Part IV

geometry,root=book1.mss

Chapter 10

Using Semantic Information to Guide Proofs

- This chapter describes the use of models to guide Horn clause proofs, illustrated mainly on the domain of Euclidean Geometry.
- Section 10.1 discusses the formalization of Euclidean Geometry as a set of Horn clauses.
- Section 10.2 describes semantic checking, the use of a diagram to prune the search tree.
- Section 10.3 describes the use of a diagram to suggest constructions in geometric proof.
- Section 10.4 shows that a diagram is a model of the axioms and hypotheses of a geometry conjecture.
- Section 10.5 generalizes semantic checking and applies it to a conjecture in Arithmetic.
- Section 10.6 explains why semantic checking does not generalize to non-Horn clauses.
- Section 10.7 proves that semantic checking preserves the completeness of Horn clause proofs.

When humans try to prove theorems they are not guided solely by the syntactic properties of the formulae they are manipulating. They have some idea of what it all *means* (whatever that means?) and they can bring this knowledge to bear.

Even in the original formal mathematical theory, Euclidean Geometry, we have a diagram to guide us. The diagram even has a place in the official layout of the theorem statement.

Note that the conventions for upper and lower case are rather different, in the official layout of Euclidean geometry theorem statements, than the one we have been using. We will have to alter the official layout to correspond to our convention.

Statement The angle bisector is equidistant from the rays of the angle.

Diagram

Given
1. $Seg\ DB$ bisects $Angle\ ABC$
2. $Seg\ DA \perp Seg\ BA$
3. $Seg\ DC \perp Seg\ BC$

Required to Prove
4. $Seg\ AD = Seg\ CD$

The observation that humans usually use diagrams when trying to prove Euclidean Geometry theorems, led Herbert Gelernter, to try to build a theorem proving program which could use a diagram to guide the proof [25, 24]. Gelernter called his program the *Geometry Machine*.

In this section we will make a rational reconstruction of the Geometry Machine. That is, we will explain its main ideas using the terminology developed in the earlier part of this book rather than the terminology used by Gelernter. We will sometimes gloss over or ignore aspects of the Geometry Machine which are incidental to our main theme of how the diagram was used to guide the search. Defying historical niceties we will sometimes refer to this rational reconstruction as the Geometry Machine.

10.1 Formalising Geometry

Before we can discuss how a diagram might guide the search for a proof we must have a search to guide. This means choosing some inference system, and designing some formalism in which axioms and conjectures may be expressed.

The following, functions, predicates, axioms, etc are adapted from [25] and from [27]. Some were suggested by the standard abbreviations of Euclidean Geometry, some, especially the more basic ones, were invented from scratch.

One of Gelernter's goals, when designing the Geometry Machine was for it to produce proofs comparable to the normal human ones. This led him to reject the standard axiomatizations of Geometry, e.g.the ones due to Tarski, Hilbert and Forder. Instead he tried to capture, in axioms, the laws assumed by the school student. This led to, a somewhat ad hoc, highly redundant, rather large, but psychologically plausible list of axioms. It would be tedious to list them all here. Instead we give a small sample, including those needed to illustrate the use of the diagram to guide the proof.

Paul Gilmore, in an analysis of the Geometry Machine [27] divides the axioms into three classes.

(a) Basic axioms relating points to the higher level concepts of: line segments, angles, triangles, etc. These include such definitions as:

> If A and B are distinct points, and B and C are distinct points, and A and C are distinct points, and A, B and C are not collinear, then ABC is a triangle.

We will formalise this as:

$$\begin{array}{l} distinct(A,B) \wedge distinct(B,C) \wedge \\ distinct(A,C) \wedge notcoll(A,B,C) \\ \qquad \rightarrow is\_triangle(A,B,C). \end{array}$$

(b) Axioms expressing symmetries of these predicates and functions, e.g.,

> If segment AB equals segment DC then segment AB equals segment CD.

We will formalise this as:

$$seg(A,B) = seg(D,C) \rightarrow seg(A,B) = seg(C,D). \tag{i}$$

The more familiar axioms expressing equality between angles and line segments, congruence between triangles, parallelism between line segments, etc. e.g.

> Corresponding sides of congruent triangles are equal,

formalised as:

$$tri(A,B,C) \equiv tri(D,E,F) \rightarrow seg(A,B) = seg(D,E), \tag{ii}$$

and

> Two triangles are congruent if they have two equal angles and a corresponding equal side,

formalised as:

$$\begin{array}{l} angle(A,C,B) = angle(D,F,E) \wedge \\ angle(C,A,B) = angle(F,D,E) \wedge \\ seg(B,C) = seg(E,F) \rightarrow tri(A,B,C) \equiv tri(D,E,F). \end{array} \tag{iii}$$

Gilmore also points out that all of these axioms are . This observation has important consequences for the use of the diagram, as we will see below.

10.2 Geometric Proofs

(see section 6.3) produces proofs very like those of the Geometry Machine. So using , the above axioms and a formalization of a conjecture, it is possible to produce some automatic geometry proofs – for instance, a proof of the theorem described above that

> The angle bisector is equidistant from the rays of the angle.

The formalization of this conjecture, negated and put in clausal form is:

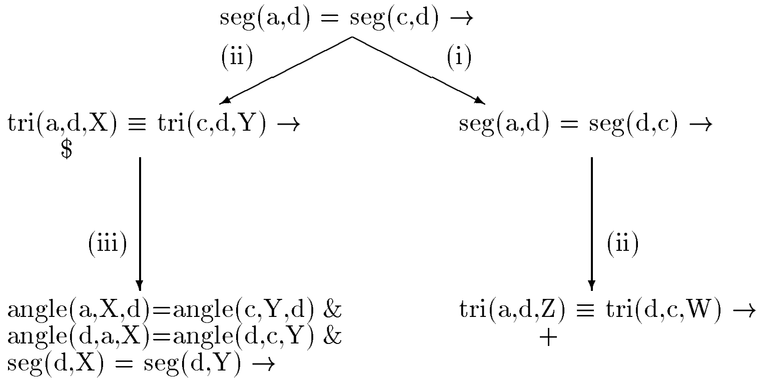

Figure 10.1: A Fragment of the Search Tree for a Simple Geometry Theorem

1. $\rightarrow seg(d, b)$ bisects $angle(a, b, c)$.
2. $\rightarrow seg(d, a) \perp seg(b, a)$.
3. $\rightarrow seg(d, c) \perp seg(b, c)$.
4. $seg(a, d) = seg(c, d) \rightarrow$.

Figure 10.1 contains a fragment of the search tree generated by taking the goal clause, 4, as top clause and applying the axioms from (b) and (c) above to it using Lush Resolution.

Consider the two 'congruence' goals, marked \$ and + in figure 10.1. NB Just the goals, i.e. the antecedent of the clause, not the whole clause. What could be substituted for the variables, X, Y, Z and W so that these goals were provable? If we restrict the possible substitutions to points occurring in the diagram the possibilities are extremely limited.

- Substituting b for X and for Y will make \$ true in the diagram and hence quite likely provable by the program.
- No substitutions of points for Z and W will make + true in the diagram; to find suitable points would require making a construction.

The Geometry Machine used such reasoning to guide the search for the proof in the following ways.

- Variables in goals were instantiated to particular points (i.e. constants) in such a way that the goal instances were true in the diagram.
- Clauses for which no such instantiation was possible were temporarily pruned from the tree.
- The nodes where such instantiations or prunings had been made were remembered as continuation nodes should the diagram ever be extended by construction and hence new points become candidates for instantiation.

Thus the diagram was used to guide the search by pruning some nodes from the tree and delaying the development of others to a second pass. We will call this technique *semantic checking*.

10.3 Constructions

If we discount loops then the resulting search tree is finite. This is because Gelernter's representation of Geometry does not use functions in any essential way: formulae like $seg(A, B) = seg(D, C)$ can be regarded as fancy notation for a four parameter predicate. As noted in section 4.3.2, and section 16.2 function-free theories are decidable. In fact the Lush Resolution search trees for such theories are finite, so exploration will eventually cease, even if no proof has been found. At this stage a continuation node can be selected and used to suggest a single construction which would allow further development of the node.

The angle bisector example above can be proved without such a construction, as can many simple geometry theorems, so to see how the construction process works we will have to look at another example.

> *Statement* If the segment joining the midpoint of the diagonals of a trapezoid is extended to intersect a side, it bisects that side.
>
> *Diagram*
>
> *Given*
> $\rightarrow quadrilateral(a, b, c, d)$
> $\rightarrow seg(b, c) \| seg(a, d)$
> $\rightarrow e\ midpt\ seg(a, c)$
> $\rightarrow f\ midpt\ seg(b, d)$
> $\rightarrow precedes(m, e, f)$
> $\rightarrow precedes(a, m, b)$
>
> *Required to Prove*
> $seg(m, b) = seg(m, a) \rightarrow$

The meanings of the new functions and predicates introduced above is obvious except for '*precedes*'. $precedes(A, B, C)$ means that points A, B and C are collinear in that order.

We will consider a proof of this conjecture which uses the lemma that a line drawn through the midpoint, D, of the line, AB, to intersect AC at E, in triangle ABC, is parallel to the base, BC, iff E is the midpoint of AC. This is illustrated in figure 10.2. A slightly extended version of the if part of this lemma is

$$\begin{array}{l} is\_triangle(A, B, C) \wedge \\ precedes(A, D, B) \wedge precedes(A, E, C) \wedge \\ seg(D, A) = seg(D, B) \wedge seg(E, A) = seg(E, C) \wedge \\ collinear(B, C, F) \\ \quad \rightarrow seg(D, E) \| seg(B, F) \end{array} \qquad \text{(iv)}$$

Figure 10.2: Key Lemma Illustration

Figure 10.3: The Construction of a New Point

The idea of the proof is to construct a line from c through f to intersect ad at k. The resulting diagram is shown in figure 10.3 Point f is established to be the midpoint of ck, because triangles cfb and kfd are congruent. The lemma is used in the if direction, on triangle cak, to establish that ef is parallel to ak. Then the lemma is used in the only if direction, on triangle bad, to establish that m is the midpoint of ab. The hardest part is thinking of the construction of point k.

To see how the diagram can be used to suggest this construction consider the fragment of proof in figure 10.4, in which axiom iv is applied. This is the point of the proof at which ef is shown parallel to ak.

During the first stage of the Geometry Machine's search, the goal clause marked \$ in figure 10.4, was temporarily pruned, because there are no instantiations for X and Y *in the current diagram* which would make all the goals in \$ simultaneously true. However, after the search tree had been exhausted it was resurrected as a continuation node.

But the goals of clause \$ can be satisfied, and can only be satisfied, by the

$$seg(e,f) \| seg(a,d) \rightarrow$$

(iv)

\$

$$\begin{array}{l} is\_triangle(X,a,Y) \wedge \\ precedes(X,e,a) \wedge precedes(X,f,Y) \wedge \\ seg(e,X) = seg(e,a) \wedge seg(f,X) = seg(f,Y) \wedge \\ collinear(a,Y,d) \rightarrow \end{array}$$

Figure 10.4: A Fragment of the Search Tree of the Second Example

substitution $\{c/X, k/Y\}$. In fact, ruler and compass style manipulations, based on these goals, will construct k. Draw a circle, centre e, radius ea, and find its intersection with ea to bind X. Now extend cf to intersect ad at k. Check that fc and fk are equal.

The information that k is the intersection of cf and ad, i.e

$$collinear(c, f, k) \quad \text{and} \quad collinear(a, d, k)$$

can now be added to the theorem hypotheses and the search can proceed from this continuation node. The proof of this example can now be found without further construction being necessary.

10.4 What is the Diagram?

What form does the Geometry Machine diagram take and how was it used?

Gelernter did not use a diagram written on a piece of paper and viewed, by the computer, through some electronic eye (although, if he had had the technology, this method could have been used instead). Rather, a cartesian representation was used, with each point mentioned in the theorem being assigned a pair of x y coordinates. Thus the diagram,

would be represented by some assignment like

$$\begin{array}{l}
a \mapsto < \sqrt{3}, \sqrt{7/6} > \\
b \mapsto < 0, 0 > \\
c \mapsto < \sqrt{3}, 0 > \\
d \mapsto < \sqrt{3}, 1 >
\end{array}$$

chosen in such a way as to make the hypotheses of the theorem (i.e. all but the goal clause) true. The above assignment makes

$\rightarrow seg(d, b)$ bisects $angle(a, b, c)$
$\rightarrow seg(d, a) \perp seg(b, a)$ and
$\rightarrow seg(d, c) \perp seg(b, c)$

all true. The assignment must be chosen by the human user of the Geometry Machine, since Gelernter did not provide a program to do it. The user is well advised to choose an assignment which minimizes the number of things which are accidently true.

The calculation procedures of real number arithmetic can now be used to assign real numbers to any variable free term and hence to assign a truth value to any variable free formula. For instance, $tri(a, d, b) \equiv tri(c, d, b)$ would be assigned the value t and $tri(a, d, b) \equiv tri(d, c, b)$ would be assigned the value f.

The diagram and these calculation procedures can be used to temporarily prune nodes out of the search tree as follows.

- Suppose $G_1 \wedge ... \wedge G_n \rightarrow$ is a newly derived goal clause containing the vector of variables, X. Suppose the vector of constants, $\mathbf{a}$, contains all the constants mentioned in the statement of the theorem. We will treat these constants as objects in a universe.
- The constants are assigned to the variables in the goal clause in all possible ways producing a series of instances. Let $G'_1 \wedge ... \wedge G'_n \rightarrow$ be one of these instances.
- The calculation procedures and the assignment to $\mathbf{a}$ are used to test $G'_1 \wedge ... \wedge G'_n$ to see whether it is true in the diagram.
- All instances failing the test are pruned.
- All instances passing the test become daughters of the original goal clause.
- If the original goal clause contains any variables it is remembered as a possible continuation node. Note that a variable free goal clause has only one instance – itself. Thus the calculation procedures either reject or accept it. There is no need to remember it as a continuation node.

For instance, if the newly derived goal clause is

$$tri(a, d, X) \equiv tri(c, d, Y) \rightarrow$$

then the instances of this are:

$$tri(a, d, a) \equiv tri(c, d, a) \rightarrow$$
$$tri(a, d, a) \equiv tri(c, d, b) \rightarrow$$
.... etc (16 in all)

Only one of these passes the 'truth' test, $tri(a, d, b) \equiv tri(c, d, b)$. So only this instance remains in the search tree.

Exercise 31 *What are the instances of* $seg(d, a) = seg(d, X) \rightarrow$*? Which of them pass the truth test?*

The other use of the diagram mentioned above was to help in the construction of new points. This is a little more difficult than the truth testing use. Given a conjunction of goals like:

$$tri(X, a, Y) \wedge precedes(X, e, a) \wedge precedes(X, f, Y) \wedge$$
$$seg(e, X) = seg(e, a) \wedge seg(f, X) = seg(f, Y) \wedge collinear(a, Y, d)$$

the diagram must suggest how new points may be constructed, which when substituted for some of the variables in the conjunction will make it true. In this case it should suggest the construction of point k, as the intersection of lines ad and cf and suggest the substitution $\{c/X, k/Y\}$.

The trick is to associate algebraic formula, e.g. equations, inequalities etc, with each of the propositions in the conjunction, and to use equation solving techniques to find an assignment of cartesian coordinates for the variables which will make the conjunction true. This assignment can then be used to recover the points corresponding to the coordinates: the points may already exist (e.g. c in the example) or may need to be defined as the intersection of two existing lines.

For instance, the formula associated with $precedes(X, e, a)$ is the equation of a line passing through the coordinates assigned to e and a. The formula associated with $seg(e, X) = seg(e, a)$ is the equation of a circle with centre the coordinates of e and radius the length of $seg(e, a)$. Solving these two equations for their intersection will produce two coordinates for X, corresponding to the existing points, a and c. Only the substitution of c for X will make $precedes(X, e, a)$ true.

The formulae associated with $precedes(c, f, Y)$ and $seg(f, c) = seg(f, Y)$ may now be used in similar fashion to calculate coordinates for Y. These correspond to no existing points, but define a point which lies on the non-parallel lines ad and cf, and so may be introduced as the intersection of these two lines.

The axiom which allows us to introduce new points

$$\begin{aligned} \neg seg(A, B) \| seg(C, D) \rightarrow \quad & \exists E \; collinear(A, B, E) \wedge \\ & collinear(C, D, E) \end{aligned}$$

is known implicitly to the Geometry Machine rather than being available to extend the search tree in the normal way. Including as an explict axiom might cause problems since putting it into clausal form produces two non-Horn clauses. It is invoked only during the resurrection of continuation nodes in order to create new points and to assert the collinearity propositions on the right hand side of the axiom. In this case having checked that lines ad and cf are not parallel we can assert two new hypotheses about the new point, namely

$\rightarrow collinear(a, d, k)$ and
$\rightarrow collinear(c, f, k)$

The procedure may be summarized as follows

- Given a continuation node labelled by a goal clause, $g_1 \wedge ... \wedge g_n \rightarrow$ containing the variables, X.

- Associate algebraic formulae with each of the goals, g_i and algebraic variables with each of the variables, X.

- Solve these formulae to assign coordinates to each of the variables, X. There may be several combinations of coordinates.

- Check that these coordinates satisfy all the g_is rejecting any combinations that do not.

- Recover points to associate with the variables by noticing that the assigned coordinates are already assigned to points or by creating new points as the intersection of two non-parallel lines.

- Assert collinearity propositions for the new points.

- Substitute the points for the associated variables in the goal clause creating a new instance for each combination of coordinates.

- Let each of these instances be a daughter of the original goal clause.

10.5 Can the Diagram be Generalized?

Is this use of the diagram just a technique useful in Geometry or can it be generalized and used in other domains?

It can be generalized. The key observation is to recognise that the job of the diagram above is to define a *model* of the hypotheses of the theorem and the axioms of Geometry – in the sense that we met in chapter 3.

Firstly, note that the diagram was an of the clauses which constitute the theorem statement, i.e. the axioms of geometry and the hypothesis and conclusion of the theorem. The of the interpretation was the set of all pairs of real numbers, e.g. $< \sqrt{3}, 1 >$. Each constant in the theorem statement was assigned one of these pairs as its value, e.g. $d \mapsto < \sqrt{3}, 1 >$. Each function and predicate in the theorem statement was assigned an arithmetic calculation procedure.

Secondly, note that the assignment of coordinates to points was done so as to make this interpretation a model of the hypotheses of the theorem and the calculation procedures were chosen so as to ensure that it was a model of the axioms of Geometry.

To see how the same technique may be used in a different domain, consider the following example from arithmetic.

Let $X \nmid Y$ mean that X does not divide Y exactly. We will try to prove the theorem that

> If 5 does not divide a number then 30 does not divide it.

We will take as our axioms,

$1. X \nmid Z \to X.Y \nmid Z$
$2. Y \nmid Z \to X.Y \nmid Z$
$3. \to 30 = 2 \cdot (3 \cdot 5)$

and the equality axioms, which we will build into the paramodulation rule.

The theorem is formalized as:

hypothesis: $\to 5 \nmid a$
and
conclusion: $30 \nmid a \to$

The search tree for this example, taking the conclusion as top clause, is given in figure 10.5. This tree, though exhausting the possibilities, is nice and small. However, if we had had some additional axioms for $\nmid$, e.g.

$3. Z \nmid X \wedge Z \nmid Y \to X \nmid Y,$

then the goal clauses marked \$ and + may have been developed indefinitely and fruitlessly. It is worth looking at ways in which they could have been recognised as hopeless and pruned from the tree.

Let us use the calculation procedures of arithmetic to build an interpretation which is a model of the axioms and hypotheses of the theorem, but in which the goals of \$ or + are false. One such interpretation, $arith2$, can be defined by letting $\cdot$, $\nmid$ and $=$ have their normal meaning, but giving 'a' the value 2. In $arith2$, $5 \nmid a$, for instance, will be true, and so will $30 \nmid a$, but $2 \nmid a$ will be false. Thus $arith2$ rejects the goal clause \$.

Figure 10.5: Search Tree for Not Divides Example

However, $arith2$ will not reject +, since it makes the goal, $3 \nmid a$, true. To reject + we will need an interpretation which makes $3 \nmid a$ false. Consider, for instance, the interpretation, $arith3$, in which $\cdot$, $\nmid$ and $=$ have their normal meanings, but 'a' has value 3. This has the desired effect. It rejects +. However, it does not reject \$. Thus to reject both \$ and + we will need to use both $arith2$ and $arith3$ in concert.

This example has two messages:

- The use of the diagram by the Geometry Machine to prune some goal clauses from the search tree can be generalized to other domains: the diagram being generalized to an interpretation of the clauses and a model of the axioms and hypotheses.

- It can also be generalized in another way, namely several interpretations can be used in concert, any one having the right of veto.

Exercise 32 *Can you find an interpretation which is a model of the axioms and hypothesis, but which rejects both \$ and + ?*

Exercise 33 *Consider the search tree for the group theory example in figure 7.1, chapter 7. Can you design a model of the axioms and hypothesis of the theorem which would reject the goal clause* $a = b \rightarrow$ *?*

10.6 The Trouble with Non-Horn Clauses

So far we have argued by example (an unsafe method). Before we proceed to a theoretical analysis let us look at one more example – a counterexample.

Consider another funny arithmetic example. The proposition, $is\_prod\_primes(X)$, means X is a product of prime numbers, $is\_prime(X)$ means X is a prime number and $is\_power\_3(X)$ means X is a power of 3. As axioms we will have:

Figure 10.6: A non-Horn clause search tree

1. $is\_prime(X) \rightarrow is\_prod\_primes(X)$
   and
2. $is\_power\_3(X) \rightarrow is\_prod\_primes(X)$

i.e. a number is a product of primes if it is a prime or a power of 3.

We will now try to prove this as a theorem. Negated, this becomes the hypothesis,

hypothesis: $\rightarrow is\_prime(a) \vee is\_power\_3(a)$

and the conclusion,

conclusion: $is\_prod\_primes(a) \rightarrow$.

Notice that the hypothesis is a non-Horn clause.

If we take the conclusion as the top clause and apply Lush Resolution we get the complete search tree shown in figure 10.6. Notice also the non-Horn clause features of this search tree:

- that the clauses labelling some nodes are not goal clauses, even though the top clause was;
- that it is necessary to use with the top clause.

I am going to present some interpretations which, if allowed right of veto, would prune all proofs from this search tree – that is, the use of semantic checking will make the theorem proving process incomplete!

As usual we will let the predicates: is_prime, is_power_3 and is_prod_primes, take their obvious meanings on the natural numbers, 0, 1, 2, 3, ...etc. Thus $is\_prime(2)$, $is\_power\_3(9)$, $is\_prod\_primes(2)$ and $is\_prod\_primes(9)$ will be true and $is\_prime(9)$ and $is\_power\_3(2)$ will be false. So any interpretations we consider will be models of the axioms, 1 and 2. The various interpretations will differ only in the meaning assigned to the constant

a. In order that they will be models of the hypothesis, we must ensure that 'a' behaves like a prime or a power of 3.

In the first interpretation, $num2$, we will give 'a' the value 2. Thus, $is\_power\_3(a)$ will be false so $num2$ will reject the goal clause +.

In the second interpretation, $num9$, we will give 'a' the value 9. Thus, $is\_prime(a)$ will be false, so $num9$ will reject the goal clause \$.

If both $num2$ and $num9$ are used in concert with right of veto then the search tree will be truncated after the first level and both the proofs will be lost.

The solution might appear to be to drop the right of veto, and insist that both interpretations agree, or to use only one interpretation. However, neither of these possible solutions is good enough. We can find a single interpretation which will prune both branches.

Consider the interpretation, $num3$, in which 'a' is given the value 3. This will reject both clauses \$\$ and ++, provided we extend the rejection criteria in the obvious way to non-goal clauses. Clause ++ is $\rightarrow is\_prime(a)$ This is equivalent to $\neg is\_prime(a) \rightarrow$, i.e it behaves like a goal clause with goal $\neg is\_prime(a)$. But $is\_prime(a)$ is true in $num3$, so $\neg is\_prime(a)$ is false and $num3$ will reject ++. Similarly $num3$ will reject \$\$.

What went wrong? Why did our nice semantic based guidance technique go sour? Could it be something to do with the fact that this example is a non-Horn clause? To find out we will have to take the theoretical excursion promised at the beginning of this section.

10.7 The Theoretical Underpinning for Semantic Checking

Let us start by considering, and tidying up, the grounds on which a clause can pass the 'truth test' set by an interpretation.

On page 130 above we said that an instance of a goal clause

$$G'_1 \wedge ... \wedge G'_n \rightarrow,$$

would pass the truth test if and only if all the G'_is were true in the interpretation. This goal clause instance is equivalent to:

$$\neg G'_1 \vee ... \vee \neg G'_n$$

so if all of the G'_is are true then the whole clause is *false*. Any instance of the goal clause which is false in the interpretation will be accepted and any true instance will be rejected.

Let us extend these grounds to any variable free clause and say that

Definition 10.1 *An interpretation accepts a variable free clause if and only if it is false in the interpretation.*

One effect of semantic checking is to ensure that any clause, which is true in a model of the axioms and the hypothesis of the conjecture, is pruned from the tree. We will show that this cannot lead to incompleteness by proving the following theorem.

Theorem 10.2 *If input$\cup\{top\}$ is a minimally unsatisfiable set of Horn clauses then there is a derivation of the from them in which 'top' is the top clause and no clause in the derivation is true in any model of 'input'.*

Proof: We know from chapter 16.5 that Lush Resolution is complete for sets of Horn clauses, so there is a derivation in which top is the top clause and each clause in the derivation is derived by resolution with a clause from input. We will show that this derivation has the required property.

Let M be a model of input. Since input U –top" is unsatisfiable, top is not true in M. Since the empty clause is not true in any model it is not true in M. We will show that if a resolvant is not true in M then one of its parents is not true in M, and hence by induction on the structure of the derivation, all the clauses in it are not true in M.

To simplify matters consider only a binary resolution between:

$C' \vee P'$ and $C'' \vee \neg P''$

producing the resolvant

$(C' \vee C'')\phi$

where ϕ is the most general unifier of P' and P''. Suppose that $C' \vee P'$ and $C'' \vee \neg P''$ are truein M and that $(C' \vee C'')\phi$ is not.

Then there is an instance of the resolvant $(C' \vee C'')\phi$ which is false in M. Hence both $C'\phi$ and $C''\phi$ are false in M. Since the original clauses are true then all their instances are true, so $P'\phi$ and $\neg P''\phi$ must both be true. But $P'\phi$ and $P''\phi$ are identical, so this is a contradiction. Therefore in order to produce a resolvant which is not true one of the its parents must itself be not true. This argument is easily extended to full resolution.

The bottom clause of the derivation is the empty clause, which is not true in M.

Assume, as an induction hypothesis, that the clause nth from the bottom in the derivation is not true in M. We will show that the $n+1$st from the bottom is also not true in M and hence, by induction that none of the clauses in the derivation are true in M.

The nth clause was derived by resolution from the $n+1$st clause and a clause from input. By assumption the input clause is true in M and by the induction hypothesis the nth clause is not true in M. Hence, by the above argument the $n+1$st clause is not true in M. Therefore, none of the clauses in the derivation are true in M. QED

This theorem partially justifies the pruning of semantic checking. It shows that the clauses which were completely pruned, the ones true in some model of the input clauses, could not have led to any proofs and were rightly ignored. If we add to Lush Resolution an instantiation rule which instantiates variables with elements from the universe and show that completeness is preserved if after this rule is applied no resolutions are necessary with the parent clause then the theorem completely justifies semantic checking.

However, the theorem breaks down for non-Horn clauses at the induction step. Because ancestor resolution is allowed, the nth clause may have been derived from the $n+1$st clause and an ancestor, say the $n+m$th clause from the bottom. It could be that the $n+1$st clause is true, but that the $n+m$th clause is not true and that the nth clause inherits its non-truth from that. Thus truth can be introduced into the derivation only to be eliminated by ancestor resolution.

Exercise 34 *Verify that that is what happens with the non-Horn clause example of section 10.6.*

10.8 Summary

Thus the situation can be summarized as follows. For Horn clauses:

1. a model of the input clauses (axioms plus conjecture hypotheses) can be used to prune the search space of true clauses without loss of completeness;
2. several models can be used, with each having the right of veto.
3. in the case of Geometry the model can be thought of as a diagram, but semantic checking is applicable to any domain;
4. in Geometry, a blocked search tree can be used in conjunction with the diagram to suggest the construction of new points and lines.

It may be possible to adapt semantic checking to non-Horn clauses. For instance, by insisting that if a true clause is introduced attempts are made to ancestor resolve it or its descendants with a non-true ancestor. To the best of my knowledge no one has attempted this.

Further Reading Suggestions

[25, 24] are the best introduction to the Geometry Machine. [27] is an analysis and rational reconstruction of Gelernter's techniques.

Chapter 11

The Productive Use of Failure

- This chapter shows how evidence from the failure of one proof method, namely Symbolic Evaluation, can be used to guide other methods, namely Induction and Generalization.
- Section 11.1 defines LISP as a Predicate Logic theory.
- Section 11.2 describes the Symbolic Evaluation method of proof.
- Section 11.3 describes the Induction method.
- Section 11.4 describes the Generalization method.
- Section 11.5 explains how the Boyer-Moore technique can be applied to Peano Arithmetic.

The uniform search strategies we have considered so far, e.g. depth first search, have involved giving up in one part of the search tree and continuing in another part, without any effort to investigate why the earlier attempt failed. This contrasts with the behaviour of human mathematicians, whose later attempts may be heavily influenced by their earlier failures (cf, for instance, the accounts of [22, 38] mentioned in section 1.3.2).

By observing their own proofs and those of others, Bob Boyer and J. Moore noticed that this was especially true when proving theorems about functions. Two basic proof methods are available: a simple one in which derived from the s are used to symbolically evaluate the theorem to be proved and a more complex one in which is used to divide the theorem into two simpler theorems. Boyer and Moore observed that the failure of the first method could often suggest what induction schema to use and what variable to induct on, in the second method.

They tested these ideas by building a theorem prover for proving the properties of computer programs, using Symbolic Evaluation and mathematical induction. The theorem prover, universally known as the *Boyer-Moore Theorem Prover*, [8], has been under continuous development since 1971. A more recent version of the system is described in [9]. This chapter is a rational reconstruction of the program, paying particular attention to how the failure of Symbolic

Evaluation suggests how to use induction. It is based mainly on early versions of the program, since these are simpler to explain, while containing the key ideas.

The domain of the Boyer-Moore theorem prover was a particular programming language, LISP, [45]. We can regard it as a mathematical theory of the properties of s in which the terms of the theory also have an an interpretation as computer programs. Their techniques will work on any theory with recursive definitions and induction, e.g. Peano arithmetic. However, LISP has a richer collection of induction schemata than arithmetic, and so provides a better illustration of power of the method.

11.1 The Formal Theory of LISP

As defined in section 4.3.3 a list is an ordered set of objects, represented as a series separated by commas and delimited by angle brackets, e.g.

$$\langle this, is, a, list \rangle.$$

The elements of the list can be constants or other lists, e.g.

$$\langle this, is, \langle a, nested, list \rangle \rangle.$$

The empty list is represented by the constant nil.

The first list manipulation function we will consider is the constructor function, *cons*. cons is a binary function, whose second parameter must be a list. Its effect is to join its first parameter onto the front of this list, e.g.

$$cons(this, \langle is, a, list \rangle) = \langle this, is, a, list \rangle$$
and
$$cons(\langle first, element \rangle, \langle of, a, list \rangle) = \langle \langle first, element \rangle, of, a, list \rangle$$

cons plays a similar role, in LISP, to that played by s (the successor function) in Peano arithmetic. Just as s was used to generate all the natural numbers from the single constant 0, so cons can be used to generate all lists from a small collection of constants, e.g. $\langle this, is, a, list \rangle$ can be represented by

$$cons(this, cons(is, cons(a, cons(list, nil)))).$$

As with the successor function, this is a theoretically useful representation, but is rather cumbersome in practice, so we will maintain the fiction that the angle bracket representation is only an abbreviation for the underlying cons representation. This is just like the situation in Peano arithmetic, where 3 was regarded as an abbreviation of $s(s(s(0)))$.

We will take this process a stage further by limiting the constants we will allow to one, nil. All other constants will be regarded as an abbreviation for some combination of cons and nil. For instance, the natural numbers, 0,1,2,3,... etc, will be represented by the lists, nil, $\langle nil \rangle$, $\langle nil, nil \rangle$, $\langle nil, nil, nil \rangle$, ... etc. We will find it convenient to have, as terms within the theory, two symbols which behave rather like truth values. We will also use the symbols tt and ff to denote these, in order to distinguish them from t and f. We will adopt the convention that tt and ff are abbreviations for the lists $cons(nil, nil)$ and nil, respectively.

Exercise 35 *Translate the lists,* $\langle nil \rangle$ *and* $\langle nil, nil, nil \rangle$ *into combinations of cons and nil.*

In Peano arithmetic we had an axiom $\neg 0 = s(X)$ to ensure that no amount of applying the successor function brought us back to 0. In LISP we will want a similar axiom to ensure that no amount of consing can get us back to the empty list.

1) $\neg nil = cons(X, Y)$

As duals of the constructor function, *cons*, we will have the destructor functions, *car* and *cdr*. The car of a list is the first element, e.g.

$car(\langle this, is, a, list \rangle) = this.$

The *cdr* of a list is the list minus its first element, e.g.

$cdr(\langle this, is, a, list \rangle) = \langle is, a, list \rangle.$

The relationship between *car*, *cdr* and *cons*, can be summarized in the axioms.

2) $car(cons(X, Y)) = X$ and
3) $cdr(cons(X, Y)) = Y$

In order to define new functions by cases we will need a conditional function, *cond*. *cond* is a ternary function. If the first parameter of *cond* is ff then it equals its third parameter, otherwise it equals its second. This can be expressed by the two axioms.

4) $cond(ff, U, V) = V$ and
5) $cond(cons(X, Y), U, V) = U$

Since LISP contains an equality predicate, *equal*, between lists we will have to represent this in our theory. Following our convention that all LISP programs are to be represented as terms, we must represent equal as a function. This distinguishes it from the predicate, =, which we have been using above. Indeed we have axioms which involve them both, like

6) $equal(X, X) = tt$

Finally, we are allowed to introduce as many new functions as we like, provided they are accompanied by new axioms which serve to define them either explicitly or by recursion. For instance, we may define a function, append, by recursion, as follows.

$$\begin{aligned} append(X, Y) = cond(&X, \\ &cons(car(X), append(cdr(X), Y)), \\ &Y) \end{aligned}$$

append is a function which takes two lists and joins them, e.g.

$append(\langle this, is \rangle, \langle a, list \rangle) = \langle this, is, a, list \rangle)$

The idea of the definition is that if the first parameter is the empty list then we return the second parameter, otherwise we take off its first element, recursively append this foreshortened list to the second parameter and then cons back the first element. Note that the use of cond enables us to dispense with the need for multiple equations when making recursive definitions. This will have consequences for the way in which the failure of Symbolic Evaluation influences the subsequent induction process.

11.2 Symbolic Evaluation

A typical conjecture which we might want to prove in list theory is that appending a *nil* onto a list has no effect, i.e.

$$equal(append(nil, X), X) = tt \tag{i}$$

Taking the universal closure of this, negating it and putting it in clausal form gives:

$$equal(append(nil, x), x) = tt \rightarrow$$

The empty clause can be derived from this using alone, i.e. by rewriting the expression in bold face into tt, yielding:

$$tt = tt \rightarrow$$

and then, using the reflexive law, into $t \rightarrow$ which is equivalent to the empty clause.

All the theorems we will prove in this chapter will normalize into the form:

$$exp = tt \rightarrow$$

and their proofs will only differ in the manner in which exp is Symbolically Evaluated to tt, so this is the only part we will mention. In the case of conjecture (i) this means considering how

$$equal(append(nil, x), x)$$

can be Symbolically Evaluated to tt.

In this particular proof we will rewrite the expression using the rules:

$$\begin{array}{ll}
append(X, Y) \Rightarrow cond(X, & \\
\qquad\qquad cons(car(X), append(cdr(X), Y)), & \\
\qquad\qquad Y) & \\
cond(ff, U, V) \Rightarrow V & \\
equal(X, X) \quad \Rightarrow tt &
\end{array}$$

in that order.

Note that, as in the case of arithmetic evaluation in section 9.2.3, the first rewrite rule is based on the recursive defining equation of append. The remaining rules are based on the non-recursive definitions of $cond$ and $equal$. In general, we will use only rewrite rules based on the recursive definitions of new functions like append and those based on the axioms 1) to 6).

Thus the proof is:

$$equal(\underline{append(nil, x), x)}$$

$$\begin{aligned}
&= equal(\underline{cond(}\underline{nil,} \\
&\qquad\qquad \underline{cons(car(nil), append(cdr(nil), x)),} \\
&\qquad\qquad \underline{x),} \\
&\qquad x) \\
&= \underline{equal(x, x)} \\
&= \underline{tt}
\end{aligned}$$

where the subterm being rewritten at each stage has been emphasised by underlining.

But Symbolic Evaluation does not always work so well. The rewrite rule set defined by the axioms 1- 6 and recursive definitions is not terminating. For instance, a non-terminating branch can be obtained by applying the definition of append, repeatedly, to the term, $append(a, b)$.

$$
\begin{aligned}
\underline{append(a,b)} = {} & cond(a, \\
& \qquad cons(car(a), \underline{append(cdr(a), b)}), \\
& \qquad b) \\
= {} & cond(a, \\
& \qquad cons(car(a), \\
& \qquad\qquad cond(cdr(a), \\
& \qquad\qquad\qquad cons(car(cdr(a)), \\
& \qquad\qquad\qquad\qquad \underline{append(cdr(cdr(a)), b)}), \\
& \qquad\qquad\qquad b), \\
& \qquad b) \\
= {} & \text{.......... (and so on)}
\end{aligned}
$$

Note that each successive appearance of append has a more deeply nested first parameter: a, $cdr(a)$, $cdr(cdr(a))$, ... etc. This reflects the definition of append, where the first element of the first parameter was repeatedly removed until the empty list was revealed. But a Skolem constant like 'a' represents an arbitrary list – there is no knowing how long it is. Consequently, the process will never terminate. The loss of termination is due to our use of cond to define recursive functions in one equation. If we had used two equations, as in section 9.2.3, e.g.

$$
\begin{aligned}
& append(nil, Y) && \Rightarrow Y \\
& append(cons(X1, X), Y) && \Rightarrow cons(X1, append(X, Y))
\end{aligned}
$$

then Symbolic Evaluation would terminate with a non-specific list.

If the result of rewriting an expression is that *car* or *cdr* is applied to one of its parameters then two cases arise:

- either the dominant function of the parameter is cons, in which case the *car* or *cdr* can be eliminated by axiom 2) or 3)
- or the dominant function is something else, in which case the *car* or *cdr* cannot be eliminated.

This second case is the major cause of non-termination. We will call such a parameter an *ugly expression*. Quite often these ugly expressions are Skolem constants, as the example above illustrates. If a rewriting threatens to produce an ugly expression then the rewriting is prohibited and Symbolic Evaluation is terminated. We will see that the ugly expressions not only tell us when to stop Symbolic Evaluation, but also provide a strong clue about what to do next.

11.3 The Method of Induction

What to do next is to try the other available method, . But here we are faced with a choice. There are several induction schemata available, and each of them can be used to induct on a different parameter.

Here are a selection of some of the induction schemata available in list theory.

$$P(nil) \wedge \forall X\ \forall X1\ [P(X) \rightarrow P(cons(X1,X))] \rightarrow \forall X\ P(X)$$
$$P(nil) \wedge \forall X\ \forall X1\ [P(X1) \wedge P(X) \rightarrow P(cons(X1,X)]$$
$$\rightarrow \forall X\ P(X)$$
$$P(nil) \wedge \forall X\ P(cons(X,nil)) \wedge$$
$$\forall X\ \forall X1 \forall X2\ [P(X) \rightarrow P(cons(X1, cons(X2,X)))]$$
$$\rightarrow \forall X\ P(X)$$

In the first one induction is on the *cdr* of the list; in the second it is on the *car* and in the third it is on both.

In each of these the role of X may be taken by any Skolem constant in the formulae being proved, e.g. either a or b above, making 6 possibilities altogether. And things do not stop there, as other induction schemata may be used, in which, for instance, the induction hypothesis covers all sublists of X.

This is where the failed attempt at Symbolic Evaluation comes in. Remember that in the example above the first rewriting produced $append(cdr(a),b)$. If we had assumptions about this formula we might have been able to do something more than get involved in an infinite regress. But such an assumption is precisely what an induction hypothesis offers us, provided we choose the right one. In this case since only the Skolem constant 'a' is being affected, and since it is its cdr that is involved then the first induction schema above is strongly suggested with 'a' substituted for X.

Let us see how this might work in the case of a real theorem proving attempt. Consider the conjecture, that append is an associative function, represented by:

$$equal(append(A, append(B,C)), append(append(A,B),C)) = tt$$

If we try to prove this by Symbolic Evaluation, i.e. by rewriting

$$equal(append(a, append(b,c)), append(append(a,b),c))$$

into tt, then we will generate four branches – one for each occurrence of *append*.

Each of these will be terminated by the threatened application of a destructor function, *car* or *cdr*, to an ugly expression.

Looking back at our attempt to Symbolically Evaluate $append(a,b)$ we can see that in each of the four cases the ugly expression will be the parameter of an occurrence of *cdr* dominated by the function, *append*. The ugly expressions involved will be two occurrences of a, one of b and one of $append(a,b)$. This is diagrammed in figure 11.1

The *cdr*s all suggest the first induction schema above, i.e. induction on the *cdr* of a list. But what list? The candidates are a and b, and as we will see below, it is possible that $append(a,b)$ could be generalized into a constant and become a candidate. Note that both occurrences of a give rise to candidates, whereas only one occurrence of b does. If b where chosen as induction variable then the fact that only one of its occurrences would be unpacked by Symbolic Evaluation would cause trouble during the induction step. An induction with b as variable is called a *flawed induction* and one with a as variable is called an *unflawed induction*. The Boyer/Moore theorem prover would choose a as the induction variable.

Thus the induction basis is to prove.

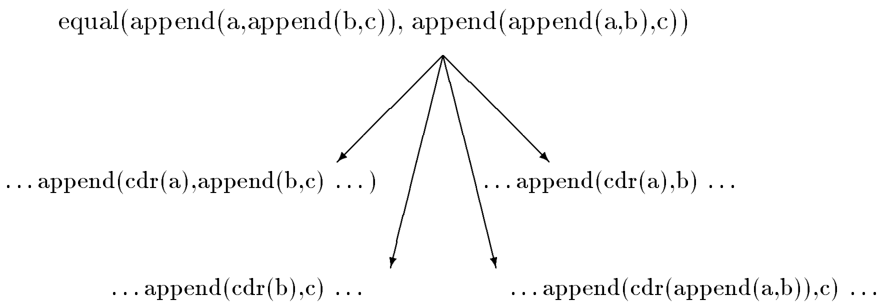

Figure 11.1: Potential Infinite Regresses in Rewriting of append Example

$$\begin{aligned} equal(\ & append(nil, append(b,c)),\\ & append(append(nil,b),c)) = tt \rightarrow \end{aligned}$$

which yields easily to Symbolic Evaluation (Note how similar it is to the example we did above). The induction step is to assume the hypothesis

$$\rightarrow equal(append(a, append(b,c)), append(append(a,b),c)) = tt$$

and then prove

$$\begin{aligned} equal(&\underline{append(cons(a1,a), append(b,c))},\\ & append(\underline{append(cons(a1,a),b)},c)) = tt \rightarrow \end{aligned}$$

The left hand side of this can be rewritten to

$$\begin{aligned} equal(&cons(a1, \underline{append(a, append(b,c))}),\\ & cons(a1, append(append(a,b),c))) \end{aligned}$$

Now the induction hypothesis can be applied, as if it were a rewrite rule, to replace the left hand side double append and get

$$\begin{aligned} \underline{equal(}\ & \underline{cons(a1, append(append(a,b),c))},\\ & \underline{cons(a1, append(append(a,b),c)))} \end{aligned}$$

which is trivially rewritten to tt.

The step where the induction hypothesis was used was not the same kind of rewriting step as the others because the induction hypothesis had the form $equal(lhs, rhs) = tt$, rather than $lhs = rhs$. Boyer and Moore call this an application of *Fertilization.*

In summary then, the Boyer-Moore technique is

1. Try to prove the theorem by Symbolic Evaluation.
2. Trap any infinite branches by looking out for a destructor function applied to an ugly expression parameter of the subterm about to be rewritten.
3. If the attempt at Symbolic Evaluation fails try Induction.
4. Use the trapped destructor function(s) to suggest the induction schema to use and the trapped ugly expression(s) to suggest what should be substituted for the induction variable, prefering those that lead to unflawed inductions.

5. Try to prove the induction basis and conclusion. Use Fertilization during the proof of the induction step.

11.4 Generalizing the Theorem to be Proved

It is often the case in proving theorems by Induction that the induction hypothesis is not strong enough to support the induction conclusion. When this happens it may be that we should try a different induction schema or it may be that the theorem should first be strengthened by generalizing it – a paradox of proofs by Induction is that it sometimes easier to prove a strong theorem than a weak one, simply because the induction hypothesis is also stronger. This is one of the major points made in [22].

This is the case with the theorem

$$\begin{aligned}&equal(append(\underline{rev(A)}, append(B,C)),\\ &\qquad append(\overline{append}(\underline{rev(A)}, B), C)) = tt\end{aligned}$$

where rev is a function which returns a list in the reverse order, e.g.

$$rev(\langle this, is, a, list\rangle) = \langle list, a, is, this\rangle$$

The induction hypothesis is just not strong enough to do its job and the Boyer-Moore technique above, fails.

However, we have already seen that a stronger version of the theorem can be easily proved, namely the theorem obtained by replacing $rev(A)$ above by a variable, say D.

$$\begin{aligned}&equal(append(\underline{D}, append(B,C)),\\ &\qquad append(append(\underline{D}, B), C)) = tt\end{aligned}$$

This replacement of a term by a variable is called *Generalization*. It can be done before Induction is attempted to improve the chances of the attempt succeeding. An obvious procedure suggests itself, namely to look for two or more identical, complex subterms (i.e. not variables or constants) in a formula and replace them with new variable. Thus the subterm, $append(b,c)$, in the theorem above would not be replaced because it only occurs once and b would not be replaced because it is a constant.

A further refinement (due to Aubin [1]) is to insist that the term to be replaced should be one of the trapped ugly expressions. The reasoning behind this is that for Generalization to achieve anything it should make available an induction variable candidate that was not available before. Induction variable candidates must all show up as trapped ugly expressions, but those that are complex terms cannot be used as they stand. Generalizing them to variables would make them available. This works in the example above, as $rev(A)$ is trapped twice as an ugly expression. Thus the failure of Symbolic Evaluation can not only suggest how to apply Induction, but also how to Generalize the conjecture to be proved.

The danger in Generalization is to over-generalize, i.e. to generalize a theorem into a non-theorem. An example of this is provided by the theorem,

$$equal(length(\underline{length(X)}), \underline{length(X)}) = tt$$

where $length$ is a function from lists to numbers, which in our formal theory are played by lists of $nils$. Thus

$$\begin{aligned}&length(length(\langle this, is, a, list\rangle))\\&\qquad equals\ length(\langle nil, nil, nil, nil\rangle)\\&\qquad equals \langle nil, nil, nil, nil\rangle\\&\qquad equals\ length(\langle this, is, a, list\rangle)\end{aligned}$$

This theorem contains two occurrences of the complex term, $length(X)$, one of which will be trapped as an ugly expression during the Symbolic Evaluation of $length(length(X))$. If these are replaced by a new variable, say Y, we get the formula

$$equal(length(Y), Y) = tt \tag{ii}$$

which is no longer a theorem! In fact (ii) is only true when Y is a number, i.e. a list of $nils$.

Boyer and Moore provided a partial answer to this problem of over-generalization by having their theorem prover attempt to invent an additional hypothesis about the newly introduced constant which would retain the truth of the theorem. In this case the theorem prover would augment ii with an additional hypothesis to the Generalization:

$$is\_number(Y) = tt \rightarrow equal(length(Y), Y) = tt$$

where $is\_number(Y)$ means Y is a list of $nils$. This formula is a theorem.

For this additional hypothesis to be meaningful any functions in it would have to be defined. In this case additional axioms would have to be asserted to define is_number. The theorem prover was able to do this in many cases, merely by examining the replaced subterm. In this case by examining the term $length(Y)$ to see what kind of list it might be.

In inventing these axioms the Boyer-Moore Theorem prover was nearly always successful, but sometimes it would over-generalize and the additional hypothesis would admit too many lists. This is why the solution was only partial. However, if the Generalized theorem could be proved, then the original theorem would follow from it.

Another way to avoid over-generalization (again due to Aubin), is to further refine the heuristic for suggesting Generalization candidates. Instead of just noting complex trapped ugly expressions as Generalization candidates, we could note which occurrences of the expression are trapped. When Symbolically Evaluating

$$equal(length(length(X)), length(X)) = tt$$

the first occurrence of $length(X)$ is trapped, but the second is not. This observation can be used to block the Generalization of $length(X)$, since the induction candidate should appear on both sides of the equality to be effective.

11.5 Applications to Arithmetic

The Boyer/Moore technique is applicable to any theory in which functions are defined recursively and theorems proved by Induction. In this section we show how this can be done to the most well known such theory, Peano arithmetic.

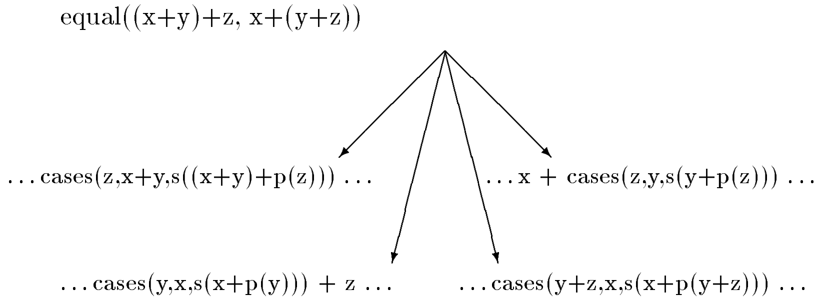

Figure 11.2: Potential Infinite Regresses in Rewriting of + Example

In arithmetic, the role of the recursive data structure, lists, is played by the natural numbers, 0, $s(0)$, $s(s(0))$, ... etc. The successor function, s, plays the part of the constructor function, *cons*, and the predecessor function, p, plays the part of the destructors, *car* and *cdr*, where

$$p(s(X)) = X \qquad \text{and} \qquad p(0) = 0$$

We need to introduce a new function, *cases*, to play the part of *cond*, i.e.

$$cases(0, Y, Z) = Y \qquad \text{and}$$
$$cases(s(X), Y, Z) = Z$$

which returns its second or third parameter, according to whether its first parameter is 0 or not. These equations defining p and *cases* must be used by Symbolic Evaluation, as left to right rewrite rules.

To use the Boyer/Moore technique as it stands, we must modify the recursive definitions of the arithmetic functions to be single equations using the cases function, e.g.

$$X + Y = cases(Y, X, s(X + p(Y))$$

Compare this with the definition of + given in section 4.2.3. We can use this definition as a rewrite rule, left to right, during Symbolic Evaluation.

We are now in a position to consider the proof of the associativity of +.

$$(X + Y) + Z = X + (Y + Z)$$

To be proved by the Boyer/Moore technique this conjecture must be represented as the goal clause.

$$equal((x + y) + z, x + (y + z)) = tt$$

Symbolic Evaluation of this goal clause, using the definition of +, yields the four terminated branches diagrammed in figure 11.2. In each of the four cases the trapped ugly expressions are parameters of the predecessor function, p, dominated by the + function. The four ugly expressions are: z, y, $y + z$ and z. The complex term, $y + z$, would be a candidate for Generalization, if it appeared more than once, but it is not yet a candidate for induction variable. Of the remaining candidates, y and z, z wins by majority vote.

Since the trapped occurrences of z both appear as parameters of p then the suggested induction scheme is the standard one,

$$Q(0) \wedge \forall X \{Q(X) \rightarrow Q(s(X))\} \rightarrow \forall X\, Q(X)$$

Applying this to our theorem generates the two subgoals:

Basis
$equal((x + y) + 0, x + (y + 0)) = tt \rightarrow$
and
Step
$\rightarrow equal((x + y) + z, x + (y + z)) = tt$
$equal((x + y) + s(z), x + (y + s(z))) = tt \rightarrow$

The basis rapidly yields to Symbolic Evaluation. The step can be proved by Symbolic Evaluation plus Fertilization with the induction hypothesis. The proof is strongly parallel to that of the associativity of append, given earlier.

Exercise 36 *Describe the proof of the step above, as it might be found by the Boyer-Moore technique.*

Exercise 37 *Describe how the Boyer-Moore technique would prove the conjecture,* $(X + Y) + Z^2 = X + (Y + Z^2)$

Thus the Boyer-Moore technique can be applied to theories other than LISP.

11.6 Summary

The Boyer-Moore Theorem Prover has available a variety of methods of proving theorems: Symbolic Evaluation; Induction; Fertilization and Generalization. It understands something of the relation between these methods. It knows when to apply each one and when the failure of one method can help in the application of another. In short, it can reason about the different methods available. We will take this idea a step further in the next chapter.

Further Reading Suggestions

[8] is a short and readable introduction to the Boyer-Moore theorem prover, although it is a little out of date. [9] is a longer and more up to date account. [49] gives a detailed account of the implementation of the original program.

Chapter 12

Formalizing Control Information

- In this chapter we see how to represent control information as a Predicate Logic theory and use inference in this theory to guide search.
- Section 12.1 introduces some control information that can be used to guide the search for a solution to an equation.
- Section 12.2 describes various equation solving solution methods: Isolation, Collection and Attraction.
- Section 12.3 defines the Meta-Theory of Algebra and shows how to represent control information with it and how to use this information to guide the search for a solution.

When a human mathematician has built up expertise, in a particular area of mathematics, he has at his disposal a variety of problem solving methods. He is able to bring these methods to bear on a problem, choosing an appropriate method for the goal he is trying to achieve and the situation in which he finds himself.

We saw, in the last chapter, some examples of such methods: symbolic evaluation, induction, generalization, etc. In this chapter we take this investigation a stage further and ask how such methods should be organised, so that they may be brought smoothly into operation when the occasion demands and be combined together to tackle complex problems. The answer we offer is: that a process of reasoning should take place about the problem to be solved and the methods available to solve it; and that the techniques we have met in earlier chapters can be used to represent this reasoning process. It is precisely such reasoning processes which are advocated by Polya in the books mentioned in section 1.3.2.

The richest examples of multiple problem solving methods can be found in some of the older branches of mathematics, where experience has accumulated over several centuries. Consider, for instance, the problem of solving symbolic, transcendental equations, like

$$\log_e(x+1) + \log_e(x-1) = c$$

There are a variety of well known methods to deal with subclasses of equations, e.g. Gaussian elimination for sets of simultaneous linear equations, completing the square for quadratic equations and some unnamed, but well known, methods for trigonometric equations.

But what about those equations, like the one above, for which there are no established methods? Most mathematicians would find the above example quite easy – they would not indulge in search, but would proceed directly to a solution. This behaviour cannot be accounted for using the theorem proving techniques we have considered so far. This equation and the axioms of algebra, will create a huge . Searching this tree in an unguided fashion will lead to a . So how do they do it? Could it be that they have available a variety of 'methods' which are not publically available in the textbooks of algebra, but are picked up from examples – the worked examples from textbooks and the successful personal solutions of the past?

12.1 Reading Between the Lines

One way to find out is to examine some worked examples or successful personal solutions and see if there is any pattern there, i.e. to analyse the solutions. Consider the following solution of the log example above.

$$\log_e(x+1) + \log_e(x-1) = c$$

...1

$$\log_e(x+1)\cdot(x-1) = c$$

...2

$$\log_e x^2 - 1 = c$$

...3

$$x^2 - 1 = e^c$$

...4

$$x^2 = e^c + 1$$

...5

$$x = \pm\sqrt{e^c + 1}$$

In order that we can read between the lines, I have labelled the spaces between them with numbers.

The most revealing part of this solution is at the bottom, so we will start our analysis there. Consider the last three steps, labelled 3, 4 and 5. There is a pattern here. At each step the function on the left hand side is removed and its inverse becomes the dominant function on the right hand side: thus log is replaced with exponentiation; minus with plus and square with square root. The effect is to gradually decrease the depth of the unknown x, until it is totally isolated on the left hand side, with the solution on the right. This is one of the most common 'unwritten' methods of equation solving. It will work with equations composed of any sort of functions, provided each function has an inverse. We will call it *Isolation*.

Isolation will only work with equations containing a single occurrence of the unknown. If there are several occurrences, as in the first and second lines of the solution above, then one of the occurrences can be isolated, but the other occurrences will be moved to the right hand side, polluting the potential solution with their presence. Armed with this observation we can now make sense of step 2.

In step 2 the number of occurrences of x is reduced from 2 to 1, using the 'difference of two squares' identity,

$$(U + V) \cdot (U - V) = U^2 - V^2.$$

This has the effect of making the Isolation method applicable. We will assume that there is a method available for reducing the number of occurrences of an unknown. We will call it *Collection*, since it collects together occurrences of unknowns. Collection can often be seen at work in equation solving. It does not have the same guarantee of success as Isolation, but an identity can often be found to collect together two occurrences, especially if they are close together.

This last observation helps us to understand step 1. The number of occurrences of x is not reduced, but they are brought closer together, thus making it more likely that they can be collected. Again we will assume that there is a method for doing this and we will call it *Attraction*.

Exercise 38 *Consider the equation*

$$(2^{x^2})^{x^3} = 2.$$

Can you solve this using the methods of Isolation, Collection and Attraction?

12.2 Equation Solving Methods

How can Isolation, Collection and Attraction be represented as computer programs?

This becomes clearer when we see what algebraic formulae they will need to apply in order to achieve their stated purpose.

12.2.1 Isolation

For instance, Isolation applies the following double implications in the solution above.

$$\begin{array}{l} \log_U V = W \leftrightarrow V = U^W \\ U - V = W \leftrightarrow U = W + V \\ U^2 = V \leftrightarrow [U = \sqrt{V} \vee U = -\sqrt{V}] \end{array}$$

These can be organised as a set of . Isolation will apply them successively until the unknown is totally isolated, when no more rules will apply.

Notice that the rules above all have a similar structure:

- the left hand side is an equation between a term and a variable;
- the term consists of a function with variable parameters;
- the right hand side is either an equation between one of these variables and a term or is a disjunction of such equations;
- this term contains the inverse of the left hand side function.

This structure can be summarized as:

Figure 12.1: The Position of x

$F(U_1, ..., U_i, ...U_n) = V \leftrightarrow U_i = F_i^{-1}(U_1, ..., V, ...U_n)$ where U_i is the variable being isolated

It is not surprising that all the rules should have this structure as it is just what is required to achieve the Isolation effect of stripping away the dominant function of the left hand side until the unknown is isolated. It does mean that the Isolation method can be very selective about how it applies the rules under its care.

- It need only attempt to rewrite complete equations
- and only then if the distinguished variable U_i is matched to a term containing the unknown it is trying to isolate.

We will call these conditions the *selection criteria* of Isolation.

The last criterion can be met if the each rule is stored with the parameter position it isolates and the position of the unknown occurrence is calculated before Isolation starts. The position of an occurrence of a term in an expression, can be represented as a list of numbers, which determine which parameter to take when traversing the expression from the dominant function (root) to the occurrence. For instance, the position of x in $\log_e(x^2 - 1)$ is $\langle 2, 1, 1 \rangle$, since it occurs in the 2nd parameter of log, the 1st of – and the 1st of exponentiation (see figure 12.1). If $expr\_at(List, Exp)$ is the expression at position $List$ in Exp then

$$expr\_at(\langle 2, 1, 1 \rangle, \log_e(x^2 - 1)) \equiv x$$

where $\equiv$ expresses identity between algebraic expressions.

Exercise 39 *What is the position of the 1 in* $\log_e(x^2 - 1)$*?*

Exercise 40 *Using the techniques of section 9.3 show that the selective application of Isolation rules will always terminate.*

12.2.2 Collection

Similarly the Collection method can be programmed as the selective application of a set of rewrite rules. The Collection rule we used in our example was:

$$(U+V)\cdot(U-V) \Rightarrow U^2 - V^2$$

This has two occurrences of U on the left hand side and one on the right. Thus if U is matched to a term containing x then the number of occurrences of x will be reduced when the rule is applied. A similar argument holds for V. Thus this particular rule collects on both U and V. This is not true of the distributive law

$$U\cdot V + U\cdot W \Rightarrow U\cdot(V+W)$$

which collects only on U.

The criterion for identifying a Collection rule is that it should contain some distinguished variable, U, which appears more often in the left hand side than the right. Thus if $occ(Var, Exp)$ is a function whose value is the number of occurrences of Var in Exp then a rule $lhs \Rightarrow rhs$ is useful for Collection relative to U iff $occ(U, lhs) > occ(U, rhs)$.

Another well known rule with this property is,

$$2\cdot\sin U\cdot\cos U \Rightarrow \sin 2\cdot U.$$

This is not in its most generally useful form. For instance, as it stands it could not be applied to

$$3\cdot\sin x\cdot\ cosx.$$

Thus it is better to use it in the form,

$$\sin U\cdot\cos U \Rightarrow 1/2\cdot\sin 2\cdot U$$

where the smallest subterm on the left hand side, which contains all the occurrences of U, is isolated. We will call such a term, a *least dominating term* for U.

If we always arrange the Collection rules so that their left hand sides are least dominating terms in the variable they are useful for collecting then the application of the rules can be especially selective:

- Collection need only try to rewrite terms which are least dominating in the unknown, e.g. it need only rewrite $(x+1)\cdot(x-1)$ in $log_e(x+1)\cdot(x-1) = c$ (see figure 12.2);

- the distinguished variable, U, must be matched to a term containing the unknown.

12.2.3 Attraction

The Attraction method is very similar to the Collection one. The sort of rewrite rules used by Attraction are:

Figure 12.2: A Least Dominating Term in x

Figure 12.3: The Distance Between Two Terms

$$\begin{array}{ll}
\log_W U + \log_W V & \Rightarrow \log_W U \cdot V \\
W \cdot U + W \cdot V & \Rightarrow W \cdot (U + V) \\
(W^U)^V & \Rightarrow W^{U \cdot V} \\
U^{V \cdot W} & \Rightarrow (U^V)^W
\end{array}$$

In each of these the distinguished variables U and V are closer together on the right hand side than they are on the left.

What does it mean for two variables to be closer together in one place than in another? What is the notion of distance involved?

A simple measure of distance is easily defined by looking at the expression tree which contains the two occurrences. Consider figure 12.3. The shortest between U and V in each of the trees is in heavy type. In the tree of $\log_W U + \log_W V$ this path has length 4. In the tree of $\log_W U \cdot V$ it has length 2. If we define the distance between two occurrences of terms as the length of the shortest path between them then U and V are closer in the second tree than in the first.

Thus a rule, $lhs \Rightarrow rhs$, is useful for attracting U and V if both U and V occur once each in lhs and rhs and if the distance between them in rhs is smaller than the distance between them in lhs. If we ever find a rule which attracts a whole bunch of variables at once then the definition of distance can easily be generalized to measure the smallest path connecting all the variables. As with Collection we will want lhs to be a least dominating term in U and V.

The selection criteria for applying these rules are also similar to those for Collection:

- the term to be rewritten should be a least dominating term in the unknown, x;
- the distinguished variables U and V should be matched to terms containing x;
- the undistinguished variables should be matched to terms not containing x (otherwise two occurrences of x may be moved closer together only to move two others further apart).

Notice that the same formula may be used as a rewrite rule by different methods, but because of the selection criteria it will be used in different ways. For instance,

$$W \cdot U + W \cdot V = W \cdot (U + V)$$

is used by both Collection and Attraction. But whereas Collection will match W to a term containing x and reduce the number of occurrences of x, Attraction will match U and V to terms containing x and move the occurrences closer together. There is no reason why the same equation should not be used in different directions by different methods: the strong selection criteria will maintain termination.

In fact, the same formula can be used in different directions *by the same method* without loss of termination. Consider, the formula

$$U^{V \cdot W} = (U^V)^W.$$

This is used in both directions by Attraction (see above rules), yet will not cause looping since, when used left to right W must be matched to a term free of x and when used right to left W must be matched to a term containing x.

12.3 Reasoning About Problems and Methods

Now we have an informal definition of the methods used in our example solution we can proceed to formalise the reasoning which was involved there.

12.3.1 Defining the Methods with Axioms

Suppose that the reasoning involved at each stage was something like:

> Does the equation contain only a single occurrence of x? If so, Isolate it, if not then find a term which contains two occurrences of x and Collect these together. If this fails then try to Attract the two occurrences together. If any of these steps succeed then conduct the same reasoning on the resulting equation.

Let us pull out from this a particular piece of knowledge: that if an equation contains a single occurrence of the unknown then isolating this unknown will produce a solution to the equation. We have already seen how to express the information that an equation contains only a single occurrence of an unknown:

$$occ(x, \log_e x^2 - 1 = c) = 1$$

We will want to express the information that a formula is the result of applying the Isolation method to an equation to isolate an unknown. It will be convenient to do this in two bites, using the function, position, to say where the occurrence of the unknown is, e.g.

$$expr\_at(\langle 1,2,1,1\rangle, \log_e x^2 - 1 = c) \equiv x$$

and introduce the new function, *isolate*, which represents the result of isolating whatever is at a particular position in an equation.

$$isolate(\langle 1,2,1,1\rangle, \log_e x^2 - 1 = c) \equiv \\ (x = \sqrt{e^c + 1} \vee x = -\sqrt{e^c + 1})$$

Finally, we will want to be able to say that a formula is a solution, for an unknown, of an equation. Since there may be several solutions we will use a predicate, solve, and say, e.g.

$$solve(\log_e x^2 - 1 = c, x, (x = \sqrt{e^c + 1} \vee x = -\sqrt{e^c + 1}))$$

Putting all this together gives us an axiom for reasoning about problems and methods, namely

$$\begin{aligned} 1) \quad & occ(X, A = B) = 1 \wedge expr\_at(List, A) \equiv X \wedge \\ & isolate(List, A = B) \equiv Ans \rightarrow solve(A = B, X, Ans) \end{aligned}$$

To get another axiom consider how the isolate function might be defined in terms of the applications of individual Isolation rewrite rules.

We will want to express the information that one expression is derived from another by applying an Isolation rewrite rule. There is only one such derived expression for each parameter position of the original expression, so we can introduce a function from parameter positions and expressions to expressions which we will call, *isolate_rewrite*, e.g.

$$isolate\_rewrite(2, \log_e x^2 - 1 = c) \equiv x^2 - 1 = e^c.$$

The Isolation method just consists of applying an Isolation rule to an equation and then entering the method recursively. This can be captured in the axiom:

$$\begin{aligned} 2) \; & isolate\_rewrite(N, Old) \equiv New \wedge \\ & \quad isolate(Rest, New) \equiv Ans \\ & \qquad \rightarrow isolate(cons(N, Rest), Old) \equiv Ans \end{aligned}$$

since the first parameter position to be isolated will be the number on the front of the position list of the unknown.

Exercise 41 *This axiom only defines isolate when the expression Old is an equation. Give an axiom which defines isolate when Old is a disjunction of equations in terms of isolate on the disjuncts.*

In a similar way we can design axioms for Collection and Attraction. The main differences being:

Figure 12.4: Search Tree for Reasoning about Problems and Methods

- since expressions can be collected and attracted in a variety of ways we will need to use predicates where we used functions above;
- neither Collection nor Attraction are applied repeatedly, like isolate, they are applied once and the resulting equation is solved.

For instance, an analogous axiom for Collection is:

$$
\begin{aligned}
3)\ & occ(X, Old) > 1 \wedge \\
& \quad collect(Old, X, New) \wedge \\
& \quad solve(New, X, Ans) \\
& \qquad \rightarrow solve(Old, X, Ans)
\end{aligned}
$$

where $collect(Old, X, New)$ means that New is one result of collecting X in Old.

To complete the picture further axioms need to be written defining occ, $position$, $isolate\_rewrite$, $collect$, etc, but we have enough to illustrate the principle of reasoning about problems and methods.

12.3.2 Searching for a Solution

If we add a goal clause defining a particular problem

$$solve(\log_e(x+1) + \log_e(x-1) = c, x, Ans) \rightarrow$$

and use this as top clause in a Lush Resolution derivation, then we can begin to see the shape of the search tree that would be developed. It is given in figure 12.4

But how does all this help solve equations?

The answer is contained in the propositions involving $isolate\_rewrite$ (and similar predicates for Collection and Attraction) which state that such and such an expression is the outcome of applying a rewrite rule to an expression. The axioms which define these propositions define a rewriting process (see axiom 4 below). When the propositions are resolved away this rewriting process takes place as a side effect. Thus the reasoning at the level of problems and methods induces some reasoning at the level of manipulating algebraic expressions.

What is the advantage of organising things this way?

The branching rate of the tree in figure 12.4 is extremely small. Often there is no branching at all for long stretches of the tree. Useless branches get terminated quickly (as both the Collection and Isolation branches above are). The induced search at the level of algebraic manipulation, similarly involves very little search.

Thus an exhaustive search in which the axioms of algebra are applied indiscriminately to the equation to be solved is replaced by two well behaved searches running in close harness.

12.3.3 Meta Level Reasoning

What mathematical theory do the axioms 1, 2 and 3 above, belong to? It is not algebra, because the predicates solve, collect etc do not express relationships between numbers as all the algebraic predicates do, e.g. $=$, $>$ etc. solve and collect express relationships between algebraic formulae, i.e. they discuss the *representation* of algebra.

Whenever we reason about the representation of a mathematical theory we are said to be operating at the and the mathematical theory itself is said to be the . Axioms 1, 2 and 3 above are axioms of the *Meta-Theory of Algebra*. The search tree in figure 12.4 represents a *meta-level search* and the algebraic search it induces is an *object-level search*. We will call this search control technique the technique of *meta-level inference*.

Notice that object-level predicates, like $=$, and object-level connectives, like $\vee$, are represented as meta-level functions, because they have to form terms which can be the parameters of meta-level predicates. We met this situation in the last chapter, when the object-level, LISP, equal predicate was represented as a meta-level function and the object level truth values, tt and ff, as meta-level constants. This discussion puts a theoretical gloss on the informal explanation in that chapter.

In fact there are two versions of $=$, $>$, $\vee$, etc: one in the object-level and one in the meta-level. We used the same symbol to represent both versions.

Exercise 42 *Go back to section 12.3.1. Mark each occurrence of an object-level $=$ sign with an $\circ$ and each occurrence of a meta-level $=$ sign with an m.*

Object-level variables must be represented as meta-level constants. Otherwise, unification can change the truth of propositions. Consider, for instance, the true proposition:

$$occ(x, \sin U) = 0$$

If unification should apply the substitution, $\{x/U\}$, then this will instantiate this true proposition into a false one, namely

$$occ(x, \sin x) = 0$$

However, if U is represented as a constant then no instantiation can take place.

As a consequence, the rewrite rules must all be represented as variable free formulae and matching and instantiating must all be defined as meta-level functions, e.g. the Isolation rules must be recorded as variable free assertions,

$$\rightarrow isolate\_rule(1, u - v = w \Rightarrow u = w + v)$$

and the definition of isolate_rewrite must involve match and instantiate,

4) $isolate\_rule(N, Lhs \Rightarrow Rhs) \wedge$
$match(Lhs, Old) \equiv Theta \wedge$
$instantiate(Rhs, Theta) \equiv New$
$\rightarrow isolate\_rewrite(N, Old, New)$

12.4 Summary

Bob Welham has embodied the technique of meta-level inference in a program, PRESS, for solving symbolic, transcendental equations [15]. PRESS is currently being developed by a team of researchers, led by the author. This chapter is a rational reconstruction of the PRESS program. We can summarize the technique as follows.

- The problem to be solved is analysed using meta-level concepts, like occ and position.
- This analysis is used to access an appropriate method of solution, e.g. Isolation or Collection.
- The entire process can be thought of as inference at the meta-level inducing inference at the object-level.

In the case of PRESS the object-level inference consists of the application of rewrite rules to an expression.

This technique generalizes to other domains. The final formalization of the group theory example of chapter 7 was a meta-theoretic axiomatization. We designed axioms like:

$\rightarrow theorem((\text{i}))$
$\rightarrow common\_technique((\text{i}), abelianness)$
$\rightarrow proof((\text{i}), a \circ b = e \circ a \circ b = ... = b \circ a)$

These axioms could have played a part in inducing an object-level inference process, where the object-level was group theory.

In fact, the reasoning of human mathematicians is usually at the meta-level of the mathematical theory they are working on, so that this technique is often a good way to model their thought process.

When I wrote, in chapter 7, that the standard axiomatizations of mathematical theories, provided by Mathematical Logic, were not the best for automating mathematical reasoning, it was meta-theoretic axiomatizations that I had in mind as a non-standard alternative. I hope that this alternative will receive considerable attention in the future.

Further Reading Suggestions

[15] is the best general introduction to PRESS. [7, 13, 14] describe recent extensions.