was calculated by the matching routine during the application of *ripple_out*. No search was required for this. The wave front around xl had to be rippled out at some time (in fact, it is best to ripple it out first). Rule (7) was the only matching wave rule because of the restriction imposed by the subexpression $list(prime)$, the type of xl.

We now apply wave rule (8) to the second wave front, yielding:

$$\boxed{\exists hd\!:\!prime}\ \exists tl\!:\!list(prime)\ \boxed{hd\times}\,prod(tl) \quad = \quad \boxed{p\times}\,x'$$

This application does not instantiate any existential variables, so does not require any changes in the existential quantifiers.

We can now apply multi-wave rule (9), simultaneously, to the second and third wave fronts, yielding:

$$\exists tl\!:\!list(prime)\ prod(tl) \quad = \quad x'$$

This rewrite instantiates hd to p. This instantiation is only possible because p is a term with the same type as hd. Since hd has been instantiated its existential quantifier must be removed. Since p is a constant, and therefore contains no new variables, no new existential quantifiers are introduced. This has the side effect of removing the first wave front. The remaining formula matches the induction hypothesis. So fertilization removes it, instantiating tl to xl', and hence xl and β to $p :: xl'$, as required.

A similar process can be carried out in the base case. The role of *ripple_out* being played by the Clam tactic, *base*, which rewrites formulae using the base cases of recursive definitions.

4.2 Using Meta-Variables for Induction Terms

We now consider the problem of choosing induction rules. Our solution is make a 'least commitment' induction step by using a schematic induction rule in which the induction term is a meta-variable. We then allow this meta-variable to become instantiated during the subsequent rippling out. Since meta-variables are not allowed in Oyster's logic, this reasoning is handled within the Clam planner. Clam applies *ripple_out* and works out what induction rule is required. It then instructs Oyster to apply the appropriate induction rule and then ripples the wave fronts that induction creates.

To illustrate this we return again to the step case of the prime factorization proof, just after the application of induction, *i.e.*

$$\begin{array}{l} x'\!:\!posint, \\ \exists xl\!:\!list(prime)\ prod(xl) = x' \quad \vdash_\beta \\ \qquad \exists \boxed{xl}\!:\!list(prime)\ prod(\boxed{xl}) = \boxed{X} \end{array}$$

but instead of the induction term, $p \times x'$, we have used the meta-variable, X, which stands for some term containing x', *i.e.* we have replaced all occurrences of x', in the induction conclusion, by X. All universally quantified variables are candidate induction variables, but x is the only candidate in this example. In general, we must try replacing each universal variable, in turn, by such a meta-variable, to see which replacements permit *ripple_out* to succeed. Note that the declaration of p has been omitted from the hypothesis, since we do not yet know what parameters might be introduced by the induction.

Rippling-out then proceeds as in §4.1 until we get to the formula:

$$\boxed{\exists hd\!:\!prime}\ \exists tl\!:\!list(prime)\ \boxed{hd\times}\,prod(tl) \quad = \quad \boxed{\mathrm{X}}$$

Now rule (9) is the only wave rule that applies to the second wave front. Applying it produces:

$$\boxed{\exists hd\!:\!prime}\ \exists tl\!:\!list(prime)\ prod(tl) \quad = \quad X'$$

where X is instantiated to $hd \times X'$. *fertilization* applies to this, instantiating X' to x' and leaving the residue:

$$\exists hd\!:\!prime\ true$$

which is provable provided hd is witnessed by a prime number.

We have thus established that the *ripple_out* tactic will succeed provided that the induction term is $hd \times x'$, where $hd : prime$. Induction rules are indexed in Clam by their induction terms, so this information enables the systems to recover the $prime\times$ induction rule and use it retrospectively. Notice how the appropriate induction rule was chosen as a side-effect of the *ripple_out* tactic.

5 CONCLUSION

In this paper we have described a technique, based on theorem proving, for synthesising programs from logical specifications. We have seen that a naive use of this technique requires eureka steps. In §2 we saw that it was necessary to produce an appropriate induction rule and appropriate existential witnesses 'out of the blue'. Furthermore, these eureka steps provided all the essential structure of the synthesised program, leaving only the verification that this program met the specification. These eureka steps constitute a barrier to the use of the program synthesis technique. This is true whether we want to use it totally automatically, with a computer providing the eureka steps, or semi-automatically, with a human providing them.

We have described a way of finessing the eureka steps, so that program synthesis can be automated. We draw on previous work in which we automated the verification part of the proof. In particular, our *ripple_out* tactic is highly successful in automatically guiding the proving of the induction conclusion from the induction hypothesis. We have arranged the proof construction so that this middle part of the proof is done first. The eureka steps emerge as a side effect of *ripple_out* — they are made in such a way as will permit *ripple_out* to continue. We call this technique *middle-out reasoning*. We are currently implementing middle-out reasoning within the Oyster-Clam system.

References

[1] C. Horn. The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, University of Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

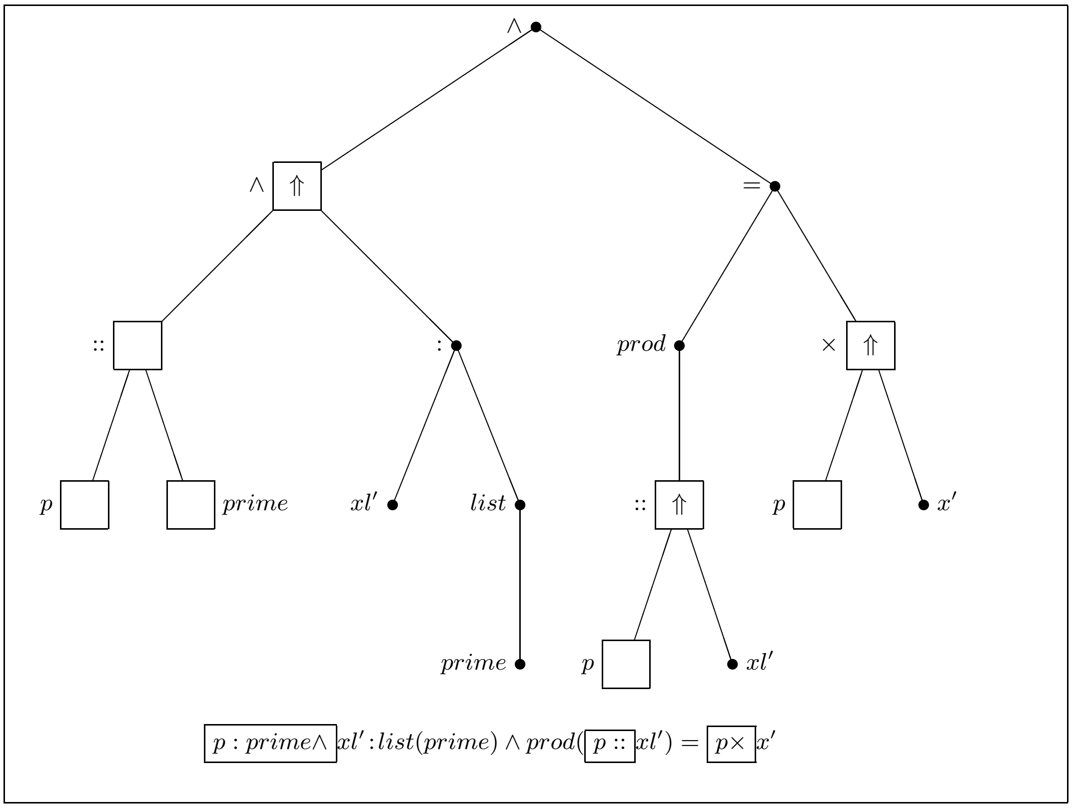
[2] R. L. Constable, S. F. Allen, H. M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[3] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[4] F. van Harmelen. The CLAM proof planner, user manual and programmer manual: version 1.4. Technical Paper TP-4, DAI, 1989.

[5] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. Research Paper 413, Dept. of Artificial Intelligence, University of Edinburgh, 1988. Appeared in Journal of Automated Reasoning, 7, 1991.

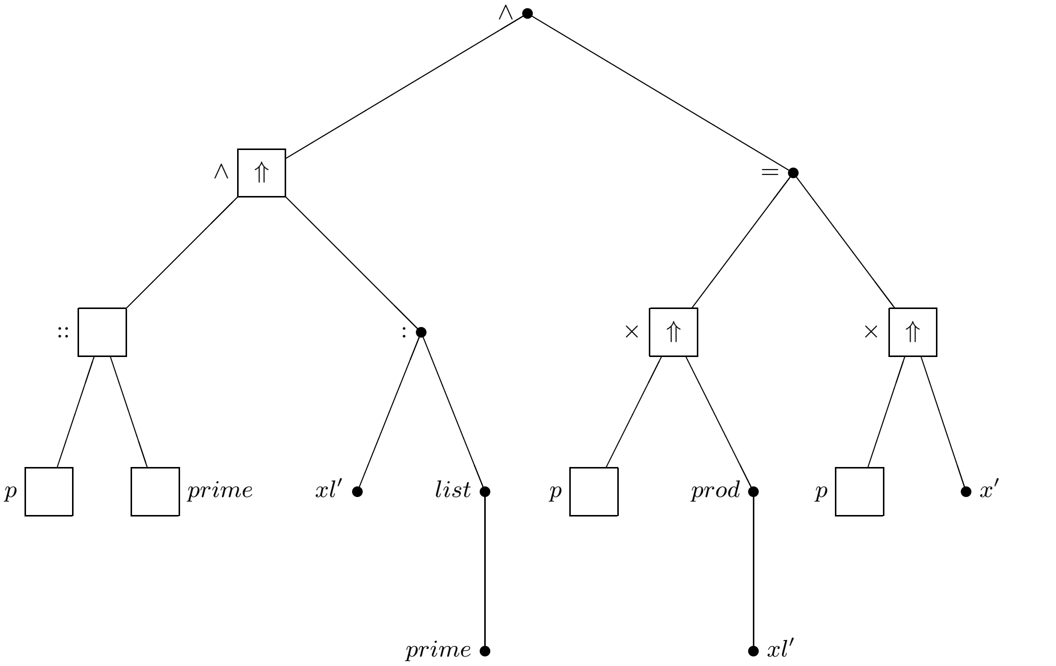
[6] A. Bundy, F. van Harmelen, and A. Smaill. Extensions to the rippling-out tactic for guiding inductive proofs. Research Paper 459, Dept. of Artificial Intelligence, University of Edinburgh, 1989. Also in the proceedings of CADE-10.



The induction conclusion is shown after the application of one wave rule both as an expression tree and as a formula.

Each non-terminal node of an expression tree is labelled by a function or predicate symbol. Each terminal node is labelled by a constant or variable. A function or predicate of arity n has n descendent subtrees: one for each of its arguments. Wave fronts are indicated by square nodes and all other nodes by dots. The ⇑ in the top square node indicates the direction of rippling-out.

Figure 1: Induction Conclusion After One Ripple



$$\boxed{p : prime \wedge}\, xl' : list(prime) \wedge \boxed{p \times}\, prod(xl') = \boxed{p \times}\, x'$$

The induction conclusion is shown after the application of a further wave rule. Note that this tree differs from the first one only in the square nodes. Each ripple moves the square nodes higher up the tree.

Figure 2: The Induction Conclusion After Two Ripples