

4 RECURSION AND TERMINATION

Recursion is a form of programming or definition where a newly defined notion may even occur in its *definienda*. Contrary to *explicit* definitions, where we can always get rid of the new notions by reduction (i.e. by rewriting the *definienda* (left-hand sides of the defining equations) to the *definienda* (right-hand sides)), reduction with *recursive* definitions may run forever.

We have already seen some recursive function definitions in §§ 3.4 and 3.5, such as the ones of `+`, `lessp`, `length`, and `count`, where these function symbols occurred in some of the right-hand sides of the equations of their own definitions; for instance, the function symbol `+` occurs in the right-hand side of `(+2)` in § 3.4.

4.1 Confluence

The restriction that is to be required for every recursive function definition is the *confluence*⁷¹ of the *rewrite relation* that results from reading the defining equations as reduction rules, in the sense that they allow us to replace occurrences of left-hand sides of instantiated equations with their respective right-hand sides, provided that their conditions are fulfilled.⁷²

The confluence restriction guarantees that no distinct objects of the data types can be equated by the recursive function definitions.⁷³ This restriction is essential for consistency if we assume axioms such as `(nat2-3)` (cf. § 3.4) or `(list(nat)2-3)` (cf. § 3.5). Indeed, without confluence, a definition of a recursive function could destroy the data type in the sense that the specification has no model anymore; for example, if we added `p(x) = 0` as a further defining equation to `(p1)`, then we would get `s(0) = p(s(s(0))) = 0`, in contradiction to the axiom `(nat2)` of § 3.4.

For the recursive function definitions admissible in the Boyer–Moore theorem provers, confluence results from the restrictions that there is only one (unconditional) defining equation for each new function symbol,⁷⁴ and that all variables occurring on the right-hand side of the definition also occur on the left-hand side of the defining equation.⁷⁵

These two restrictions are an immediate consequence of the general definition

⁷¹A relation \longrightarrow is *confluent* (or has the “Church–Rosser property”) if two sequences of steps with \longrightarrow , starting from the same element, can always be joined by an arbitrary number of further steps on each side; formally: $\longleftarrow^+ \circ \xrightarrow^+ \subseteq \xrightarrow^* \circ \longleftarrow^*$. Here \circ denotes the concatenation of binary relations; for the further notation see § 3.1.

⁷²For the technical meaning of *fulfilledness* in the recursive definition of the rewrite relation see [Wirth, 2009], where it is also explained why the rewrite relation respects the straightforward purely logical, model-theoretic semantics of positive/negative-conditional equation equations, provided that the given admissibility conditions are satisfied (as is the case for all our examples).

⁷³As constructor terms are irreducible w.r.t. this rewrite relation, if the application of a defined function symbol rewrites to two constructor terms, these constructor terms must be identical in case of confluence.

⁷⁴Cf. item (a) of the “definition principle” of [Boyer and Moore, 1979, p. 44f.]. Confluence is also discussed under the label “uniqueness” on Page 87ff. of [Moore, 1973].

⁷⁵Cf. item (c) of the “definition principle” of [Boyer and Moore, 1979, p. 44f.].

style of the list-programming language LISP. More precisely, recursive functions are to be defined in all Boyer–Moore theorem provers in the more restrictive style of *applicative* LISP.⁷⁶

EXAMPLE 6 (A Recursive Function Definition in Applicative LISP).

To avoid the association of routine knowledge, let us not consider a function definition over lists (as standard in LISP), but over the natural numbers. For example, instead of our two equations (+1), (+2) for +, we find the following single equation on Page 53 of the standard reference for the Boyer–Moore heuristics [Boyer and Moore, 1979]:

$$\begin{aligned} (\text{PLUS } X \ Y) = & (\text{IF } (\text{ZEROP } X) \\ & (\text{FIX } Y) \\ & (\text{ADD1 } (\text{PLUS } (\text{SUB1 } X) \ Y))) \end{aligned}$$

Note that (IF $x \ y \ z$) is nothing but the conditional “IF z then y else z ”, that ZEROP is a Boolean function checking for being zero, that (FIX Y) returns Y if Y is a natural number, and that ADD1 is the successor function s .

The primary difference to (+1), (+2) is that PLUS is defined in *destructor style* instead of the *constructor style* of our equations (+1), (+2) in § 3.4. As a constructor-style definition can always be transformed into an equivalent destructor-style definition, let us do so for our definition of + via (+1), (+2).

In place of the untyped destructor SUB1, let us use the typed destructor p defined by either by (p1) or by (p1') of § 3.4, which — just as SUB1 — returns the predecessor of a positive natural number. Now our destructor-style definition of + consists of the following two positive/negative-conditional equations:

$$\begin{aligned} (+1') \quad x + y = y & \quad \Leftarrow x = 0 \\ (+2') \quad x + y = s(p(x) + y) & \quad \Leftarrow x \neq 0 \end{aligned}$$

If we compare this definition of + to the one via the equations (+1), (+2), then we find that the constructors 0 and s have been removed from the left-hand sides of the defining equations; they are replaced with the destructor p on the right-hand side and with some conditions.

Now it is easy to see that (+1'), (+2') represent the above definition of PLUS in positive/negative-conditional equations, provided that we ignore that Boyer–Moore theorem provers have no types and no typed variables. \square

If we considered the recursive equation (+2) together with the alternative recursive equation (+2'), then we could rewrite $s(x) + y$ on the one hand with (+2) into $s(x + y)$, and, on the other hand, with (+2') into $s(p(s(x)) + y)$. This does not seem to be problematic, because the latter result can be rewritten to the former one by (p1). In general, however, confluence is undecidable and criteria sufficient for confluence are extremely hard to develop.

⁷⁶See [McCarthy *et al.*, 1965] for the definition of LISP. The “applicative” subset of LISP lacks side effects via global variables and the imperative commands of LISP, such as variants of PROG, SET, GO, and RETURN, as well as all functions or special forms that depend on the concrete allocation on the system heap, such as EQ, RPLACA, and RPLACD, which can be used in LISP to realize circular structures or to save space on the system heap.

The **only decidable criterion** that is sufficient for confluence of conditional equations and applies to all our example specifications, but does not require termination, is found in [Wirth, 2009].⁷⁷ It can be more easily tested than the admissibility conditions of the Boyer–Moore theorem provers and avoids divergence even in case of non-termination; the proof that it indeed guarantees confluence is very involved.

4.2 Termination and Reducibility

There are two restrictions that are additionally required for any function definition in the Boyer–Moore theorem provers, namely *termination* of the rewrite relation and *reducibility* of all ground terms w.r.t. the rewrite relation.

The requirement of termination should be intuitively clear; we will further discuss it in § 4.4.

To understand the requirement of reducibility, note that it is not only so that we can check the soundness of $(+1')$ and $(+2')$ independently from each other, we can even omit one of the equations, resulting in a partial definition of the function $+$. Indeed, for the function p we did not specify any value for $p(0)$; so $p(0)$ is not reducible in the rewrite relation that results from reading the specifying equations as reduction rules.

A function defined in a Boyer–Moore theorem prover, however, must always be specified completely, in the sense that every application of such a function to (constructor) ground terms must be reducible. This reducibility immediately results from the LISP definition style, which requires all arguments of the function symbol on the left-hand side of its defining equation to be distinct variables.⁷⁸

4.3 Constructor Variables

The two further restrictions of the Boyer–Moore theorem provers, namely reducibility and termination of the rewrite relation that results from reading the specifying equations as reduction rules, **are not essential**; neither for the semantics of recursive function definitions with data types given by constructors,⁷⁹ nor for confluence and consistency.⁸⁰

Note that these two additional restrictions imply that only *total recursive* functions⁸¹ are admissible in the Boyer–Moore theorem provers.

As a termination restriction is not in the spirit of the LISP logic of the Boyer–Moore theorem provers, we have to ask why Boyer and Moore brought up this additional restriction.

⁷⁷The effective confluence test of [Wirth, 2009] requires *binding-triviality* or *-complementary* of every critical peak, and *effective weak-quasi-normality*, i.e. that each equation in the condition must be restricted to constructor variables (cf. § 4.3), or that one of its top terms either is a constructor term or occurs as the argument of a definedness literal in the same condition.

⁷⁸Cf. item (b) of the “definition principle” of [Boyer and Moore, 1979, p. 44f.].

⁷⁹Cf. [Wirth and Gramlich, 1994b].

⁸⁰Cf. [Wirth, 2009].

⁸¹You may follow the explicit reference to [Schoenfeld, 1967] as the basis for the logic of the PURE LISP THEOREM PROVER on Page 93 of [Moore, 1973].

When both reducibility and termination are given, then — similar to the classical case of explicitly defined notions — we can get rid of all recursively defined function symbols by rewriting, but in general only for terms without variables.

A better potential answer is found on Page 87ff. of [Moore, 1973], where confluence of the rewrite relation is discussed and a reference to Russell’s Paradox serves as an argument that confluence alone would not be sufficient for consistency. The argumentation is essentially the following: First, a Boolean function `russell` is recursively defined by

(`russell1`) `russell(b) = false` \Leftarrow `russell(b) = true`
 (`russell2`) `russell(b) = true` \Leftarrow `russell(b) = false`

Then it is claimed that this function definition would result in an inconsistent specification on the basis of the axioms (`bool1–2`) of § 3.5.

This inconsistency, however, arises only if the variable b of the axiom (`bool1`) can be instantiated with the term `russell(b)`, which is actually not our intention and which we do not have to permit: If all variables we have introduced so far are *constructor variables*⁸² in the sense that they can only be instantiated with terms formed from constructor function symbols (incl. constructor constants) and constructor variables, then irreducible terms such as `russell(b)` can denote *junk objects* different from `true` and `false`, and no inconsistency arises.⁸³

Note that these constructor variables are implicitly part of the LISP semantics with its innermost evaluation strategy. For instance, in Example 6 of § 4.1, neither the LISP definition of `PLUS` nor its representation via the positive/negative-conditional equations (`+1'`), (`+2'`) is intended to be applied to a non-constructor term in the sense that X or x should be instantiated to a term that is a function call of a (partially) defined function symbol that may denote a junk object.

Moreover, there is evidence that Moore considered the variables already in 1973 as constructor variables: On Page 87 in [Moore, 1973], we find formulas on definedness and confluence, which make sense only for constructor variables; the one on definedness of the Boolean function `AND` reads⁸⁴ $\exists Z \text{ (IF } X \text{ (IF } Y \text{ T NIL) NIL) = } Z$, which is trivial for a general variable Z and makes sense only if Z is taken to be a constructor variable.

Finally, the way termination is established via induction templates in Boyer–Moore theorem provers and as we will describe it in § 4.4, is sound for the rewrite relation of the defining equations only if we consider the variables of these equations to be constructor variables (or if we restrict the termination result to an innermost rewriting strategy and require that all function definitions are total).

⁸²Such *constructor variables* were formally introduced in [Wirth *et al.*, 1993] and became an essential part of the frameworks found in [Wirth and Gramlich, 1994a; 1994b], [Kühler and Wirth, 1996; 1997], [Wirth, 1997; 2009] [Kühler, 2000], [Avenhaus *et al.*, 2003], and in [Schmidt-Samoa, 2006a; 2006b; 2006c].

⁸³For the appropriate semantics see [Wirth and Gramlich, 1994b], [Kühler and Wirth, 1997].

⁸⁴In the logic of the PURE LISP THEOREM PROVER, the special form `IF` is actually called “`COND`”. This is most confusing because `COND` is a standard special form in LISP, different from `IF`. Therefore, we will ignore this peculiarity and write “`IF`” here for every “`COND`” of the PURE LISP THEOREM PROVER.

4.4 Termination and General Induction Templates

In addition to a LISP-like definition style, the theorem provers for explicit induction **require** termination of the rewrite relation that results from reading the specifying equations as reduction rules. More precisely, in all Boyer–Moore theorem provers except the PURE LISP THEOREM PROVER,⁸⁵ *before* a new function symbol f_k is admitted to the specification, a “valid induction template” — which immediately implies termination — has to be constructed from the defining equation of f_k .⁸⁶

Induction templates were first used in THM and received their name when they were first described in [Boyer and Moore, 1979].

Every time a new recursive function f_k is defined, a system for explicit reduction immediately tries to construct *valid induction templates*; if it does not find any, then the new function symbol is rejected w.r.t. the given definition; otherwise the system links the function name with its definition and its valid induction templates.

The induction templates serve actually two purposes: as witnesses for termination and as the basic tools of the induction rule of explicit induction for generating the step cases.

For a finite number of mutually recursive functions f_k with arity n_k ($k \in K$), an induction template in the most general form consists of the following:

1. A *relational description*⁸⁷ of the changes in the argument pattern of these recursive functions as found in their recursive defining equations:

For each $k \in K$ and for each positive/negative-conditional equation with a left-hand side of the form $f_k(t_1, \dots, t_{n_k})$, we take the set R of recursive function calls of the $f_{k'}$ ($k' \in K$) occurring in the right-hand side or the condition, and some case condition C , which must be a subset of the conjunctive condition literals of the defining equation. Typically, C is empty (i.e. always true) in **case** of constructor-style definitions, and just sufficient to guarantee proper destructor applications in **case** of destructor-style definitions.

Together they form the triple $(f_k(t_1, \dots, t_{n_k}), R, C)$, and a set containing such a triple for each such defining equation forms the relational description.

For our definition of $+$ via $(+1)$, $(+2)$ in §3.4, there is only one recursive equation and only one relevant relational description, namely the following one with an empty case condition:

$$\{ (\text{s}(x) + y, \{x + y\}, \emptyset) \}.$$

Also for our definition of $+$ with $(+1')$, $(+2')$ in Example 6, there is only one recursive equation and only one relevant relational description, namely

$$\{ (x + y, \{p(x) + y\}, \{x \neq 0\}) \}.$$

⁸⁵Note that termination is not proved in the PURE LISP THEOREM PROVER; instead, the soundness of the induction proofs comes with the *proviso* that the rewrite relation of all defined function symbols terminate.

⁸⁶See also item (d) of the “definition principle” of [Boyer and Moore, 1979, p. 44f.] for a formulation that avoids the technical term “induction template”.

⁸⁷The name “relational description” comes from [Walther, 1992; 1993].

2. For each $k \in K$, a variable-free weight term w_{f_k} in which the position numbers $(1), \dots, (n_k)$ are used in place of variables. The position numbers actually occurring in the term are called the *measured positions*.

For our two relational descriptions, only the weight term (1) (consisting just of a position number) makes sense as w_+ , resulting in the set of measured positions $\{1\}$. Indeed, $+$ terminates in both definitions because the argument in the first position gets smaller.

3. A binary predicate $<$ that is known to represent a well-founded relation.
For our two relational descriptions, the predicate $\lambda x, y. (\text{lessp}(x, y) = \text{true})$, is appropriate.

Now, an induction template is *valid* if for each element of the relational description as given above, and for each $f_{k'}(t'_1, \dots, t'_{n_{k'}}) \in R$, the following conjecture is valid:

$$w_{f_{k'}}\{(1) \mapsto t'_1, \dots, (n_{k'}) \mapsto t'_{n_{k'}}\} < w_{f_k}\{(1) \mapsto t_1, \dots, (n_k) \mapsto t_{n_k}\} \Leftarrow \bigwedge C.$$

For our two relational descriptions, this amounts to showing $\text{lessp}(x, s(x)) = \text{true}$ and $\text{lessp}(p(x), x) = \text{true} \Leftarrow x \neq 0$, respectively; so their templates are both valid by lemma (`lessp4`) and axioms (`nat1-2`) and (`p1`).

EXAMPLE 7 (Two Induction Templates with different Measured Positions).

For the ordering predicate `lessp` as defined by (`lessp1-3`) of §3.4, we get two appropriate induction templates with the sets of measured positions $\{1\}$ and $\{2\}$, respectively, both with the relational description

$$\{ (\text{lessp}(s(x), s(y)), \{\text{lessp}(x, y)\}, \emptyset) \},$$

and both with the well-founded ordering $\lambda x, y. (\text{lessp}(x, y) = \text{true})$. The first template has the weight term (1) and the second one has the weight term (2). The validity of both templates is given by lemma (`lessp4`) of §3.4. \square

EXAMPLE 8 (One Induction Template with Two Measured Positions).

For the Ackermann function `ack` as defined by (`ack1-3`) of §3.4, we get only one appropriate induction template. The set of its measured positions is $\{1, 2\}$, because of the weight function `cons((1), cons((2), nil))`, which we will abbreviate in the following with $[(1), (2)]$. The well-founded relation is the lexicographic ordering $\lambda l, k. (\text{lexlimless}(l, k, s(s(s(0)))) = \text{true})$. The relational description has two elements: For the equation (`ack2`) we get

$$(\text{ack}(s(x), 0), \{\text{ack}(x, s(0))\}, \emptyset),$$

and for the equation (`ack3`) we get

$$(\text{ack}(s(x), s(y)), \{\text{ack}(s(x), y), \text{ack}(x, \text{ack}(s(x), y))\}, \emptyset).$$

The validity of the template is expressed in the three equations

$$\begin{aligned} \text{lexlimless}([x, s(0)], [s(x), 0], s(s(s(0)))) &= \text{true}; \\ \text{lexlimless}([s(x), y], [s(x), s(y)], s(s(s(0)))) &= \text{true}; \\ \text{lexlimless}([x, \text{ack}(s(x), y)], [s(x), s(y)], s(s(s(0)))) &= \text{true}; \end{aligned}$$

which follow deductively from (`lessp4`), (`lexlimless1`), (`lexless2-4`), (`length1-2`). \square

For valid induction templates of destructor-style definitions see Examples 18 and 19 in §5.3.7.

4.5 Termination of the Rewrite Relation on Ground Terms

The proof that the existence of a valid induction template for a new set of recursive functions f_k ($k \in K$) actually implies termination of the rewrite relation on ground terms given after addition of the new equations for the f_k can be executed in any model of the old specification as follows:

For an *argumentum ad absurdum*, suppose that there is an infinite sequence of rewrite steps on ground terms. Let us now consider each term to be replaced with the multiset of the weight terms of its function calls for f_k with $k \in K$. Then the rewrite steps with the *old* equations of previous function definitions (of symbols not among the f_k) can only change the multiset by deleting some elements for the following two reasons:

1. The new function symbols do not occur in the old equations.
2. We consider all our variables to be constructor variables as explained in § 4.3.

Moreover, a rewrite step with a new equation reduces the multiset in the well-founded relation given by the multiset extension of the well-founded relation of the template in the assumed model of the old specification because of the fulfilledness of the conditions of the equation and the validity of the template. Thus, in each rewrite step, the the multiset gets smaller in a well-founded ordering or does not change. Moreover, if we assume that rewriting with the old equations terminates, then the new equations must be applied infinitely often in this sequence, and so the multiset gets smaller in infinitely many steps, which is impossible in a well-founded ordering.

4.6 Applicable Induction Templates for Explicit Induction

We restrict the discussion in this section to recursive functions that are not mutually recursive, partly for simplicity and partly because explicit induction is hardly helpful for finding proofs involving mutually recursive functions. Moreover, in principle, users can always encode mutually recursive functions $f_k(\dots)$ by means of a single recursive function $f(k, \dots)$, and tend to provide relevant additional heuristic information via such an encoding, namely by the way they standardize the argument list w.r.t. length and position (cf. the “changeable positions” below).

Thus, all the f_k with arity n_k of § 4.4 simplify to one symbol f with arity n . Moreover, under this restriction it is easy to partition the measured positions of a template into “changeable” and “unchangeable” ones.⁸⁸

Changeable are those measured positions i of the template which sometimes change in the recursion, i.e. for which there is a triple $(f(t_1, \dots, t_n), R, C)$ in the relational description of the template, and an $f(t'_1, \dots, t'_n) \in R$ such that $t'_i \neq t_i$. The remaining measured positions of the template are called *unchangeable*. Unchangeable positions typically result from the inclusion of a global variable into the argument list of a function (to observe an applicative programming style).

⁸⁸This partition into changeable and unchangeable positions (actually: variables) originates in [Boyer and Moore, 1979, p. 185f.].

To improve the applicability of the induction hypotheses of the step cases produced by the induction rule, these induction hypotheses should mirror the recursive calls of the unfolding of the definition of a function f occurring in the induction rule's input formula, say

$$A[f(t''_1, \dots, t''_n)].$$

An induction template is *applicable* to the indicated occurrence of its function symbol f if the terms t''_i at the changeable positions i of the template are *distinct variables* and none of these variables occurs in the terms $t''_{i'}$ that fill the unchangeable positions i' of the template.⁸⁹ For templates of constructor-style equations we additionally have to require here that the first element $f(t_1, \dots, t_n)$ of each triple of the relational description of the template matches $(f(t''_1, \dots, t''_n))\xi$ for some *constructor substitution* ξ that may replace the variables of $f(t''_1, \dots, t''_n)$ with constructor terms, i.e. terms consisting of constructor symbols and variables, such that $t''_i \xi = t''_i$ for each unchangeable position i of the template.

EXAMPLE 9 (Applicable Induction Templates).

Let us consider the conjecture (ack4) from §3.4. From the three induction templates of Examples 7 and 8, only the one of Example 8 is applicable. The two of Example 7 are not applicable because $\text{lessp}(s(x), s(y))$ cannot be matched to $(\text{lessp}(y, \text{ack}(x, y)))\xi$ for any constructor substitution ξ . \square

4.7 Induction Schemes

Let us recall that for every recursive call $f(t'_{j',1}, \dots, t'_{j',n})$ in a positive/negative-conditional equation with left-hand side $f(t_1, \dots, t_n)$, the relational description of an induction template for f contains a triple

$$(f(t_1, \dots, t_n), \{ f(t'_{j,1}, \dots, t'_{j,n}) \mid j \in J \}, C),$$

such that $j' \in J$ (by definition of an induction template).

Let us assume that the induction template is valid and applicable to the occurrence indicated in the input formula $A[f(t''_1, \dots, t''_n)]$ to the induction rule of explicit induction. Let σ be the substitution whose domain are the variables of $f(t_1, \dots, t_n)$ and which matches the first element $f(t_1, \dots, t_n)$ of the triple to $(f(t''_1, \dots, t''_n))\xi$ for some constructor substitution ξ whose domain are the variables of $f(t''_1, \dots, t''_n)$, such that $t''_i \xi = t''_i$ for each unchangeable position i of the template. Then we have $t_i \sigma = t''_i \xi$ for $i \in \{1, \dots, n\}$.

Now, for the well-foundedness of the generic step-case formula

$$\left((A[f(t''_1, \dots, t''_n)])\xi \Leftarrow \bigwedge_{j \in J} (A[f(t''_1, \dots, t''_n)])\mu_j \right) \Leftarrow \bigwedge C \sigma$$

to be implied by the validity of the induction template, it suffices to take substitutions μ_j whose domain $\text{dom}(\mu_j)$ is the set of variables of $f(t''_1, \dots, t''_n)$, such that the constraint $t''_i \mu_j = t'_{j,i}$ is satisfied for each measured position i of the template and for each $j \in J$ (because of $t''_i \xi = t_i \sigma$).

⁸⁹This definition of applicability originates in [Boyer and Moore, 1979, p.185f].

If i is an unchangeable position of the template, then we have $t_i = t'_{j,i}$ and $t''_i \xi = t''_i$. Therefore, we can satisfy the constraint by requiring μ_j to be the identity on the variables of t''_i , simply because then we have $t''_i \mu_j = t''_i = t''_i \xi = t_i \sigma = t'_{j,i} \sigma$.

If i is a changeable position, then we know by the applicability of the template that t''_i is a variable not occurring in another changeable or unchangeable position in $f(t''_1, \dots, t''_n)$, and we can satisfy the constraint simply by defining $t''_i \mu_j := t'_{j,i} \sigma$.

On the remaining variables of $f(t''_1, \dots, t''_n)$, we define μ_j in a way that we get $t''_i \mu_j = t'_{j,i} \sigma$ for as many unmeasured positions i as possible, and otherwise as the identity. This is not required for well-foundedness, but it improves the likeliness of applicability of the induction hypothesis $(A[f(t''_1, \dots, t''_n)])\mu_j$ after unfolding $f(t''_1, \dots, t''_n)\xi$ in $(A[f(t''_1, \dots, t''_n)])\xi$. Note that such an eager instantiation is required in explicit induction unless the logic admits one of the following: existential quantification, existential variables,⁹⁰ lazy induction-hypothesis generation.

An *induction scheme* for the given input formula consists of the following items:

1. The *position set* contains the position of $f(t''_1, \dots, t''_n)$ in $A[f(t''_1, \dots, t''_n)]$. Merging of induction schemes may lead to non-singleton position sets later.
2. The set of the *induction variables*, which are defined as the variables at the changeable positions of the induction template in $f(t''_1, \dots, t''_n)$.
3. To obtain a *step-case description* for all step cases by means of the generic step-case formula displayed above, each triple in the relational description of the considered form is replaced with the new triple

$$(\xi, \{\mu_j \mid j \in J\}, C\sigma).$$

To make as many induction hypotheses available as possible in each case, we assume that step-case descriptions are implicitly kept normalized by the following associative commutative operation: If two triples are identical in their first elements and in their last elements, we replace them with the single triple that has the same first and last elements and the union of the middle elements as new middle element.

4. We also add the *hitting ratio* of all substitutions μ_j with $j \in J$:

$$\frac{|\{(j, i) \in J \times \{1, \dots, n\} \mid t''_i \mu_j = t'_{j,i} \sigma\}|}{|J \times \{1, \dots, n\}|},$$

where J actually has to be the disjoint sum over all the J occurring as index sets of second elements of triples like the one displayed above.

Note that the resulting step-case description is a set describing all step cases of an induction scheme; these step cases are guaranteed to be well-founded,⁹¹ but — for providing a sound induction formula — they still have to be complemented by base cases, which may be analogously described by triples (ξ, \emptyset, C) , such that all substitutions in the first elements of the triples together describe a distinction of cases that is complete for constructor terms and, for each of these substitutions, its case conditions describe a complete distinction of cases again.

EXAMPLE 10 (Induction Scheme).

The template for `ack` of Example 8 is the only one that is applicable to `(ack4)` according to Example 9. It yields the following induction scheme.

The *position set* is $\{1.1.2\}$. It describes the occurrence of `ack` in the second subterm of the left-hand side of the first literal of the formula `(ack4)` as input to the induction rule of explicit induction: $(\text{ack4})/1.1.2 = \text{ack}(x, y)$.

The set of *induction variables* is $\{x, y\}$, because both positions of the induction template are changeable.

The relational description of the induction template is replaced with the *step-case description*

$$\{ (\xi_1, \{\mu_{1,1}\}, \emptyset), (\xi_2, \{\mu_{2,1}, \mu_{2,2}\}, \emptyset) \}.$$

that is given as follows.

The first triple of the relational description, namely

$$(\text{ack}(s(x), 0), \{\text{ack}(x, s(0))\}, \emptyset)$$

(obtained from the equation `(ack2)`) is replaced with $(\xi_1, \{\mu_{1,1}\}, \emptyset)$, where $\xi_1 = \{x \mapsto s(x'), y \mapsto 0\}$ and $\mu_{1,1} = \{x \mapsto x', y \mapsto s(0)\}$. This can be seen as follows. The substitution called σ in the above discussion — which has to match the first element of the triple to $((\text{ack4})/1.1.2)\xi_1$ — has to satisfy $(\text{ack}(s(x), 0))\sigma = (\text{ack}(x, y))\xi_1$. Taking ξ_1 as the minimal constructor substitution given above, this determines $\sigma = \{x \mapsto x'\}$. Moreover, as both positions of the template are changeable, $\mu_{1,1}$ has to match `(ack4)/1.1.2` to the σ -instance of the single element of the second element of the triple, which determines $\mu_{1,1}$ as given.

The second triple of the relational description, namely

$$(\text{ack}(s(x), s(y)), \{\text{ack}(s(x), y), \text{ack}(x, \text{ack}(s(x), y))\}, \emptyset)$$

(obtained from the equation `(ack3)`) is replaced with $(\xi_2, \{\mu_{2,1}, \mu_{2,2}\}, \emptyset)$, where $\xi_2 = \{x \mapsto s(x'), y \mapsto s(y')\}$, $\mu_{2,1} = \{x \mapsto s(x'), y \mapsto y'\}$, and $\mu_{2,2} = \{x \mapsto x', y \mapsto \text{ack}(s(x'), y')\}$. This can be seen as follows. The substitution called σ in the above discussion has to satisfy $(\text{ack}(s(x), s(y)))\sigma = (\text{ack}(x, y))\xi_2$. Taking ξ_2 as the minimal constructor substitution given above, this determines $\sigma = \{x \mapsto x', y \mapsto y'\}$. Moreover, we get the constraints $(\text{ack}(x, y))\mu_{2,1} = (\text{ack}(s(x), y))\sigma$ and $(\text{ack}(x, y))\mu_{2,2} = (\text{ack}(x, \text{ack}(s(x), y)))\sigma$, which determine $\mu_{2,1}$ and $\mu_{2,2}$ as given above.

The hitting ratio for the three constraints on the two arguments of `(ack4)/1.1.2` is $\frac{6}{6} = 1$.

To achieve completeness of the substitutions ξ_k for constructor terms we have to add the base case $(\xi_0, \emptyset, \emptyset)$ with $\xi_0 = \{x \mapsto 0, y \mapsto y\}$ to the step-case description.

The three new triples now describe exactly the three formulas displayed at the beginning of Example 5 in § 3.9. \square

⁹⁰Well-foundedness is indeed guaranteed according to the above discussion. As a consequence, the induction scheme does not need the weight term and the well-founded relation of the induction template anymore.

⁹¹Existential variables are called “free variables” in modern tableau systems (see the 2nd rev. edn. [Fitting, 1996], but not its 1st edn. [Fitting, 1990]) and occur with extended functionality under different names in the inference systems of [Wirth, 2004; 2012b; 2013].

5 AUTOMATED EXPLICIT INDUCTION

5.1 *The Application Context of Automated Explicit Induction*

Since the development of programmable computing machinery in the middle of the 20th century, a major problem of hard- and software has been and still is the uncertainty that they actually always do what they should do. **The only viable solution to this problem** seems to be:

Specify the intended functionality in a language of formal logic, and then supply a formal mechanically checked proof that the program actually satisfies the specification!

Such an approach also requires formalizing the platforms on which the system is implemented. This may include the hardware, operating system, programming language, sensory input, etc. One may additionally formalize and prove that the underlying platforms are implemented correctly and **this may ultimately involve proving**, for example, that a network of logical gates and wires implements a given abstract machine. Eventually, however, one must make an engineering judgment that certain physical objects (e.g. printed circuit boards, gold plated pins, power supplies, etc.) reliably behave as specified. To be complete, such an approach would also require a verification that the verification system is sound and correctly implemented.⁹²

A crucial problem, however, is the cost — in time and money — of doing the many proofs required, given the huge amounts of application hard- and software in our modern economies. Thus, we can expect formal verification only in areas where the managers expect that mere testing does not suffice, that the costs of the verification process are lower than the costs of bugs in the hard- or software, and that the competitive situation admits the verification investment. **A good candidate** is the area of central processing units (CPUs) in standard processors.

To reduce the costs of verification, we can hope to automate it with automated theorem-proving systems; this has to include an automation of mathematical induction because **most data types** used in digital design, such as natural numbers, arrays, lists, and trees, require induction for the verification of their properties. Decision methods (many of them exploiting finiteness, e.g. the use of 32-bit data paths) allow automatic verification of some modules, but — barring a completely unexpected breakthrough in the future — the verification of a new hard- or software system will always require human users who help the theorem-proving systems to explore and develop the notions and theories that properly match the new system. Already today, however, ACL2 often achieves complete automation in verifying minor modifications of previously verified modules — an activity called *proof maintenance* which is increasingly important in the microprocessor-design industry.

⁹²See, for example, [Davis, 2009].

5.2 *The* PURE LISP THEOREM PROVER

Our overall task is to answer — from a historical perspective — the question:

How could Robert S. Boyer and J Strother Moore — starting virtually from zero⁹³ in the middle of 1972 — actually invent their long-lived solutions to the hard heuristic problems in the automation of induction and implement them in the sophisticated theorem prover THM as described [Boyer and Moore, 1979]?

As already described in §1, the breakthrough in the heuristics for automated inductive theorem proving was achieved with the “PURE LISP THEOREM PROVER”, developed and implemented by Boyer and Moore. It was presented by Moore at the third IJCAI, which took place in Stanford (CA) in August 1973,⁹⁴ but it is best documented in Part II of Moore’s PhD thesis [1973], defended in November 1973.

The PURE LISP THEOREM PROVER is given no name in the before-mentioned publications. The only occurrence of the name in publication seems to be in [Moore, 1975a, p.1], where it is actually called “the Boyer–Moore PURE LISP THEOREM PROVER”.

To make a long story short, the fundamental insights were

- to exploit the duality of recursion and induction to formulate explicit induction hypotheses,
- to abandon “random” search and focus on simplifying the goal by rewriting and normalization techniques to lead to opportunities to use the induction hypotheses, and
- to support generalization to prepare subgoals for subsequent inductions.

Thus, it is not enough for us to focus here just on the induction heuristics *per se*, but it is necessary to place them in the context of the development of the Boyer–Moore waterfall (cf. Figure 1).

⁹³No heuristics at all were explicitly described, for instance, in Burstall’s 1968 work on program verification by induction over recursive functions in [Burstall, 1969], where the proofs were not even formal, and an implementation seemed to be more or less utopian:

“The proofs presented will be mathematically rigorous but not formalised to the point where each inference is presented as a mechanical application of elementary rules of symbol manipulation. This is deliberate since I feel that our first aim should be to devise methods of proof which will prove the validity of non-trivial programs in a natural and intelligible manner. Obviously we will wish at some stage to formalise the reasoning to a point where it can be performed by a computer to give a mechanised debugging service.” [Burstall, 1969, p. 41]

As far as we are aware, the only implementation of an automatically invoked mathematical induction heuristic prior to 1972 is in a set-theory prover by Bledsoe [1971], which uses structural induction over 0 and s (cf. §3.4) on a randomly picked, universally quantified variable of type nat.

⁹⁴Cf. [Boyer and Moore, 1973].

To understand the achievements a bit better, let us now discuss the material of Part II of Moore’s PhD thesis in some detail, because it provides some explanation on how Boyer and Moore could be so surprisingly successful. Especially helpful for understanding the process of creation are those procedures of the PURE LISP THEOREM PROVER that are provisional w.r.t. to their refinement in later Boyer–Moore theorem provers. Indeed, these provisional procedures help to decompose the giant leap from nothing to THM, which was achieved by only two men in less than eight years of work.

As W. W. Bledsoe (1921–1995) was Boyer’s PhD advisor, it is no surprise that the PURE LISP THEOREM PROVER shares many design features with Bledsoe’s provers.⁹⁵

Boyer and Moore report that in late 1972 and early 1973 they were doing proofs about list data structures on the blackboard and verbalizing to each other the heuristics behind their choices on how to proceed with the proof.⁹⁶ This means that, although explicit induction is not the approach humans would choose for non-trivial induction tasks, the heuristics of the PURE LISP THEOREM PROVER are learned from human heuristics after all.

Note that Boyer’s and Moore’s method of learning computer heuristics from their own human behavior in mathematical logic was a step of two young men against the spirit of the time: the use of vast amounts of computational power to *search* an even more enormous space of possibilities. Boyer’s and Moore’s goal, however, was in a sense more modest:

“The program was designed to behave properly on simple functions. The overriding consideration was that it should be automatically able to prove theorems about simple LISP functions in the straightforward way we prove them.” [Moore, 1973, p. 205]

It may be that the orientation toward human-like or “intelligible” methods and heuristics in the automation of theorem proving had also some tradition in Edinburgh at the time,⁹⁷ but here the major influence on Boyer and Moore is again W. W. Bledsoe.⁹⁸

The source code of the PURE LISP THEOREM PROVER was written in the programming language POP–2.⁹⁹ Boyer and Moore were the only programmers involved in the implementation. The average time in the central processing unit (CPU) of the ICL–4130 for the proof of a theorem is reported to be about ten seconds.¹⁰⁰ This was considered fast at the time, compared to the search-dominated proofs by resolution systems. Moore explains the speed:

⁹⁵On Page 172 of [Moore, 1973] we read on the PURE LISP THEOREM PROVER:

“The design of the program, especially the straightforward approach of ‘hitting’ the theorem over and over again with rewrite rules until it can no longer be changed, is largely due to the influence of W. W. Bledsoe.”

⁹⁶Cf. [Wirth, 2012d].

⁹⁷Cf. e.g. the quotation from [Burstall, 1969] in Note 93.

⁹⁸Cf. e.g. [Bledsoe *et al.*, 1972].

⁹⁹Cf. [Burstall *et al.*, 1971].

“Finally, it should be pointed out that the program uses no search. At no time does it ‘undo’ a decision or back up. This is both the primary reason it is a fast theorem prover, and strong evidence that its methods allow the theorem to be proved in the way a programmer might ‘observe’ it. The program is designed to make the right guess the first time, and then pursue one goal with power and perseverance.”

[Moore, 1973, p.208]

One remarkable omission in the PURE LISP THEOREM PROVER is lemma application. As a consequence, the success of proving a set of theorems cannot depend on the order of their presentation to the theorem prover. Indeed, just as the resolution theorem provers of the time, the PURE LISP THEOREM PROVER starts every proof right from scratch and does not improve its behavior with the help of previously proved lemmas. This was a design decision; one of the reasons was:

“Finally, one of the primary aims of this project has been to demonstrate clearly that it is possible to prove program properties entirely automatically. A total ban on all built-in information about user defined functions thus removes any taint of user supplied information.”

[Moore, 1973, p.203]

Moreover, all induction orderings in the PURE LISP THEOREM PROVER are recombinations of constructor relations, such that all inductions it can do are structural inductions over combinations of constructors. As a consequence, contrary to later Boyer–Moore theorem provers, the well-foundedness of the induction orderings does not depend on the termination of the recursive function definitions.¹⁰¹

Nevertheless, the soundness of the PURE LISP THEOREM PROVER depends on the termination of the recursive function definitions, but only in one aspect: It simplifies and evaluates expressions under the assumption of termination. For instance, both $(\text{IF}^{102} a d d)$ and $(\text{CDR} (\text{CONS} a d))$ simplify to d , **no matter whether a terminates;** and it is admitted to rewrite with a recursive function definition even if an argument of the function call does not terminate.

The termination of the recursively defined functions, however, is not checked by the PURE LISP THEOREM PROVER, but comes as a *proviso* for its soundness.

The logic of the PURE LISP THEOREM PROVER is an applicative¹⁰³ subset of the logic of LISP. The only *destructors* in this logic are CAR and CDR. They are overspecified on the only *constructors* NIL and CONS by the following equations:

¹⁰⁰Here is the actual wording of the timing result found on Page 171f. of [Moore, 1973]:

“Despite these inefficiencies, the ‘typical’ theorem proved requires only 8 to 10 seconds of CPU time. For comparison purposes, it should be noted that the time for CONS in 4130 POP-2 is 400 microseconds, and CAR and CDR are about 50 microseconds each. The hardest theorems solved, such as those involving SORT, require 40 to 50 seconds each.”

¹⁰¹Note that the well-foundedness of the constructor relations depends on distinctness of the constructor ground terms in the models, but this does not really depend on the termination of the recursive functions because (as discussed in § 4.1) confluence is sufficient here.

¹⁰²Cf. Note 84.

$$\begin{array}{l|l} (\text{CAR } (\text{CONS } a \ d)) = a & (\text{CAR NIL}) = \text{NIL} \\ (\text{CDR } (\text{CONS } a \ d)) = d & (\text{CDR NIL}) = \text{NIL} \end{array}$$

As standard in LISP, every term of the form $(\text{CONS } a \ d)$ is taken to be **true** in the logic of the PURE LISP THEOREM PROVER if it occurs at an argument position with Boolean intention. The actual truth values (to be returned by Boolean functions) are NIL (representing **false**) and T, which is an abbreviation for (CONS NIL NIL) and represents **true**.¹⁰⁴ Unlike conventional LISPs (both then and now), the natural numbers are represented by lists of NILs to keep the logic simple; the natural number 0 is represented by NIL and the successor function $s(d)$ is represented by $(\text{CONS NIL } d)$.¹⁰⁵

Let us now discuss the behavior of the PURE LISP THEOREM PROVER by describing the instances of the stages of the Boyer–Moore waterfall (cf. Figure 1) as they are described in Moore’s PhD thesis.

5.2.1 Simplification in the PURE LISP THEOREM PROVER

The first stage of the Boyer–Moore waterfall — “simplification” in Figure 1 — is called “normalation”¹⁰⁶ in the PURE LISP THEOREM PROVER. It applies the following simplification procedures to LISP expressions until the result does not change anymore: “evaluation”, “normalization”, and “reduction”.

“*Normalization*” tries find sufficient conditions for a given expression to have the soft type “Boolean” and to normalize logical expressions. Contrary to clausal logic over equational atoms, LISP admits EQUAL and IF to appear not only at the top level, but in nested terms. To free later tests and heuristics from checking for their triggers in every equivalent form, such a normalization w.r.t. propositional logic and equality is part of most theorem provers today.

“*Reduction*” is a form of what today is called *contextual rewriting*. It is based on the fact that — in the logic of the PURE LISP THEOREM PROVER — in the conditional expression

$$(\text{IF } c \ p \ n)$$

we can simplify occurrences of c in p to $(\text{CONS } (\text{CAR } c) \ (\text{CDR } c))$, and in n to NIL. The replacement with $(\text{CONS } (\text{CAR } c) \ (\text{CDR } c))$ is executed only at positions with Boolean intention and can be improved in the following two special cases:

1. If we know that c is of soft type “Boolean”, then we rewrite all occurrences of c in p actually to T.

¹⁰³Cf. Note 76.

¹⁰⁴Cf. 2nd paragraph of Page 86 of [Moore, 1973].

¹⁰⁵Cf. 2nd paragraph of Page 87 of [Moore, 1973].

¹⁰⁶During the oral defense of the dissertation, Moore’s committee abhorred the non-word and instructed him to choose a word. Some copies of the dissertation call the process “simplification.”

2. If c is of the form $(\text{EQUAL } l \ r)$, then we can rewrite occurrences of l in p to r (or vice versa). Note that we have to treat the variables in l and r as constants in this rewriting. The PURE LISP THEOREM PROVER rewrites in this case only if either l or r is a ground term;¹⁰⁷ then the other cannot be a ground term because the equation would otherwise have been simplified to T or NIL in the previously applied “evaluation”. So replacing the latter term with the ground term everywhere in p must terminate, and this is all the contextual rewriting with equalities that the PURE LISP THEOREM PROVER does in “reduction”.¹⁰⁸

“Evaluation” is a procedure that evaluates expressions partly by simplification within the elementary logic as given by Boolean operations and the equality predicate. Moreover, “evaluation” executes some rewrite steps with the equations defining the recursive functions. Thus, “evaluation” can roughly be seen as normalization with the rewrite relation resulting from the elementary logic and from the recursive function definitions. The rewrite relation is applied according to the innermost left-to-right rewriting strategy, which is standard in LISP.

By “evaluation”, ground terms are completely evaluated to their normal forms. Terms containing (implicitly universally quantified) variables, however, have to be handled in addition. Surprisingly, the considered rewrite relation is not necessarily terminating on non-ground terms, although the LISP evaluation of ground terms terminates because of the assumed termination of recursive function definitions (cf. § 4.4). The reason for this non-termination is the following: Because of the LISP definition style via *unconditional* equations, the positive/negative conditions are actually part of the *right-hand sides* of the defining equations, such that the rewrite step can be executed even if the conditions evaluate neither to false nor to true. For instance, in Example 6 of § 4.1, a rewrite step with the definition of PLUS can always be executed, whereas a rewrite step with $(+1')$ or $(+2')$ requires $x=0$ to be definitely true or definitely false. This means that non-termination may result from the rewriting of cases that do not occur in the evaluation of any ground instance.¹⁰⁹

As the final aim of the stages of the Boyer–Moore waterfall is a formula that provides concise and sufficiently strong induction hypotheses in the last of these

¹⁰⁷A **ground term is a term without variables**. Actually, this ground term here is always a *constructor* ground term because the previously applied “evaluation” procedure has reduced any ground term to a constructor ground term, provided that the termination *proviso* is satisfied.

¹⁰⁸Note, however, that further contextual rewriting with equalities is applied in a later stage of the Boyer–Moore waterfall, named *cross-fertilization*.

¹⁰⁹It becomes clear in the second paragraph on Page 118 of [Moore, 1973] that the code of both the positive and the negative case of a conditional will be evaluated, unless one of them can be canceled by the complete evaluation of the governing condition to true or false. Note that the evaluation of both cases is necessary indeed and cannot be avoided in practice.

Moreover, note that a stronger termination requirement that guarantees termination independent of the governing condition is not feasible for recursive function definitions in practice.

Later Boyer–Moore theorem provers also use lemmas for rewriting during symbolic evaluation, which is another source of possible non-termination.

stages, symbolic evaluation must be prevented from unfolding function definitions unless the context admits us to expect an effect of simplification.¹¹⁰

Because the main function of “evaluation” — only to be found in this first one of the Boyer–Moore provers — is to collect data on which base and step cases should be chosen later by the induction rule, the PURE LISP THEOREM PROVER applies a unique procedure to stop the unfolding of recursive function definitions:

A rewrite step with an equation defining a recursive function f is canceled if there is a CAR or a CDR in an argument to an occurrence of f in the right-hand side of the defining equation that is encountered during the control flow of “evaluation”, and if this CAR or CDR is not removed by the “evaluation” of the arguments of this occurrence of f under the current environment updated by matching the left-hand side of the equation to the redex. For instance, “evaluation” of (PLUS (CONS NIL X) Y) returns (CONS NIL (PLUS X Y)); whereas “evaluation” of (PLUS X Y) returns (PLUS X Y) and informs the induction rule that only (CDR X) occurred in the recursive call during the trial to rewrite with the definition of PLUS. In general, such occurrences indicate which induction hypotheses should be generated by the induction rule.^{111 112}

“Evaluation” provides a link between symbolic evaluation and the induction rule of explicit induction. The question “Which case distinction on which variables should be used for the induction proof and how should the step cases look?” is reduced to the quite different question “Where do destructors like CAR and CDR heap up during symbolic evaluation?”. This reduction helps to understand by which intermediate steps it was possible to develop the most surprising, sophisticated recursion analysis of later Boyer–Moore theorem provers.

5.2.2 *Destructor Elimination in the PURE LISP THEOREM PROVER*

There is no such stage in the PURE LISP THEOREM PROVER.¹¹³

5.2.3 *(Cross-) Fertilization in the PURE LISP THEOREM PROVER*

Fertilization is just contextual rewriting with an equality, described before for the “reduction” that is part of the simplification of the PURE LISP THEOREM PROVER, but now with an equation between *two non-ground* terms.

¹¹⁰In QUODLIBET this is achieved by *contextual rewriting*, where evaluation stops when the governing conditions cannot be established from the context. Cf. [Schmidt-Samoa, 2006b; 2006c].

¹¹¹Actually, “evaluation” also informs which occurrences of CAR or CDR besides the arguments of recursive occurrences of PLUS were permanently introduced during that trial to rewrite. Such occurrences trigger an additional case analysis to be generated by the induction rule, mostly as a compensation for the omission of the stage of “destructor elimination” in the PURE LISP THEOREM PROVER.

¹¹²The mechanism for partially enforcing termination of “evaluation” according to this procedure is vaguely described in the last paragraph on Page 118 of Moore’s PhD thesis. As this kind of “evaluation” is only an intermediate solution on the way to more refined control information for the induction rule in later Boyer–Moore theorem provers, the rough information given here may suffice.

¹¹³See, however, Note 111 and the discussion of the PURE LISP THEOREM PROVER in §5.3.2.

The most important case of fertilization is called “*cross-fertilization*”. It occurs very often in step cases of induction proofs of equational theorems, and we have seen it already in Example 4 of § 3.8.1.

Neither Boyer nor Moore ever explicitly explained why cross-fertilization is “cross”, but in [Moore, 1973, p. 142] we read:

“When two equalities are involved and the fertilization was right-side”
 [of the induction hypothesis put] “into left-side” [of the induction conclusion,] “or left-side into right-side, it is called ‘cross-fertilization’.”

“Cross-fertilization” is actually a term from genetics referring to the alignment of haploid genetic code from male and female to a diploid code in the egg cell. This image may help to recall that only that side (i.e. left- or right-hand side of the equation) of the induction conclusion which was activated by a successful simplification is further rewritten during cross-fertilization, namely *everywhere where the same side of the induction hypothesis occurs as a redex* — just like two haploid chromosomes have to start at the same (activated) sides for successful recombination. In [Moore, 1973, p. 139] we find the reason for this: Cross-fertilization frequently produces a new goal that is easy to prove because its uniform “genre” in the sense that its subterms uniformly come from just one side of the original equality.

Furthermore — for getting a sufficiently powerful new induction hypothesis in a follow-up induction — it is crucial to delete the equation used for rewriting (i.e. the old induction hypothesis), which can be remembered by the fact that — in the image — only one (diploid) genetic code remains.

The only noteworthy difference between cross-fertilization in the PURE LISP THEOREM PROVER and later Boyer–Moore theorem provers is that the generalization that consists in the deletion of the used-up equations **is done in a halfhearted way, which admits a later identification of the deleted equation.**

5.2.4 Generalization in the PURE LISP THEOREM PROVER

Generalization in the PURE LISP THEOREM PROVER works as described in § 3.9. The only difference to our presentation there is the following: Instead of just replacing all occurrences of a non-variable subterm t with a new variable z , the definition of the top function symbol of t is used to generate the definition of a new predicate p , such that $p(t)$ holds. Then the generalization of $T[t]$ becomes $T[z] \Leftarrow p(z)$ instead of just $T[z]$. The version of this automated function synthesis actually implemented in the PURE LISP THEOREM PROVER is just able to generate simple type properties, such as being a number or being a Boolean value.¹¹⁴

Note that generalization is essential for the PURE LISP THEOREM PROVER because it does not use lemmas, and so it cannot build up a more and more complex theory successively. It is clear that this limits the complexity of the theorems it can prove, because a proof can only be successful if the implemented

¹¹⁴See § 3.7 of [Moore, 1973]. As explained on Page 156f. of [Moore, 1973], Boyer and Moore failed with the trial to improve the implemented version of the function synthesis, so that it could generate a predicate on a list being ordered from a simple sorting-function.

non-backtracking heuristics work out all the way from the theorem down to the most elementary theory.

5.2.5 Elimination of Irrelevance in the PURE LISP THEOREM PROVER

There is no such stage in the PURE LISP THEOREM PROVER.

5.2.6 Induction in the PURE LISP THEOREM PROVER

This stage of the PURE LISP THEOREM PROVER applies the induction rule of explicit induction as described in §3.8. Induction is tried only after the goal formula has been maximally simplified and generalized by repeated trips through the waterfall. The induction heuristic takes a formula as input and returns a conjunction of base and step cases to which the input formula reduces. Contrary to later Boyer–Moore theorem provers that gather the relevant information via induction schemes gleaned by preprocessing recursive definitions,¹¹⁵ the induction rule of the PURE LISP THEOREM PROVER is based solely on the information provided by “evaluation” as described in §5.2.1.

Instead of trying to describe the general procedure, let us just put the induction rule of the PURE LISP THEOREM PROVER to test with two paradigmatic examples. In these examples we ignore the here irrelevant fact that the PURE LISP THEOREM PROVER actually uses a list representation for the natural numbers. The only effect of this is that the destructor \mathbf{p} takes over the rôle of the destructor \mathbf{CDR} .

EXAMPLE 11 (Induction Rule in the Explicit Induction Proof of (ack4)).

Let us see how the induction rule of the PURE LISP THEOREM PROVER proceeds w.r.t. the proof of (ack4) that we have seen in Example 5 of §3.9. The substitutions ξ_1, ξ_2 computed as instances for the induction conclusion in Example 10 of §4.7 suggest an overall case analysis with a base case given by $\{x \mapsto 0\}$, and two step cases given by $\xi_1 = \{x \mapsto \mathbf{s}(x'), y \mapsto 0\}$ and $\xi_2 = \{x \mapsto \mathbf{s}(x'), y \mapsto \mathbf{s}(y')\}$. The PURE LISP THEOREM PROVER requires the axioms (ack1), (ack2), (ack3) to be in destructor instead of constructor style:

$$\begin{aligned} (\text{ack1}') \quad \text{ack}(x, y) = \mathbf{s}(y) & \Leftarrow x = 0 \\ (\text{ack2}') \quad \text{ack}(x, y) = \text{ack}(\mathbf{p}(x), \mathbf{s}(0)) & \Leftarrow x \neq 0 \wedge y = 0 \\ (\text{ack3}') \quad \text{ack}(x, y) = \text{ack}(\mathbf{p}(x), \text{ack}(x, \mathbf{p}(y))) & \Leftarrow x \neq 0 \wedge y \neq 0 \end{aligned}$$

“Evaluation” does not rewrite the input conjecture with this definition, but writes a “fault description” for the permanent occurrences of \mathbf{p} as arguments of the three occurrences of ack on the right-hand sides, essentially consisting of the following three “pockets”: $(\mathbf{p}(x))$, $(\mathbf{p}(x), \mathbf{p}(y))$, and $(\mathbf{p}(y))$, respectively. Similarly, the pockets gained from the fault descriptions of rewriting the input conjecture with the definition of lessp essentially consists of the pocket $(\mathbf{p}(y), \mathbf{p}(\text{ack}(x, y)))$. Similar to the non-applicability of the induction template for lessp in Example 9 of §4.6, this

¹¹⁵Cf. §4.7.

fault description does not suggest any induction because one of the arguments of \mathbf{p} in one of the pockets is not a variable. As this is not the case for the previous fault description, it suggests the set of all arguments of \mathbf{p} in all pockets as induction variables. As this is the only suggestion, no merging of suggested inductions is required here.

So the PURE LISP THEOREM PROVER picks the right set of induction variables. Nevertheless, it fails to generate appropriate base and step cases, because the overall case analysis results in two base cases given by $\{x \mapsto 0\}$ and $\{y \mapsto 0\}$, and a step case given by $\{x \mapsto \mathbf{s}(x'), y \mapsto \mathbf{s}(y')\}$.¹¹⁶ This turns the first step case of the proof of Example 5 into a base case. The PURE LISP THEOREM PROVER finally fails with the step case it actually generates:

$$\text{lessp}(\mathbf{s}(y'), \text{ack}(\mathbf{s}(x'), \mathbf{s}(y'))) = \text{true} \Leftarrow \text{lessp}(y', \text{ack}(x', y')) = \text{true}.$$

This step case has only one hypothesis, which is neither of the two we need. \square

EXAMPLE 12 (Proof of (lessp7) by Explicit Induction with Merging).

Let us write $T(x, y, z)$ for (lessp7) of §3.4. From the proof of (lessp7) in Example 3 of §3.7 we can learn the following: The proof becomes simpler when we take $T(0, \mathbf{s}(y'), \mathbf{s}(z'))$ as base case (besides say $T(x, y, 0)$ and $T(x, 0, \mathbf{s}(z'))$), instead of any of $T(0, y, \mathbf{s}(z'))$, $T(0, \mathbf{s}(y'), z)$, $T(0, y, z)$. The crucial lesson from Example 3, however, is that the step case of explicit induction has to be

$$T(\mathbf{s}(x'), \mathbf{s}(y'), \mathbf{s}(z')) \Leftarrow T(x', y', z').$$

Note that the Boyer–Moore heuristics for using the induction rule of explicit induction look only one rewrite step ahead, separately for each occurrence of a recursive function in the conjecture.

This means that there is no way for their heuristic to apply case distinctions on variables step by step, most interesting first, until finally we end up with an instance of the induction hypothesis as in Example 3.

Nevertheless, even the PURE LISP THEOREM PROVER manages the pretty hard task of suggesting exactly the right step case. It requires the axioms (lessp1), (lessp2), (lessp3) to be in destructor style:

$$\begin{aligned} (\text{lessp1}') \quad \text{lessp}(x, y) = \text{false} & \Leftarrow y = 0 \\ (\text{lessp2}') \quad \text{lessp}(x, y) = \text{true} & \Leftarrow y \neq 0 \wedge x = 0 \\ (\text{lessp3}') \quad \text{lessp}(x, y) = \text{lessp}(\mathbf{p}(x), \mathbf{p}(y)) & \Leftarrow y \neq 0 \wedge x \neq 0 \end{aligned}$$

“Evaluation” does not rewrite any of the occurrences of lessp in the input conjecture with this definition, but writes one “fault description” for each of these occurrences about the permanent occurrences of \mathbf{p} as argument of the one occurrence of lessp on the right-hand sides, resulting in one “pocket” in each fault description, which essentially consist of $((\mathbf{p}(z)))$, $((\mathbf{p}(x), \mathbf{p}(y)))$, and $((\mathbf{p}(y), \mathbf{p}(z)))$, respectively. The PURE LISP THEOREM PROVER merges these three fault descriptions to the single one $((\mathbf{p}(x), \mathbf{p}(y), \mathbf{p}(z)))$, and so suggests the proper step case indeed, although it suggests the base case $T(0, y, z)$ instead of $T(0, \mathbf{s}(y'), \mathbf{s}(z'))$, which requires some extra work, but does not result in a failure. \square

¹¹⁶We can see this from a similar case on Page 164 and from the explicit description on the bottom of Page 166 in [Moore, 1973].

5.2.7 Conclusion on the PURE LISP THEOREM PROVER

The PURE LISP THEOREM PROVER establishes the historic breakthrough regarding the heuristic automation of inductive theorem proving in theories specified by recursive function definitions.

Moreover, it is the first implementation of a prover for explicit induction going beyond most simple structural inductions over s and 0 .

Furthermore, the PURE LISP THEOREM PROVER has most of the stages of the Boyer–Moore waterfall (cf. Figure 1), and these stages occur in the final order and with the final overall behavior of throwing the formulas back to the center pool after a stage was successful in changing them.

As we have seen in Example 11 of § 5.2.6, the main weakness of the PURE LISP THEOREM PROVER is the realization of its induction rule, which ignores most of the structure of the recursive calls in the right-hand sides of recursive function definitions.¹¹⁷ In the PURE LISP THEOREM PROVER, all information on this structure taken into account by the induction rule comes from the fault descriptions of previous applications of “evaluation”, which **drop** a lot of information that is actually required for finding the proper instances for the eager instantiation of induction hypotheses required in explicit induction.

As a consequence, all induction hypotheses and conclusions of the PURE LISP THEOREM PROVER are instantiations of the input formula with mere constructor terms. Nevertheless, the PURE LISP THEOREM PROVER can generate multiple hypotheses for astonishingly complicated step cases, which go far beyond the simple ones typical for structural induction over s and 0 .

Although the induction stage of the PURE LISP THEOREM PROVER is pretty underdeveloped compared to the sophisticated *recursion analysis* of the later Boyer–Moore theorem provers, it somehow contains all essential later ideas in a rudimentary form, such as recursion analysis and the merging of step cases. As we have seen in Example 12, the simple merging procedure of the PURE LISP THEOREM PROVER is surprisingly successful.

The PURE LISP THEOREM PROVER cannot succeed, however, in the rare cases where a step case has to follow a destructor different from `CAR` and `CDR` (such as `delfirst` in § 3.5), or in the more general case that the arguments of the recursive calls contain recursively defined functions at the measured positions (such as the Ackermann function in Example 11).

The weaknesses and provisional procedures of the PURE LISP THEOREM PROVER we have documented, help to decompose the giant leap ~~from~~ nothing to THM, and so fulfill our historiographic intention expressed at the beginning of § 5.2.

Especially the link between symbolic evaluation and the induction rule of explicit induction described at the end of § 5.2.1 may be crucial for the success of the entire development of recursion analysis and explicit induction.

¹¹⁷There are indications that the induction rule of the PURE LISP THEOREM PROVER had to be implemented in a hurry. For instance, on top of Page 168 of [Moore, 1973], we read on the PURE LISP THEOREM PROVER: “The case for n term induction is much more complicated, and is not handled in its full generality by the program.”

5.3 THM

“THM” is the name used in this article for a release of the prover described in [Boyer and Moore, 1979]. Note that the clearness, precision, and detail of the natural-language descriptions of heuristics in [Boyer and Moore, 1979] is unique and unrivaled.¹¹⁸ To the best of our knowledge, there is no similarly broad treatment of heuristics in theorem proving.

Except for ACL2, Boyer and Moore never gave names to their theorem provers.¹¹⁹ The names “THM” (for “theorem prover”), “QTHM” (“quantified THM”), and “NQTHM” (“new quantified THM”) were actually the directory names under which the different versions of their theorem provers were developed and maintained.¹²⁰ QTHM was never released and its development was discontinued soon after the “quantification” in NQTHM had turned out to be superior; so the name “QTHM” was never used in public. Until today, it seems that “THM” appeared in publication only as a mode in NQTHM,¹²¹ which simulates the release previous to the release of NQTHM (i.e. before “quantification” was introduced) with a logic that is a further development of the one described in [Boyer and Moore, 1979]. It was Matt Kaufmann (*1952) who started calling the prover “NQTHM”, in the second half of the 1980s.¹²² The name “NQTHM” appeared in publication first in [Boyer and Moore, 1988b] as a mode in NQTHM.

In this section we describe the enormous heuristic improvements documented in [Boyer and Moore, 1979] as compared to [Moore, 1973] (cf. § 5.2). In case of the minor differences of the logic described in [Boyer and Moore, 1979] and of the later released version that is simulated by the THM mode in NQTHM as documented in [Boyer and Moore, 1988b; 1998], we try to follow the later descriptions, partly because of their elegance, partly because NQTHM is still an available program. For this reason, we have entitled this section “THM” instead of “The standard reference on the Boyer–Moore heuristics [Boyer and Moore, 1979]”.

From 1973 to 1981 Boyer and Moore were researchers at Xerox Palo Alto Research Center (Moore only) and — just a few miles away — at SRI International in Menlo Park (CA). From 1981 they were both professors at The University of

¹¹⁸In [Boyer and Moore, 1988b, p. xi] and [Boyer and Moore, 1998, p. xv] we can read about the book [Boyer and Moore, 1979]:

“The main purpose of the book was to describe in detail how the theorem prover worked, its organization, proof techniques, heuristics, etc. One measure of the success of the book is that we know of three independent successful efforts to construct the theorem prover from the book.”

¹¹⁹The only further exception seems to be [Moore, 1975a, p. 1], where the PURE LISP THEOREM PROVER is called “the Boyer–Moore Pure LISP Theorem Prover”, probably because Moore wanted to stress that, though Boyer appears in the references of [Moore, 1975a] only in [Boyer and Moore, 1975], Boyer has had an equal share in contributing to the PURE LISP THEOREM PROVER right from the start.

¹²⁰Cf. [Boyer, 2012].

¹²¹For the occurrences of “THM” in publications, and for the exact differences between the THM and NQTHM modes and logics, see Pages 256–257 and 308 in [Boyer and Moore, 1988b], as well as Pages 303–305, 326, 357, and 386 in the second edition [Boyer and Moore, 1998].

¹²²Cf. [Boyer, 2012].

Texas at Austin or scientists at Computational Logic Inc. in Austin (TX). So they could easily meet and work together. And — just like the PURE LISP THEOREM PROVER — the provers THM and NQTHM were again developed and implemented exclusively by Boyer and Moore.¹²³

In the six years separating THM from the PURE LISP THEOREM PROVER, Boyer and Moore extended the system in four important ways that especially affect inductive theorem proving. The first major extension is the provision for an arbitrary number of inductive data types, where the PURE LISP THEOREM PROVER supported only CONS. The second is the formal provision of a definition principle with its explicit termination analysis based on well-founded relations which we discussed in §4. The third major extension is the expansion of the proof techniques used by the waterfall, notably including the use of previously proved theorems, most often as rewrite rules via what would come to be called “contextual rewriting”, and by which the THM user can “guide” the prover by posing lemmas that the system cannot discover on its own. The fourth major extension is the synthesis of induction schemes from definition-time termination analysis and the application and manipulation of those schemes at proof-time to create “appropriate” inductions for a given formula, in place of the PURE LISP THEOREM PROVER’s less structured reliance on symbolic evaluation. We discuss THM’s inductive data types, waterfall, and induction schemes below.

By means of the new *shell principle*,¹²⁴ it is now possible to define new data types by describing the *shell*, a constructor with at least one argument, each of whose arguments may have a simple type restriction, and the optional *base object*, a nullary constructor.¹²⁵ Each argument of the shell can be accessed¹²⁶ by its destructor, for which a name and a default value (for the sake of totality) have to be given in addition. The user also has to supply a name for the predicate that that recognizes¹²⁶ the objects of the new data type (as the logic remains untyped).

NIL lost its elementary status and is now an element of the shell PACK of symbols.¹²⁷ T and F now abbreviate the nullary function calls (TRUE) and (FALSE),

¹²³In both [Boyer and Moore, 1988b, p. xv] and [Boyer and Moore, 1998, p. xix] we read:

“Notwithstanding the contributions of all our friends and supporters, we would like to make clear that ours is a very large and complicated system that was written entirely by the two of us. Not a single line of LISP in our system was written by a third party. Consequently, every bug in it is ours alone. Soundness is the most important property of a theorem prover, and we urge any user who finds such a bug to report it to us at once.”

¹²⁴Cf. [Boyer and Moore, 1979, p. 37ff.].

¹²⁵Note that this restriction to at most two constructors, including exactly one with arguments, is pretty uncomfortable. For instance, it neither admits simple enumeration types (such as the Boolean values), nor disjoint unions (e.g., as part of the popular record types with variants, say of [Wirth, 1971]). Moreover, mutually recursive data types are not possible, such as and-or-trees, where each element is a list of or-and-trees, and vice versa, as given by the following four constructors:

empty-or-tree :	or-tree;	or :	and-tree, or-tree	→	or-tree;
empty-and-tree :	and-tree;	and :	or-tree, and-tree	→	and-tree.

¹²⁶Actually, in the jargon of [Boyer and Moore, 1979; 1988b; 1998], the destructors are called *accessor functions*, and the type predicates are called *recognizer functions*.

respectively, which are the only Boolean values. Any argument with Boolean intention besides `F` is taken to be `T` (including `NIL`).

Instead of discussing the shell principle in detail with all its intricacies resulting from the untyped framework, we just present the first two shells:

1. The shell `(ADD1 X1)` of the *natural numbers*, with
 - type restriction `(NUMBERP X1)`,
 - base object `(ZERO)`, abbreviated by `0`,
 - destructor¹²⁶ `SUB1` with default value `0`, and
 - type predicate¹²⁶ `NUMBERP`.
2. The shell `(CONS X1 X2)` of *pairs*, with
 - destructors `CAR` with default value `0`,
`CDR` with default value `0`, and
 - type predicate `LISTP`.

According to the shell principle, these two shell declarations add axioms to the theory, which are equivalent to the following ones:

#	Axioms Generated by Shell <code>ADD1</code>	Axioms Generated by Shell <code>CONS</code>
0.1	$(\text{NUMBERP } X) = T \vee (\text{NUMBERP } X) = F$	$(\text{LISTP } X) = T \vee (\text{LISTP } X) = F$
0.2	$(\text{NUMBERP } (\text{ADD1 } X1)) = T$	$(\text{LISTP } (\text{CONS } X1 X2)) = T$
0.3	$(\text{NUMBERP } 0) = T$	
0.4	$(\text{NUMBERP } T) = F$	$(\text{LISTP } T) = F$
0.5	$(\text{NUMBERP } F) = F$	$(\text{LISTP } F) = F$
0.6		$(\text{LISTP } X) = F \vee (\text{NUMBERP } X) = F$
1	$(\text{ADD1 } (\text{SUB1 } X)) = X$ $\Leftarrow X \neq 0 \wedge (\text{NUMBERP } X) = T$	$(\text{CONS } (\text{CAR } X) (\text{CDR } X)) = X$ $\Leftarrow (\text{LISTP } X) = T$
2	$(\text{ADD1 } X1) \neq 0$	
3	$(\text{SUB1 } (\text{ADD1 } X1)) = X1$ $\Leftarrow (\text{NUMBERP } X1) = T$	$(\text{CAR } (\text{CONS } X1 X2)) = X1$ $(\text{CDR } (\text{CONS } X1 X2)) = X2$
4	$(\text{SUB1 } 0) = 0$	
5.1	$(\text{SUB1 } X) = 0 \Leftarrow (\text{NUMBERP } X) = F$	$(\text{CAR } X) = 0 \Leftarrow (\text{LISTP } X) = F$ $(\text{CDR } X) = 0 \Leftarrow (\text{LISTP } X) = F$
5.2	$(\text{SUB1 } (\text{ADD1 } X1)) = 0$ $\Leftarrow (\text{NUMBERP } X1) = F$	
L1 ¹²⁸	$(\text{ADD1 } X) = (\text{ADD1 } 0)$ $\Leftarrow (\text{NUMBERP } X) = F$	
L2 ¹²⁹	$(\text{NUMBERP } (\text{SUB1 } X)) = T$	

¹²⁷There are the following two different declarations for the shell `PACK`: In [Boyer and Moore, 1979], the shell `CONS` is defined after the shell `PACK` because `NIL` is the default value for the destructors `CAR` and `CDR`; moreover, `NIL` is an abbreviation for `(NIL)`, which is the base object of the shell `PACK`.

In [Boyer and Moore, 1988b; 1998], however, the shell `PACK` is defined after the shell `CONS`, we have $(\text{CAR } \text{NIL}) = 0$, the shell `PACK` has no base object, and `NIL` just abbreviates $(\text{PACK } (\text{CONS } 78 (\text{CONS } 73 (\text{CONS } 76 0))))$.

When we discuss the logic of [Boyer and Moore, 1979], we tacitly use the shells `CONS` and `PACK` as described in [Boyer and Moore, 1988b; 1998].

Note that the two occurrences of “(NUMBERP X1)” in Axioms 3 and 5.2 are exactly the ones that result from the type restriction of ADD1. Moreover, the occurrence of “(NUMBERP X)” in Axiom 0.6 is allocated at the right-hand side because the shell ADD1 is declared *before* the shell CONS.

Let us discuss the axioms generated by declaration of the shell ADD1. Roughly speaking, Axioms 0.1–0.3 are return-type declarations, Axioms 0.4–0.6 are about disjointness of types, Axiom 1 and Lemma L2 imply the axiom (nat1) from § 3.4, Axioms 2 and 3 imply axioms (nat2) and (nat3), respectively. Axioms 4 and 5.1–5.2 overspecify SUB1. Note that Lemma L1 is equivalent to 5.2 under 0.2–0.3 and 1–3.

Analogous to Lemma L1, every shell forces each argument not satisfying its type restriction into behaving like the default object of the argument’s destructor.

By contrast, the arguments of the shell CONS (just as every shell argument without type restriction) are not forced like this, and so — a clear advantage of the untyped framework — even objects of later defined shells (such as PACK) can be properly paired by the shell CONS. For instance, although NIL belongs to the shell PACK defined after the shell CONS (and so (CDR NIL) = 0),¹²⁷ we have (CAR (CONS NIL NIL)) = NIL by Axiom 3.

Nevertheless, the shell principle also allows us to declare a shell

(CONSNAT X1 X2)

of the *lists of natural numbers only* — similar to the ones of § 3.5 — say, with a type predicate LISTNATP, type restrictions (NUMBERP X1), (LISTNATP X2), base object (NILNAT), and destructors CARNAT, CDRNAT with default values 0, (NILNAT), respectively.

Let us now come to the admissible definitions of new functions in THM. In § 4 we have already discussed the *definition principle*¹³⁰ of THM in detail. The definition of recursive functions has not changed compared to the PURE LISP THEOREM PROVER besides that a function definition is admissible now only after a termination proof, which proceeds as explained in § 4.4. To this end, THM can apply its additional axiom of the well-foundedness of the irreflexive ordering LESSP on the natural numbers,¹³¹ and the theorem of the well-foundedness of the lexicographic combination of two well-founded orderings.

We again follow the Boyer–Moore waterfall (cf. Figure 1) and sketch how the stages of the waterfall are realized in THM in comparison to the PURE LISP THEOREM PROVER.

¹²⁸Proof of Lemma L1 from 0.2, 1–2, 5.2: Under the assumption of (NUMBERP X) = F, we show (ADD1 X) = (ADD1 (SUB1 (ADD1 X))) = (ADD1 0). The first step is a backward application of the conditional equation 1 via {X ↦ (ADD1 X)}, where the condition is fulfilled because of 2 and 0.2. The second step is an application of 5.2, where the condition is fulfilled by assumption.

¹²⁹Proof of Lemma L2 from 0.1–0.3, 1–4, 5.1–5.2 by *argumentum ad absurdum*: For a counterexample X, we get (SUB1 X) ≠ 0 by 0.3, as well as (NUMBERP (SUB1 X)) = F by 0.1. From the first we get X ≠ 0 by 4, and (NUMBERP X) = T by 5.1 and 0.1. Now we get the contradiction (SUB1 X) = (SUB1 (ADD1 (SUB1 X))) = (SUB1 (ADD1 0)) = 0; the first step is a backward application of the conditional equation 1, the second of L1, and the last of 3 (using 0.3).

¹³⁰Cf. [Boyer and Moore, 1979, p. 44f.].

5.3.1 Simplification in THM

We discussed simplification in the PURE LISP THEOREM PROVER in §5.2.1. Simplification in THM is covered in Chapters VI–IX of [Boyer and Moore, 1979], and the reader interested in the details is strongly encouraged to read these very well-written descriptions of heuristic procedures for simplification.

To compensate for the extra complication of the untyped approach in THM, which has a much higher number of interesting soft types than the PURE LISP THEOREM PROVER, soft-typing rules are computed for each new function symbol based on types that are disjunctions (actually: bit-vectors) of the following disjoint types: one for T, one for F, one for each shell, and one for objects not belonging to any of these.¹³² These soft-typing rules are pervasively applied in all stages of the theorem prover, which we cannot discuss here in detail. Some of these rules can be expressed in the LISP logic language as a theorem and presented in this form to the human users. Let us see two examples on this.

EXAMPLE 13. (continuing Example 6 of §4.1)
As THM knows (NUMBERP (FIX X)) and (NUMBERP (ADD1 X)), it produces the theorem (NUMBERP (PLUS X Y)) immediately after the termination proof for the definition of PLUS in Example 6. Note that this would neither hold in case of non-termination of PLUS, nor if there were a simple Y instead of (FIX Y) in the definition of PLUS. In the latter case, THM would only register that the return-type of PLUS is among NUMBERP and the types of its second argument Y. □

EXAMPLE 14. As THM knows that the type of APPEND is among LISTP and the type of its second argument, it produces the theorem (LISTP (FLATTEN X)) immediately after the termination proof for the following definition:

$$\begin{aligned} (\text{FLATTEN } X) = & (\text{IF } (\text{LISTP } X) \\ & (\text{APPEND } (\text{FLATTEN } (\text{CAR } X)) (\text{FLATTEN } (\text{CDR } X))) \\ & (\text{CONS } X \text{ NIL})) \end{aligned} \quad \square$$

¹³¹See Page 52f. of [Boyer and Moore, 1979] for the informal statement of this axiom on well-foundedness of LESSP.

Because THM is able to prove (LESSP X (ADD1 X)), well-foundedness of LESSP would imply — together with Axiom 1 and Lemma L2 — that THM admits only the standard model of the natural numbers, as explained in Note 40.

Matt Kaufmann, however, was so kind and made clear in a private e-mail communication that non-standard models are not excluded, because the statement “We assume LESSP to be a well-founded relation.” of [Boyer and Moore, 1979, p. 53] is actually to be read as the well-foundedness of the formal definition of §3.1 with the *additional assumption* that the predicate Q must be definable in THM.

Note that in Pieri’s argument on the exclusion of non-standard models (as described in Note 40), it is not possible to replace the reflexive and transitive closure of the successor relation s with the THM-definable predicate $\{ Y \mid (\text{NUMBERP } Y) = \text{T} \wedge ((\text{LESSP } Y \ X) = \text{T} \vee Y = X) \}$, because (by the THM-analog of axiom (lessp2’) of Example 12 in §5.2.6) this predicate will contain 0 as a minimal element even for a non-standard natural number X; thus, in non-standard models, LESSP is a *proper* super-relation of the reflexive and transitive closure of s.

¹³²See Chapter VI in [Boyer and Moore, 1979].

The standard representation of a propositional expression has improved from the multifarious LISP representation of the PURE LISP THEOREM PROVER toward today's standard of clausal representation. A *clause* is a disjunctive list of literals. *Literals*, however, deviating from the standard of being optionally negated atoms, are just LISP terms here, because every LISP function can be seen as a predicate.

This means that the “water” of the waterfall now consists of clauses, and the conjunction of all clauses in the waterfall represents the proof task.

Based on this clausal representation, we find a full-fledged description of *contextual rewriting* in Chapter IX of [Boyer and Moore, 1979], and its applications in Chapters VII–IX. This description comes some years before the term “contextual rewriting” became popular in automated theorem proving, and the term does not appear in [Boyer and Moore, 1979]. It is probably the first description of contextual rewriting in the history of logic, unless one counts the rudimentary contextual rewriting in the “reduction” of the PURE LISP THEOREM PROVER as such.¹³³

As indicated before, the essential idea of contextual rewriting is the following: While focusing on one literal of a clause for simplification, we can assume all other literals — the *context* — to be false, simply because the literal in focus is irrelevant otherwise. Especially useful are literals that are negated equations, because they can be used as a ground term-rewrite system. A non-equational literal t can always be taken to be the negated equation ($t \neq \mathbf{F}$). The free universal variables of a clause have to be treated as constants during contextual rewriting.¹³⁴

To bring contextual rewriting to full power, all occurrences of the function symbol **IF** in the literals of a clause are expelled from the literals as follows. If the condition of an **IF**-expression can be simplified to be definitely false **F** or definitely true (i.e. non-**F**, e.g. if **F** is not set in the bit-vector as a potential type), then the **IF**-expression is replaced with its respective case. Otherwise, after the **IF**-expression could not be removed by those rewrite rules for **IF** whose soundness depends on termination,¹³⁵ it is moved to the top position (outside-in), by replacing each case with itself in the **IF**'s context, such that the literal $C[(\mathbf{IF} \ t_0 \ t_1 \ t_2)]$ is intermediately replaced with $(\mathbf{IF} \ t_0 \ C[t_1] \ C[t_2])$, and then this literal splits its clause in two: one with the two literals (**NOT** t_0) and $C[t_1]$ in place of the old one, and one with t_0 and $C[t_2]$ instead.

¹³³Cf. § 5.2.1.

¹³⁴This has the advantage that we could take any well-founded ordering that is total on ground terms and run the terminating ground version of a Knuth–Bendix completion procedure [Knuth and Bendix, 1970] for all literals in a clause representation that have the form $l_i \neq r_i$, and replace the literals of this form with the resulting confluent and terminating rewrite system and normalize the other literals of the clause with it. Note that this transforms a clause into a logically equivalent one. None of the Boyer–Moore theorem provers does this, however.

¹³⁵These rewrite rules whose soundness depends on termination are $(\mathbf{IF} \ X \ Y \ Y) = Y$; $(\mathbf{IF} \ X \ X \ \mathbf{F}) = X$; and for Boolean X : $(\mathbf{IF} \ X \ T \ \mathbf{F}) = X$; tested for applicability in the given order.

THM eagerly removes variables in solved form: If the variable X does not occur in the term t , but the literal $(X \neq t)$ occurs in a clause, then we can remove that literal after rewriting all occurrences of X in the clause to t . This removal is a logical equivalence transformation, because the single remaining occurrence of X is implicitly universally quantified and so $(X \neq t)$ must be false because it implies $(t \neq t)$.

It now remains to describe the rewriting with function definitions and with lemmas tagged for rewriting, where the context of the clause is involved again.

Non-recursive function definitions are always unfolded by THM.

Recursive function definitions are treated in a way very similar to that of the PURE LISP THEOREM PROVER. The criteria on the unfolding of a function call of a recursively defined function f still depend solely on the terms introduced as arguments in the recursive calls of f in the body of f , which are accessed during the simplification of the body. But now, instead of rejecting the unfolding in case of the presence of new destructor terms in the simplified recursive calls, rejections are based on whether the simplified recursive calls contain subterms not occurring elsewhere in the clause. That is, an unfolding is approved if all subterms of the simplified recursive calls already occur in the clause. This basic *occurrence heuristic* is one of the keys to THM's success at induction. As we will see, instead of the PURE LISP THEOREM PROVER's phrasing of inductive arguments with "constructors in the conclusion", such as $P(s(x)) \Leftarrow P(x)$, THM uses "destructors in the hypothesis", such as $(P(x) \Leftarrow P(p(x))) \Leftarrow x \neq 0$. Thanks to the occurrence heuristic, the very presence of a well-chosen induction hypothesis gives the rewriter "permission" to unfold certain recursive functions in the induction conclusion (which is possible because all function definitions are in destructor style).

There are also two less important criteria which individually suffice to unblock the unfolding of recursive function definitions:

1. An increase of the number of arguments of the function to be unfolded that are constructor ground terms.
2. The second is a decrease of the number of function symbols in the arguments of the function to be unfolded at the measured positions of an induction template for that function.

So the clause

$$C[\text{lessp}(x, s(y))]$$

will be expanded by $(\text{lessp}2')$, $(\text{lessp}3')$, and $(p1)$ into the clauses

$$x \neq 0, C[\text{true}]$$

and

$$x = 0, C[\text{lessp}(p(x), y)]$$

— even if $p(x)$ is a newly occurring subterm! — because the second argument position of lessp is such a set of measured positions according to Example 18 of § 5.3.7.¹³⁶

THM is able to exploit previously proved lemmas. When the user submits a theorem for proof, the user tags it with tokens indicating how it is to be used in the future *if it is proved*. THM supports four tags and they indicate that the lemma is to be used as a rewrite rule, as a rule to eliminate destructors, as a rule to restrict generalizations, or as a rule to suggest inductions. The paradigm of tagging theorems for use by certain proof techniques focus the user on developing general “tactics” (within a very limited framework) while allowing the user to think mainly about relevant mathematical truths. This paradigm has been a hallmark of all Boyer–Moore theorem provers since THM and partially accounts for their reputation of being “automatic”.

Rewriting with lemmas that have been proved and tagged for rewriting — so-called *rewrite lemmas* — differs from rewriting with recursive function definitions mainly in one aspect: There is no need to block them because the user has tagged them explicitly for rewriting, and because rewrite lemmas have the form of conditional equations instead of unconditional ones. Simplification with lemmas tagged for rewriting and the heuristics behind the process are nicely described in [Schmidt-Samoa, 2006c], where a rewrite lemma is not just tagged for rewriting, but where the user can also mark the condition literals on how they should be dealt with. In THM there is no lazy rewriting with rewrite lemmas, i.e. no case splits are introduced to be able to apply the lemma.¹³⁷ This means that all conditions of the rewrite lemma have to be shown to be fulfilled in the current context. In partial compensation there is a process of backward chaining, i.e. the conditions can be shown to be fulfilled by the application of further conditional rewrite lemmas. The termination of this backward chaining is achieved by avoiding the generation of conditions into which the previous conditions can be homeomorphically embedded.¹³⁸ In addition, rewrite lemmas can introduce IF-expressions, splitting the rewritten clause into cases. There are provisions to instantiate extra variables of conditions eagerly, which is necessary because there are no existential variables.¹³⁹ Some collections of rewrite lemmas can cause THM’s rewriter not to terminate. For permutative rules like commutativity, however, termination is assured by simple term ordering heuristics.¹⁴⁰

5.3.2 Destructor Elimination in THM

We have already seen constructors such as `s` (in THM: `ADD1`) and `cons` (`CONS`) with the destructors `p` (`SUB1`) and `car` (`CAR`), and `cdr` (`CDR`), respectively.

¹³⁶See Page 118f. of [Boyer and Moore, 1979] for the details of the criteria for unblocking the unfolding of function definitions.

¹³⁷Matt Kaufmann and J Strother Moore added support for “forcing” and “case split” annotations to ACL2 in the mid-1990s.

¹³⁸See Page 109ff. of [Boyer and Moore, 1979] for the details.

¹³⁹See Page 111f. of [Boyer and Moore, 1979] for the details.

¹⁴⁰See Page 104f. of [Boyer and Moore, 1979] for the details.

EXAMPLE 15 (From Constructor to Destructor Style and back).

We have presented several function definitions both in constructor and in destructor style. Let us do careful and generalizable equivalence transformations (reverse step justified in parentheses) starting with the constructor-style rule (**ack3**) of § 3.4:

$$\text{ack}(s(x), s(y)) = \text{ack}(x, \text{ack}(s(x), y)).$$

Introduce (delete) the solved variables x' and y' for the constructor terms $s(x)$ and $s(y)$ occurring on the left-hand side, respectively, and add (delete) two further conditions by applying the definition (**p1'**) (cf. § 3.4) twice.

$$\text{ack}(s(x), s(y)) = \text{ack}(x, \text{ack}(s(x), y)) \Leftarrow \left(\begin{array}{l} x' = s(x) \wedge p(x') = x \\ \wedge y' = s(y) \wedge p(y') = y \end{array} \right).$$

Normalize the conclusion with leftmost equations of the condition from right to left (left to right).

$$\text{ack}(x', y') = \text{ack}(x, \text{ack}(x', y)) \Leftarrow \left(\begin{array}{l} x' = s(x) \wedge p(x') = x \\ \wedge y' = s(y) \wedge p(y') = y \end{array} \right).$$

Normalize the conclusion with rightmost equations of the condition from right to left (left to right).

$$\text{ack}(x', y') = \text{ack}(p(x'), \text{ack}(x', p(y'))) \Leftarrow \left(\begin{array}{l} x' = s(x) \wedge p(x') = x \\ \wedge y' = s(y) \wedge p(y') = y \end{array} \right).$$

Add (Delete) two conditions by applying axiom (**nat2**) twice.

$$\text{ack}(x', y') = \text{ack}(p(x'), \text{ack}(x', p(y'))) \Leftarrow \left(\begin{array}{l} x' = s(x) \wedge p(x') = x \wedge x' \neq 0 \\ \wedge y' = s(y) \wedge p(y') = y \wedge y' \neq 0 \end{array} \right).$$

Delete (Introduce) the leftmost equations of the condition by applying lemma (**s1'**) (cf. § 3.4) twice, and delete (introduce) the solved variables x and y for the destructor terms $p(x')$ and $p(y')$ occurring in the left-hand side of the equation in the conclusion, respectively.

$$\text{ack}(x', y') = \text{ack}(p(x'), \text{ack}(x', p(y'))) \Leftarrow x' \neq 0 \wedge y' \neq 0.$$

Up to renaming of the variables, this is the destructor-style rule (**ack3'**) of Example 11 (cf. § 5.2.6). \square

Our data types are defined inductively over constructors.¹⁴¹ Therefore constructors play the main rôle in our semantics, and practice shows that step cases of simple induction proofs work out much better with constructors than with the respective destructors, which are secondary (i.e. defined) operators in our semantics and have a more complicated case analysis in application.

¹⁴¹Here the term “inductive” means the following: We start with the empty set and take the smallest fixpoint under application of the constructors, which contains only finite structures, such as natural numbers and lists. Co-inductively over the destructors we would obtain different data types, because we start with the universal class and obtain the greatest fixed point under inverse application of the destructors, which typically contains infinite structures. For instance, for the unrestricted destructors `car`, `cdr` of the list of natural numbers `list(nat)` of § 3.5, we co-inductively obtain the data type of infinite streams of natural numbers.

There are **two further positive effects** of destructor elimination:

1. It tends to standardize the representation of a clause in the sense that the numbers of occurrences of identical subterms tend to be increased.
2. Destructor elimination also brings the subterm property in line with the sub-structure property; e.g., Y is both a sub-structure of $(\text{CONS } X \ Y)$ and a subterm of it, whereas $(\text{CDR } Z)$ is a sub-structure of Z in case of $(\text{LISTP } Z)$, but not a subterm of Z .

Both effects improve the chances that the clause passes the follow-up stages of cross-fertilization and generalization with good success.¹⁴²

As noted earlier, the PURE LISP THEOREM PROVER does induction using step cases with constructors, such as $P(s(x)) \Leftarrow P(x)$, whereas THM does induction using step cases with destructors, such as

$$(P(x) \Leftarrow P(p(x))) \Leftarrow x \neq 0.$$

So destructor elimination was not so urgent in the PURE LISP THEOREM PROVER, simply because there were fewer destructors around. Indeed, the stage “destructor elimination” does not exist in the PURE LISP THEOREM PROVER.

THM does not do induction with constructors because there are generalized destructors that do not have a straightforward constructor (see below), and because the induction rule of explicit induction has to fix in advance whether the step cases are destructor or constructor style. So with destructor style in all step cases and in all function definitions, explicit induction and recursion in THM choose the style that is always applicable. Destructor elimination then confers the advantages of constructor-style proofs when possible.

EXAMPLE 16 (A Generalized Destructor Without Constructor).

A generalized destructor that does not have a straightforward constructor is the function `delfirst` defined in §3.5. To verify the correctness of a deletion-sort algorithm based on `delfirst`, a useful step case for an induction proof is of the form¹⁴³

$$(P(l) \Leftarrow P(\text{delfirst}(\text{max}(l), l))) \Leftarrow l \neq \text{nil}.$$

A constructor version of this induction scheme would need something like an insertion function with an additional free variable indicating the position of insertion — a complication that further removes the proof obligations from the algorithm being verified. \square

Proper destructor functions take only one argument. The generalized destructor `delfirst` we have seen in Example 16 has actually two arguments; the second one is the *proper destructor argument* and the first is a *parameter*. After the elimination of a set of destructors, the terms at the parameter positions of the destructors are typically still present, whereas the terms at the proper destructor argument are removed.

¹⁴²See Page 114ff. of [Boyer and Moore, 1979] for a nice example for the advantage of destructor elimination for cross-fertilization.

¹⁴³See Page 143f. of [Boyer and Moore, 1979].

EXAMPLE 17 (Division with Remainder as a pair of Generalized Destructors). In case of $y \neq 0$, we can construct each natural number x in the form of $(q * y) + r$ with $\text{lessp}(r, y) = \text{true}$. The related generalized destructors are the quotient $\text{div}(x, y)$ of x by y , and its remainder $\text{rem}(x, y)$. Note that in both functions, the first argument is the proper destructor argument and the second the parameter, which must not be 0. The rôle that the definition (p1') and the lemma (s1') of § 3.4 play in Example 15 (and which the definitions (car1'), (cdr1') and the lemma (cons1') of § 3.5 play in the equivalence transformations between constructor and destructor style for lists) is here taken by the following lemmas on the generalized destructors div and rem and on the generalized constructor $\lambda q, r. ((q * y) + r)$:

$$\begin{array}{ll} (\text{div1}') & \text{div}(x, y) = q \Leftarrow y \neq 0 \wedge (q * y) + r = x \wedge \text{lessp}(r, y) = \text{true} \\ (\text{rem1}') & \text{rem}(x, y) = r \Leftarrow y \neq 0 \wedge (q * y) + r = x \wedge \text{lessp}(r, y) = \text{true} \\ (+9') & (q * y) + r = x \Leftarrow y \neq 0 \wedge q = \text{div}(x, y) \wedge r = \text{rem}(x, y) \end{array}$$

If we have a clause with the literal $y = 0$, in which the destructor terms $\text{div}(x, y)$ or $\text{rem}(x, y)$ occur, we can — just as in the of Example 15 (reverse direction) — introduce the new literals $\text{div}(x, y) \neq q$ and $\text{rem}(x, y) \neq r$ for fresh q, r , and apply lemma (+9') to introduce the literal $x \neq (q * y) + r$. Then we can normalize with the first two literals, and afterwards with the third. Then all occurrences of $\text{div}(x, y)$, $\text{rem}(x, y)$, and x are gone.¹⁴⁴ \square

To enable the form of elimination of generalized destructors described in Example 17, THM allows the user to tag lemmas of the form (s1'), (cons1'), or (+9') as *elimination lemmas* to perform destructor elimination. In clause representation, this form is in general the following: The first literal of the clause is of the form $(t^c = x)$, where x is a variable which does not occur in the (generalized) constructor term t^c . Moreover, t^c contains some distinct variables y_0, \dots, y_n , which occur only on the left-hand sides of the first literal and of the last $n+1$ literals of the clause, which are of the form $(y_0 \neq t_0^d), \dots, (y_n \neq t_n^d)$, for distinct (generalized) destructor terms t_0^d, \dots, t_n^d .¹⁴⁵

The idea of application for destructor elimination in a given clause is, of course, the following: If, for an instance of the elimination lemma, the literals not mentioned above (i.e. in the middle of the clause, such as $y \neq 0$ in (+9')) occur in the given clause, and if t_0^d, \dots, t_n^d occur in the given clause as subterms, then rewrite all their occurrences with $(y_0 \neq t_0^d), \dots, (y_n \neq t_n^d)$ from right to left and then use the first literal of the elimination lemma from right to left for further normalization.¹⁴⁶

¹⁴⁴For a nice, but non-trivial example on why proofs tend to work out much easier after this transformation, see Page 135ff. of [Boyer and Moore, 1979].

¹⁴⁵THM adds one more restriction here, namely that the generalized destructor terms have to consist of a function symbol applied to a list containing exactly the variables of the clause, besides y_0, \dots, y_n .

Moreover, note that THM actually does not use our flattened form of the elimination lemmas, but the one that results from replacing each y_i in the clause with t_i^d , and then removing the literal $(y_i \neq t_i^d)$. Thus, THM would accept only the non-flattened versions of our elimination lemmas, such as (s1) instead of (s1') (cf. § 3.4), and such as (cons1) instead of (cons1') (cf. § 3.5).

After a clause enters the destructor-elimination stage of THM, its most simple (actually: the one defined first) destructor that can be eliminated is eliminated, and destructor elimination is continued until all destructor terms introduced by destructor elimination are eliminated if possible. Then, before further destructors are eliminated, the resulting clause is returned to the center pool of the waterfall. So the clause will enter the simplification stage where the (generalized) constructor introduced by destructor elimination may be replaced with a (generalized) destructor. Then the resulting clauses re-enter the destructor-elimination stage, which may result in infinite looping.

For example, destructor elimination turns the clause

$$x' = 0, C[\text{lessp}(\mathbf{p}(x'), x')], C'[\mathbf{p}(x'), x']$$

by the elimination lemma (s1) into the clause

$$\mathbf{s}(x) = 0, C[\text{lessp}(x, \mathbf{s}(x))], C'[x, \mathbf{s}(x)].$$

Then, in the simplification stage of the waterfall, $\text{lessp}(x, \mathbf{s}(x))$ is unfolded, resulting in the clause

$$x = 0, C[\text{lessp}(\mathbf{p}(x), x)], C'[x, \mathbf{s}(x)]$$

and another one.¹⁴⁷

Looping could result from eliminating the destructor introduced by simplification (such as it is actually the case for our destructor \mathbf{p} in the last clause). To avoid looping, before returning a clause to the center pool of the waterfall, the variables introduced by destructor elimination (such as our variable x) are marked. (Generalized) destructor terms containing marked variables are blocked for further destructor elimination. This marking is removed only when the clause reaches the induction stage of the waterfall.¹⁴⁸

5.3.3 (Cross-) Fertilization in THM

This stage has already been described in § 5.2.3 because there is no noticeable difference between the PURE LISP THEOREM PROVER and THM here, besides some heuristic fine tuning.¹⁴⁹

¹⁴⁶If we add the last literals of the elimination lemma to the given clause, use them for contextual rewriting, and remove them only if this can be achieved safely via application of the definitions of the destructors (as we could do in all our examples), then the elimination of destructors is an equivalence transformation. Destructor elimination in THM, however, may (over-) generalize the conjecture, because these last literals are not present in the non-flattened elimination lemma of THM and its variables y_i are actually introduced in THM by generalization. Thus, instead of trying to delete the last literals of our deletion lemmas safely, THM never adds them.

¹⁴⁷The latter step is given in more detail in the context of the second of the two less important criteria of § 5.3.1 for unblocking the unfolding of $\text{lessp}(x, \mathbf{s}(y))$.

¹⁴⁸See Page 139 of [Boyer and Moore, 1979]. In general, for more sophisticated details of destructor elimination in THM, we have to refer the reader to Chapter X of [Boyer and Moore, 1979].

¹⁴⁹See Page 149 of [Boyer and Moore, 1979].

5.3.4 Generalization in THM

THM adds only one new rule to the universally applicable heuristic rules for generalization on a term t mentioned in § 3.9:

“Never generalize on a destructor term t !”

This new rule makes sense in particular after the preceding stage of destructor elimination in the sense that destructors that outlast their elimination probably carry some relevant information. Another reason for not generalizing on destructor terms is that the clause will enter the center pool in case another generalization is possible, and then the destructor elimination might eliminate the destructor term more carefully than generalization would do.¹⁵⁰

The main improvement of generalization in THM over the PURE LISP THEOREM PROVER, however, is the following: Suppose again that the term t is to be replaced at all its occurrences in the clause $T[t]$ with the fresh variable z . Recall that the PURE LISP THEOREM PROVER restricts the fresh variable with a predicate synthesized from the definition of the top function symbol of the replaced term. THM instead restricts the new variable in two ways. Both ways add additional literals to the clause before the term is replaced by the fresh variable:

1. Assuming all literals of the clause $T[t]$ to be false (i.e. of type F), the bit-vector describing the soft type of t is computed and if only one bit is set, then, for the respective type predicate, say the bit expressing NUMBERP, then a new literal is added to the clause, such as (NOT (NUMBERP t)).
2. The user can tag certain lemmas as *generalization lemmas*; such as
 (SORTEDP (SORT X))
 for a sorting function SORT; and if (SORT X) matches t , the respective instance of (NOT (SORTEDP (SORT X))) is added to $T[t]$.¹⁵¹ In general, for the addition of such a literal (NOT t''), a proper subterm t' of a generalization lemma must match t .¹⁵²

¹⁵⁰See Page 156f. of [Boyer and Moore, 1979].

¹⁵¹Cf. Note 114.

¹⁵²Moreover, the literal is actually added to the generalized clause only if the top function symbol of t does no longer occur in the literal after replacing t with x . This means that, for a generalization lemma (EQUAL (FLATTEN (GOPHER X)) (FLATTEN X)), the literal
 (NOT (EQUAL (FLATTEN (GOPHER t'')) (FLATTEN t'')))
 is added to $T[t]$ in case of t being of the form (GOPHER t'''), but not in case of t being of the form (FLATTEN t''') where the first occurrence of FLATTEN is not removed by the generalization. See Page 156f. of [Boyer and Moore, 1979] for the details.

5.3.5 Elimination of Irrelevance in THM

THM includes another waterfall stage not in the PURE LISP THEOREM PROVER, the elimination of irrelevant literals. This is the last transformation before we come to “induction”. Like generalization, this stage may turn a valid clause into an invalid one. The main reason for taking this risk is that the subsequent heuristic procedures for induction assume all literals to be relevant: irrelevant literals may suggest inappropriate induction schemes which may result in a failure of the induction proof. Moreover, if all literals seem to be irrelevant, then the goal is probably invalid and we should not do a costly induction but just fail immediately.¹⁵³

Let us call two literals *connected* if there is a variable that occurs in both of them. Consider the partition of a clause into its equivalence classes w.r.t. the reflexive and transitive closure of connectedness. If we have more than one equivalence class in a clause, this is an alarm signal for irrelevance: if the original clause is valid, then a sub-clause consisting only of the literals of one of these equivalence classes must be valid as well. This is a consequence of the logical equivalence of $\forall x. (A \vee B)$ with $A \vee \forall x. B$, provided that x does not occur in A . Then we should remove one of the irrelevant equivalence classes after the other from the original clause. To this end, THM has two heuristic tests for irrelevance.


1. An equivalence class of literals is irrelevant if it does not contain any properly recursive function symbol.

Based on the assumption that the previous stages of the waterfall are sufficiently powerful to prove clauses composed only of **primitive** functions, the justification for this heuristic test is the following: If the clause of the equivalence class were valid, then the previous stages of the waterfall should already have established the validity of this equivalence class.

2. An equivalence class of literals is irrelevant if it consists of only one literal and if this literal is the application of a properly recursive function to a list of distinct variables.

Based on the assumption that the soft typing rules are sufficiently powerful and that the user has not defined a tautological, but tricky predicate,¹⁵⁴ the justification for this heuristic test is the following: The bit-vector of this literal must contain **the singleton type of F**; otherwise the validity of the literal and the clause would have been recognized by the stage “simplification”. This means that **F** is most probably a possible value for some combination of arguments.

¹⁵³See Page 160f. of [Boyer and Moore, 1979] for a typical example of this.

¹⁵⁴This assumption is critical because it often occurs that updated program code contains recursive predicates that are actually trivially true, but very tricky. See § 3.2 of [Wirth, 2004] for such an example. 

5.3.6 Induction in THM as compared to the PURE LISP THEOREM PROVER

As we have seen in §5.2.6, the *recursion analysis* in the PURE LISP THEOREM PROVER is only rudimentary. Indeed, the whole information on the body of the recursive function definitions comes out of the **poor feedback** of the “evaluation” procedure of the simplification stage of the PURE LISP THEOREM PROVER. Roughly speaking, this information consists only in the two facts

1. that a destructor symbol occurring as an argument of the recursive function call in the body is not removed by the “evaluation” procedure in the context of the current goal and in the local environment, and
2. that it is not possible to derive that this recursive function call is unreachable in this context and environment.

In THM, however, the first part of recursion analysis is done at *definition time*, i.e. at the time the function is defined, and applied at *proof time*, i.e. at the time the induction rule produces the base and step cases. Surprisingly, there is no reachability analysis for the recursive calls in this second part of the recursion analysis in THM. While the information in item 1 is thoroughly improved as compared to the PURE LISP THEOREM PROVER, the information in item 2 is partly weaker because all recursive function calls are assumed to be reachable during recursion analysis. The overwhelming success of THM means that the heuristic decision to abandon reachability analysis in THM was appropriate.¹⁵⁵

5.3.7 Induction Templates generated by Definition-Time Recursion Analysis

The first part of recursion analysis in THM consists **in** a termination analysis of every recursive function at the time of its definition. The system does not only look for one termination proof that is sufficient for the admissibility of the function definition, but actually looks through **all termination proofs** in a finite search space and gathers from them all information required for justifying the termination of the recursive function definition, as well as for justifying the soundness and for improving the feasibility of the step cases to be generated by the induction rule.

To this end, THM constructs valid induction templates very similar to our description in §4.4.¹⁵⁶ Let us approach the idea of a valid induction template with some typical examples, which are actually the templates for the constructor-style examples of §4.4, but now for the destructor-style definitions of `lessp` and `ack`, because only destructor-style definitions are admissible in THM.

¹⁵⁵Note that in most cases the step formula of the reachable cases works somehow in THM, as long as no better step case was canceled because of unreachable step cases, which, of course, are trivial to prove, simply because their condition is false. Moreover, note that, contrary to *descente infinie* which can get along with the first part of recursion analysis alone, the heuristics of explicit induction have to guess the induction steps eagerly, which is always a fault-prone procedure, to be corrected by additional induction proofs, as we have seen in Example 4 of §3.8.1.

EXAMPLE 18 (Two Induction Templates with different Measured Positions).
 For the ordering predicate `lessp` as defined by (`lessp1'-3'`) in Example 12 of § 5.2.6, we get two induction templates with the sets of measured positions $\{1\}$ and $\{2\}$, respectively, both for the well-founded ordering $\lambda x, y. (\text{lessp}(x, y) = \text{true})$. The first template has the weight term (1) and the relational description

$$\{ (\text{lessp}(x, y), \{ \text{lessp}(\text{p}(x), \text{p}(y)) \}, \{ x \neq 0 \}) \}.$$

The second one has the weight term (2) and the relational description

$$\{ (\text{lessp}(x, y), \{ \text{lessp}(\text{p}(x), \text{p}(y)) \}, \{ y \neq 0 \}) \}. \quad \square$$

EXAMPLE 19 (One Induction Template with Two Measured Positions).

For the Ackermann function `ack` as defined by (`ack1'-3'`) in Example 11 of § 5.2.6, we get only one appropriate induction template. The set of its measured positions is $\{1, 2\}$, because of the weight function `cons((1), cons((2), nil))` (in THM actually: (`CONS x y`)) in the well-founded lexicographic ordering

$$\lambda l, k. (\text{lexlimless}(l, k, \text{s}(\text{s}(0)))) = \text{true}.$$

The relational description has two elements: For the equation (`ack2'`) we get

$$(\text{ack}(x, y), \{ \text{ack}(\text{p}(x), \text{s}(0)) \}, \{ x \neq 0 \}),$$

and for the equation (`ack3'`) we get

$$(\text{ack}(x, y), \{ \text{ack}(x, \text{p}(y)), \text{ack}(\text{p}(x), \text{ack}(x, \text{p}(y))) \}, \{ x \neq 0, y \neq 0 \}). \quad \square$$

To find valid induction templates automatically by exhaustive search, THM allows the user to tag certain theorems as “*induction lemmas*”. An induction lemma consists of the application of a well-founded relation to two terms with the same top function symbol w , playing the rôle of the weight term; plus a condition without extra variables, which is used to generate the case conditions of the induction template. Moreover, the arguments of the application of w occurring as the second argument of the well-founded relation must be distinct variables in THM, mirroring the left-hand side of its function definitions in destructor style.

Certain induction lemmas are generated with each shell declaration. Such an induction lemma generated for the shell `ADD1`, which is roughly

$$(\text{LESSP} (\text{COUNT} (\text{SUB1} X)) (\text{COUNT} X)) \leftarrow (\text{NOT} (\text{ZEROP} X)),$$

suffices for generating the two templates of Example 18. Note that `COUNT`, playing the rôle of w here, is a special function in THM, which is generically extended by every shell declaration in an object-oriented style for the elements of the new shell. On the natural numbers here, `COUNT` is the identity. On other shells, `COUNT` is defined similar to our function `count` from § 3.5.¹⁵⁷

¹⁵⁶Those parts of the condition of the equation that contain the new function symbol f must be ignored in the case conditions of the induction template because the definition of the function f is admitted in THM only *after* it has passed the termination proof.

That THM ignores the governing conditions that contain the new function symbol f is described in the 2nd paragraph on Page 165 of [Boyer and Moore, 1979]. Moreover, an example for this is the definition of `OCCUR` on Page 166 of [Boyer and Moore, 1979].

After one successful termination proof, however, the function can be admitted in THM, and then these conditions could actually be admitted in the templates. So the actual reason why THM ignores these conditions in the templates is that it generates the templates with the help of previously proved *induction lemmas*, which, of course, cannot contain the new function yet.

5.3.8 Proof-Time Recursion Analysis in THM

The induction rule uses the information from the induction templates as follows: For each recursive function occurring in the input formula, all *applicable* induction templates are retrieved and turned into *induction schemes* as described in §4.7. Any induction scheme that is *subsumed* by another one is deleted after adding its hitting ratio to the one of the other. The remaining schemes are *merged* into new ones with a higher hitting ratio, and finally, after the *flawed* schemes are deleted, **the scheme with the highest hitting ratio will be used by the induction rule** to generate the base and step cases.

EXAMPLE 20 (Applicable Induction Templates).

Let us consider the conjecture (ack4) from §3.4. From the three induction templates of Examples 18 and 19, only the second one of Example 18 is not applicable because the second position of `lessp` (which is the only measured position of that template) is changeable, but filled in (ack4) by the non-variable `ack(x, y)`. \square

From the destructor-style definitions (`lessp1'-3'`) (cf. Example 12) and (`ack1'-3'`) (cf. Example 11), we have generated two induction templates applicable to

(ack4) `lessp(y, ack(x, y)) = true`

They yield the two induction schemes of Example 21. See also Example 10 for the single induction scheme for the constructor-style definitions (`lessp1-3`) and (`ack1-3`).

EXAMPLE 21 (Induction Schemes).

The induction template for `lessp` of Example 18 that is applicable to (ack4) according to Example 20 and whose relational description contains only the triple

$$(\text{lessp}(x, y), \{ \text{lessp}(p(x), p(y)) \}, \{ x \neq 0 \})$$

yields the induction scheme with position set $\{1.1\}$ (i.e. left-hand side of first literal in (ack4)); the step-case description is $\{(\{x, y\} \upharpoonright \text{id}, \{\mu_1\}, \{y \neq 0\})\}$, where $\mu_1 = \{x \mapsto x, y \mapsto p(y)\}$; the set of induction variables is $\{y\}$; and the hitting ratio is $\frac{1}{2}$.

This can be seen as follows: The substitution called ξ in the discussion of §4.7 can be chosen to be the identity substitution $\{x, y\} \upharpoonright \text{id}$ on $\{x, y\}$ because the first element of the triple does not contain any constructors. This is always the case for induction templates for destructor-style definitions such as (`lessp1'-3'`). The substitution called σ in §4.7 (which has to match the first element of the triple to the term (ack4)/1.1, i.e. the term at the position 1.1 in (ack4)) is $\sigma = \{x \mapsto y, y \mapsto \text{ack}(x, y)\}$. So the constraints for μ_1 (which tries to match (ack4)/1.1 to the σ -instance of the second element of the triple) are: $y\mu_1 = p(y)$ for the first (measured) position of `lessp`; and $\text{ack}(x, y)\mu_1 = p(\text{ack}(x, y))$ for the second (unmeasured) position, which cannot be achieved and is skipped. This results in a hitting ratio of only $\frac{1}{2}$. The single measured position 1 of the induction template results in the induction variable (ack4)/1.1.1 = y .

¹⁵⁷For more details on the recursion analysis a definition time in THM, see Page 180ff. of [Boyer and Moore, 1979].

The template for `ack` of Example 19 yields an induction scheme with the position set $\{1.1.2\}$, and the set of induction variables $\{x, y\}$. The triple

$$(\text{ack}(x, y), \{\text{ack}(p(x), s(0))\}, \{x \neq 0\})$$

(generated by the equation (`ack2'`)) is replaced with $(\{x, y\} \upharpoonright \text{id}, \{\mu'_{1,1}\}, \{x \neq 0\})$, where $\mu'_{1,1} = \{x \mapsto p(x), y \mapsto s(0)\}$. The triple

$$(\text{ack}(x, y), \{\text{ack}(x, p(y)), \text{ack}(p(x), \text{ack}(x, p(y)))\}, \{x \neq 0, y \neq 0\})$$

(generated by (`ack3'`)) is replaced with $(\{x, y\} \upharpoonright \text{id}, \{\mu'_{2,1}, \mu'_{2,2}\}, \{x \neq 0, y \neq 0\})$, where $\mu'_{2,1} = \{x \mapsto x, y \mapsto p(y)\}$, and $\mu'_{2,2} = \{x \mapsto p(x), y \mapsto \text{ack}(x, p(y))\}$.

This can be seen as follows: The substitution called σ in the above discussion is $\{x, y\} \upharpoonright \text{id}$ in both cases, and so the constraints for the (measured) positions are $x\mu'_{1,1} = p(x)$, $y\mu'_{1,1} = s(0)$; $x\mu'_{2,1} = x$, $y\mu'_{2,1} = p(y)$; $x\mu'_{2,2} = p(x)$, $y\mu'_{2,2} = \text{ack}(x, p(y))$.

As all six constraints are satisfied, the hitting ratio is $\frac{6}{6} = 1$. \square

An induction scheme that is either *subsumed by* or *merged into* another induction scheme adds its hitting ratio and sets of positions and induction variables to those of the other's, respectively, and then it is deleted.

The most important case of subsumption are schemes that are identical except for their position sets, where — no matter which scheme is deleted — the result is the same. The more general case of proper subsumption occurs when the subsumer provides the essential structure of the subsumee, but not vice versa.

Merging and proper subsumption of schemes — seen as binary algebraic operations — are not commutative, however, because the second argument inherits the well-foundedness guarantee alone and somehow absorbs the first argument, and so the result for swapped arguments is often undefined.

More precisely, subsumption is given if the step-case description of the first induction scheme can be injectively mapped to the step-case description of the second one, such that (using the notation of § 4.7 and Example 21), for each step case $(\text{id}, \{\mu_j \mid j \in J\}, C)$ mapped to $(\text{id}, \{\mu'_j \mid j \in J \uplus J'\}, C')$, we have $C \subseteq C'$, and the set of substitutions $\{\mu_j \mid j \in J\}$ can be injectively¹⁵⁸ mapped to $\{\mu'_j \mid j \in J \uplus J'\}$ (w.l.o.g. say μ_i to μ'_i for $i \in J$), such that, for each $j \in J$ and $x \in \text{dom}(\mu_j)$: $x \in \text{dom}(\mu'_j)$; $x\mu_j = x$ implies $x\mu'_j = x$; and $x\mu_j$ is a subterm of $x\mu'_j$.

EXAMPLE 22 (Subsumption of Induction Schemes).

In Example 21, the induction scheme for `lessp` is subsumed by the induction scheme for `ack`, because we can map the only element of the step-case description of the former to the second element of the step-case description of latter: the case condition $\{y \neq 0\}$ is a subset of the case condition $\{x \neq 0, y \neq 0\}$, and we have $\mu_1 = \mu'_{2,1}$. So the former scheme is deleted and the scheme for `ack` is updated to have the position set $\{1.1, 1.1.2\}$ and the hitting ratio $\frac{3}{2}$. \square

¹⁵⁸From a logical viewpoint, it is not clear why this second injectivity requirement is found here, just as in different (but equivalent) form in [Boyer and Moore, 1979, p. 191, 1st paragraph]. (The first injectivity requirement may prevent us from choosing an induction ordering that is too small, cf. § 5.3.9.) An omission of the second requirement would just admit a term of the subsumer to have multiple subterms of the subsumee, which seems reasonable. Nevertheless, as pointed out in § 5.3.9, only practical testing of the heuristics is what matters here. See also Note 159.

In Example 12 of § 5.2.6 we have already seen a rudimentary, but pretty successful kind of *merging of suggested step cases* in the PURE LISP THEOREM PROVER. As THM additionally has induction schemes, it applies a more sophisticated *merging of induction schemes* instead.

Two substitutions μ_1 and μ_2 are [*non-trivially*] *mergeable* if $x\mu_1 = x\mu_2$ for each $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ [and there is a $y \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ with $y\mu_1 \neq y$].

Two triples $(V_1 \upharpoonright \text{id}, A_1, C_1)$ and $(V_2 \upharpoonright \text{id}, A_2, C_2)$ of two step-case descriptions of two induction schemes, each with domain $V_k = \text{dom}(\mu_k)$ for all $\mu_k \in A_k$ (for $k \in \{1, 2\}$), are [*non-trivially*] *mergeable* if for each $\mu_1 \in A_1$ there is a $\mu_2 \in A_2$ such that μ_1 and μ_2 are [*non-trivially*] mergeable. The result of their merging is $(V_1 \cup V_2 \upharpoonright \text{id}, m(A_1, A_2), C_1 \cup C_2)$, where $m(A_1, A_2)$ is the set containing all substitutions $\mu_1 \cup \mu_2$ with $\mu_1 \in A_1$ and $\mu_2 \in A_2$ such that μ_1 and μ_2 are mergeable as well as all substitutions $V_1 \setminus V_2 \upharpoonright \text{id} \cup \mu_2$ with $\mu_2 \in A_2$ for which there is no substitution $\mu_1 \in A_1$ such that μ_1 and μ_2 are mergeable.

Two induction schemes are *mergeable* if the step-case description of the first induction scheme can be injectively¹⁵⁹ mapped to the step-case description of the second one, such that each argument and its image are non-trivially mergeable. The step-case description of the induction scheme that results from *merging the first induction scheme into the second* contains the merging of all mergeable triples of the step-case descriptions of first and second induction scheme, respectively.

Finally, we have to describe what it means that an induction scheme is *flawed*. This simply is the case if — after merging is completed — the intersection of its induction variables with the (common) domain of the substitutions of the step-case description of another remaining induction scheme is non-empty.

If an induction scheme is flawed by another one that cannot be merged with it, this indicates that an induction on it will probably result in a permanent clash between the induction conclusion and the available induction hypotheses at some occurrences of the induction variables.¹⁶⁰

	pos. set	ind. var.s	step-case description	hit. ratio
1	{1}	{x}	{({x,z}↑id, {μ ₁ }, {x ≠ 0})}	1
2	{2}	{x}	{({x,y}↑id, {μ ₂ }, {x ≠ 0})}	1
3	{2}	{y}	{({x,y}↑id, {μ ₂ }, {y ≠ 0})}	1
4	{3}	{y}	{({y,z}↑id, {μ ₃ }, {y ≠ 0})}	1
5	{3}	{z}	{({y,z}↑id, {μ ₃ }, {z ≠ 0})}	1
6	{2}	{x, y}	{({x,y}↑id, {μ ₂ }, {x ≠ 0, y ≠ 0})}	2
7	{3}	{y, z}	{({y,z}↑id, {μ ₃ }, {y ≠ 0, z ≠ 0})}	2
8	{2, 3}	{x, y, z}	{({x,y,z}↑id, {μ ₄ }, {x ≠ 0, y ≠ 0, z ≠ 0})}	4
9	{1, 2, 3}	{x, y, z}	{({x,y,z}↑id, {μ ₄ }, {x ≠ 0, y ≠ 0, z ≠ 0})}	5

$$\begin{aligned} \mu_1 &= \{x \mapsto p(x), z \mapsto p(z)\}, & \mu_2 &= \{x \mapsto p(x), y \mapsto p(y)\}, \\ \mu_3 &= \{y \mapsto p(y), z \mapsto p(z)\}, & \text{and} & \mu_4 = \{x \mapsto p(x), y \mapsto p(y), z \mapsto p(z)\}. \end{aligned}$$

pos. = position; ind. var.s = set of induction variables; hit. = hitting.

Figure 3. The induction schemes of Example 23

EXAMPLE 23 (Merging and Flawedness of Induction Schemes).

Let us reconsider merging in the proof of lemma (lessp7) w.r.t. the definition of lessp via (lessp1'–3'), just as we did in Example 12. Let us abbreviate $p = \text{true}$ with p , just as in our very first proof of lemma (lessp7) in Example 3, and also following the LISP style of THM. Simplification reduces (lessp7) first to the clause

(lessp7') $\text{lessp}(x, p(z)), \neg\text{lessp}(x, y), \neg\text{lessp}(y, z), z = 0$

Then the Boyer–Moore waterfall sends this clause through three rounds of reduction between destructor elimination and simplification as discussed at the end of § 5.3.2, finally returning again to (lessp7'), but now with all its variables marked as being introduced by destructor elimination, which prevents looping by blocking further destructor elimination.

Note that the marked variables refer actually to the predecessors of the values of the original lemma (lessp7'), and that these three rounds of reduction already include all that is required for the entire induction proof, such that *descente infinie* would now conclude the proof with an induction-hypothesis application. This most nicely illustrates the crucial similarity between recursion and induction, which Boyer and Moore “exploit” . . . “or, rather, contrived”.¹⁶¹

The proof by explicit induction in THM, however, now just starts to compute induction schemes. The two induction templates for lessp found in Example 18 are applicable five times, resulting in the induction schemes 1–5 in Figure 3.

From the domains of the substitutions in the step-case descriptions, it is obvious that — among schemes 1–5 — only the two pairs of schemes 2 and 3 as well as 4 and 5 are candidates for subsumption, which is not given here, however, because the case conditions of these two pairs of schemes are not subsets of each other.

Nevertheless, these pairs of schemes merge, resulting in the schemes 6 and 7, respectively, which merge again, resulting in scheme 8.

Now only the schemes 1 and 8 remain. As each of them has x as an induction variable, both schemes would be flawed if they could not be merged.

It does not matter that the scheme 1 is subsumed by scheme 8 simply because the phase of subsumption is already over; but they are also mergeable, actually with the same result as subsumption would have, namely the scheme 9, which admits us to prove the generic step-case formula it describes without further induction, and so THM achieves the crucial task of heuristic anticipation of an appropriate induction hypotheses, just as well as the PURE LISP THEOREM PROVER.¹⁶² \square

¹⁵⁹From a logical viewpoint, it is again not clear why and injectivity requirement is found here, just as in different (but equivalent) form in [Boyer and Moore, 1979, p. 193, 1st paragraph]. An omission of the injectivity requirement would admit to define merging as a commutative associative operation. Nevertheless, as pointed out in § 5.3.9, only practical testing of the heuristics is what matters here. See also Note 158.

¹⁶⁰See Page 194f. of [Boyer and Moore, 1979] for a short further discussion and a nice example.

¹⁶¹Cf. [Boyer and Moore, 1979, p. 163, last paragraph].

¹⁶²The base cases show no improvement to the proof with the PURE LISP THEOREM PROVER in Example 12 and a further additional, but also negligible overhead is the preceding reduction from (lessp7) over (lessp7') to a version of (lessp7') with marked variables.

5.3.9 Conclusion on THM

Logicians reading on THM may ask themselves many questions such as: Why is merging of induction schemes — seen as a binary algebraic operation — not realized to satisfy the constraint of associativity, so that the result of merging become independent of the order of the operations? Why does merging not admit the subterm-property in the same way as subsumption of induction schemes does? Why do some of the injectivity requirements¹⁶³ of subsumption and mergeability lack a meaningful justification, and how can it be that they do not matter?

The answer is trivial, although it is easily overlooked: The part of the automation of induction we have discussed in this section on THM, belongs mostly to the field of heuristics and not in the field of logics. Therefore, the final judgment cannot come from logical and intellectual adequacy and comprehensibility — which are not much more applicable here than in the field of neural nets for instance — but must come from complete testing with a huge and growing corpus of example theorems. A modification of an operation, say merging of induction schemes, that may have some practical advantages for some examples or admit humans some insight or understanding, can be accepted only if it admits us to run, as efficiently as before, all the lemmas that could be automatically proved with the system before. All in all, logical and formal considerations may help us to find new heuristics, but they cannot play any rôle in their evaluation.¹⁶⁴

Moreover, it is remarkable that the well-founded relation that is expressed by the subsuming induction scheme is smaller than that expressed by the subsumed one, and the relation expressed by a merged scheme is typically smaller than those expressed by the original ones. This means that the newly generated induction schemes do not represent a more powerful induction ordering (say, in terms of Noetherian induction), but actually achieve an improvement w.r.t. the eager instantiation of the induction hypothesis (both for a direct proof and for generalization), and provide case conditions that further a successful generalization without further case analysis.

Since the end of the 1970s until today, THM has set the standard for explicit induction; moreover, THM and its successors NQTHM and ACL2 have given many researchers a hard time trying to demonstrate weaknesses of their explicit-induction heuristics, because examples carefully devised to fail with certain steps of the construction of induction schemes (or other stages of the waterfall) tend to end up with alternative proofs not imagined before.

Restricted to the mechanization of explicit induction, no significant progress has been seen beyond THM and we do not expect any for the future. A heuristic

¹⁶³Cf. Notes 158 and 159.

¹⁶⁴While Christoph Walther is well aware of the primacy of testing in [Walther, 1992; 1993], this awareness is not reflected in the sloppy language of the most interesting papers [Stevens, 1988] and [Bundy *et al.*, 1989]: A “rational reconstruction” or “meta-theoretic analysis” of heuristics does not make sense in our opinion, and heuristics cannot be “bugged” or “have serious flaws”, but can only turn out to be inferior to others w.r.t. a standard corpus.

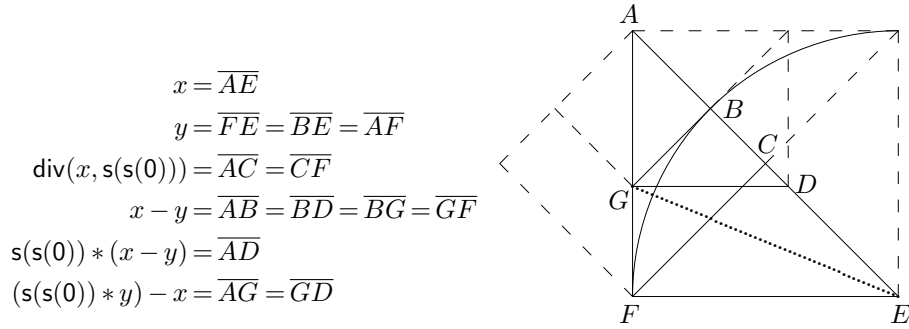


Figure 4. Four possibilities to descend with rational representations of $\sqrt{2}$:
From the triangle with right angle at F to those at C , G , or B .

approach that has to anticipate appropriate induction steps with a lookahead of one individual rewrite step for each recursive function occurring in the input formula cannot go much further than the carefully developed and exhaustively tested explicit-induction heuristics THM.

Working with THM (or NQTHM) for the first time will always fascinate informaticians and mathematicians, simply because it helps to save more time with the standard everyday inductive proof work than it takes, and the system often comes up with completely unexpected proofs. Mathematicians, however, should be warned that the less trivial mathematical proofs that require some creativity and would deserve to be explicated in a mathematics lecture, will require some hints, especially if the induction ordering is not a combination of the termination orderings of the given function definitions. This is already the case for the simple proofs of the lemma on the irrationality of the square root of two, simply because the induction orderings of the typical proofs exist only under the assumption that the lemma is wrong. To make THM find the standard proof, the user has to define a function such as the following one:

```

(sqrtio1)  sqrtio(x, y)
           = and(sqrtio(y, div(x, s(s(0))))),
             and(sqrtio(s(s(0)) * (x - y), (s(s(0)) * y) - x),
                sqrtio((s(s(0)) * y) - x, x - y))
           ⇐ x * x = s(s(0)) * y * y ∧ y ≠ 0

```

Note that the condition of (sqrtio1) cannot be fulfilled. The three different occurrences of sqrtio on the right-hand side of the positive/negative-conditional equation become immediately clear from Figure 4. Actually, any single one of these occurrences is sufficient for a proof of the irrationality lemma with THM, provided that we give the hint that the induction templates of sqrtio should be used for computing the induction schemes, in spite of the fact that sqrtio does not occur in the lemma.

5.4 NQTHM

Subsequent theorem provers by Boyer and Moore did not add much to the mechanization of induction. While both NQTHM and ACL2 have been very influential in theorem proving, their inductive heuristics are nearly the same as those in THM and their waterfalls have quite similar structures. Since we are concerned with the history of the mechanization of induction, we just sketch developments since 1979.

The one change from THM to NQTHM that most directly affected the inductions carried out by the system is the abandonment of fixed lexicographic relations on natural numbers as the only available well-founded relations. NQTHM introduces a formal representation of the ordinals up to ε_0 , i.e. up to $\omega^{\omega^{\dots}}$, and assumes that the “less than” relation on such ordinals is well-founded. This did not change the induction heuristics themselves, it just allowed the admission of more complex function definitions and the justification of more sophisticated induction templates.

After the publication of [Boyer and Moore, 1979] describing THM, Boyer and Moore turned to the question of providing limited support for higher-order functions in their first-order setting. This had two very practical motivations. One was to allow the user to extend the prover by defining and mechanically verifying new proof procedures in the pure LISP dialect supported by THM. The other was to allow the user the convenience of LISP’s “map functions” and LOOP facility. Both required formally defining the semantics of the logical language in the logic, i.e. axiomatizing the evaluation function EVAL. Ultimately this resulted in the provision of *metafunctions* [Boyer and Moore, 1981b] and the non-constructive “value-and-cost” function **V&C\$** [Boyer and Moore, 1988a], which were provided as part of the NQTHM system described in [Boyer and Moore, 1988b; 1998].

The most important side-effect of these additions, however, is under the hood; Boyer and Moore contrived to make the representation of constructor ground terms in the logic be identical to their representation as constants in its underlying implementation language LISP: Integers are represented directly as LISP integers; for instance, `s(s(s(0)))` is represented by the machine-oriented internal LISP representation of 3, instead of the previous `(ADD1 (ADD1 (ADD1 (ZERO))))`. Symbols and list structures are embedded this way as well, so that they can profit from the very efficient representation of these basic data types in LISP. It thus also became possible to represent symbolic machine states containing actual assembly code or the parse trees of actual programs in the logic of NQTHM. Metafunctions were put to good use canonicalizing symbolic state expressions. The exploration of formal operational semantics with NQTHM blossomed.

In addition, NQTHM adds a rational linear-arithmetic¹⁶⁵ decision procedure to the simplification stage of the waterfall [Boyer and Moore, 1988c], reducing the amount of user interaction necessary to prove arithmetic theorems. The incompleteness of the procedure when operating on terms beyond the linear fragment is of little practical importance since induction is available (and often automatic).

¹⁶⁵Linear arithmetic is traditionally called “Presburger Arithmetic” after Mojżesz Presburger (actually: “Prezburger”) (1904–1943?); cf. [Presburger, 1930], [Stansifer, 1984], [Zygmunt, 1991].

With NQTHM it became possible to formalize and verify problems beyond the scope of THM, such as the correctness of a netlist implementing the instruction-set architecture of a microprocessor [Hunt, 1985], Gödel's first incompleteness theorem,¹⁶⁶ the verified hard- & software stack of Computational Logic, Inc., relating a fabricated microprocessor design through an assembler, linker, loader, several compilers, and an operating system to simple verified application programs,¹⁶⁷ and the verification of the Berkeley C String Library.¹⁶⁸ Many more examples are listed in [Boyer and Moore, 1998].

5.5 ACL2

Because of the pervasive change in the representation of constants, the LISP subset supported by NQTHM is exponentially more efficient than the LISPs supported by THM and the PURE LISP THEOREM PROVER. It is still too inefficient, however: Emerging applications of NQTHM in the late 1980s included models of commercial microprocessors; users wished to run their models on industrial test suites. The root cause of the inefficiency was that ground execution in NQTHM was done by a purpose-built interpreter implemented by Boyer and Moore. To reach competitive speeds, it would have been necessary to build a good compiler and full runtime system for the LISP subset axiomatized in NQTHM. Instead, in August 1989, less than a year after the publication of [Boyer and Moore, 1988b] describing NQTHM, Boyer and Moore decided to axiomatize a practical subset of COMMON LISP [Steele, 1990], the then-emerging standard LISP, and to build an NQTHM-like theorem prover for it. To demonstrate that the subset was a practical programming language, they decided to code the theorem prover applicatively in that subset. Thus, ACL2 was born.

Boyer left Computational Logic, Inc., (CLI) and returned to his duties at the The University of Texas at Austin in 1989, while Moore resigned his tenure and stayed at CLI. This meant Moore was working full-time on ACL2, whereas Boyer was working on it only at night.

¹⁶⁶Cf. [Shankar, 1994]. In [Shankar, 1994, p. xii] we read on this work with NQTHM:

“This theorem prover is known for its powerful heuristics for constructing proofs by induction while making clever use of previously proved lemmas. The Boyer–Moore theorem prover did not discover proofs of the incompleteness theorem but merely checked a detailed but fairly high-level proof containing over 2000 definitions and lemmas leading to the main theorems. These definitions and lemmas were constructed through a process of interaction with the theorem prover which was able to automatically prove a large number of nontrivial lemmas. By thus proving a well-chosen sequence of lemmas, the theorem prover is actually used as a *proof checker* rather than a theorem prover.

If we exclude the time spent thinking, planning, and writing about the proof, the verification of the incompleteness theorem occupied about eighteen months of effort with the theorem prover.”

¹⁶⁷Cf. [Moore, 1989b; 1989a], [Bevier *et al.*, 1989], [Hunt, 1989], [Young, 1989], [Bevier, 1989].

¹⁶⁸Via verification of its gcc-generated Motorola MC68020 machine code [Boyer and Yu, 1996].

Matt Kaufmann (*1952), who had worked with Boyer and Moore since the mid-1980s on NQTHM and had joined them at CLI, was invited to join the ACL2 project.

By the mid-1990s, Boyer requested that his name be removed as an author of ACL2 because he no longer knew every line of code.

The only major change to inductive reasoning introduced by ACL2 was the further refinement of the induction templates computed at definition time. While NQTHM built the case analysis from the case conditions “governing” the recursive calls, ACL2 uses the more restrictive notion of the tests “ruling” the recursive calls. Compare the definition of *governors* on Page 180 of [Boyer and Moore, 1998] to the definition of *rulers* on Page 90 of [Kaufmann *et al.*, 2000b].

ACL2 represents a major step, however, toward Boyer and Moore’s dream of a *computational logic* because it is a theorem prover for a practical programming language. Because it is so used, *scaling* its algorithms and heuristics to deal with enormous models and the formulas they generate has been a major concern, as has been the efficiency of ground execution. Moreover, it also added many other proof techniques including congruence-based contextual rewriting, additional decision procedures, disjunctive search (meaning the waterfall no longer has just one pool but may generate several, one of which must be “emptied” to succeed), and many features made possible by the fact that the system code and state is visible to the logic and the user.

Among the landmark applications of ACL2 are the verification of a Motorola digital signal processor [Brock and Hunt, 1999] and of the floating-point division microcode for the AMD K5™ microprocessor [Moore *et al.*, 1998], the routine verification of all elementary floating point arithmetic on the AMD Athlon™ [Russinoff, 1998], the certification of the Rockwell Collins AAMP7G™ for multi-level secure applications by the US National Security Agency based on the ACL2 proofs [Anon, 2005], and the integration of ACL2 into the work-flow of Centaur Technology, Inc., a major manufacturer of X86 microprocessors [Hunt and Swords, 2009]. Some of this work was done several years before the publications appeared because the early use of formal methods was considered proprietary. For example, the work for [Brock and Hunt, 1999] was completed in 1994, and that for [Moore *et al.*, 1998] in 1995.

In most industrial applications of ACL2, induction is not used in every proof. Many of the proofs involve huge intermediate formulas, some requiring megabytes of storage simply to represent, let alone simplify. Almost all the proofs, however, depend on lemmas that require induction to prove.

To be successful, ACL2 must be good at both induction and simplification and *integrate* them seamlessly in a well-engineered system, so that the user can state and prove in a single system all the theorems needed.

ACL2 is most relevant to the historiography of inductive theorem proving because it demonstrates that the induction heuristics and the waterfall provide such an integration in ways that can be scaled to industrial-strength applications.

ACL2 and, by extension, inductive theorem proving, have changed the way microprocessors and low-level critical software are designed. Proof of correctness, or at least proof of some important system properties, is now a possibility.

Boyer, Moore, and Kaufmann were awarded the 2005 ACM Software Systems Award for “the Boyer–Moore Theorem Prover”:

“The Boyer–Moore Theorem Prover is a highly engineered and effective formal-methods tool that pioneered the automation of proofs by induction, and now provides fully automatic or human-guided verification of critical computing systems. The latest version of the system, ACL2, is the only simulation/verification system that provides a standard modeling language and industrial-strength model simulation in a unified framework. This technology is truly remarkable in that simulation is comparable to C in performance, but runs inside a theorem prover that verifies properties by mathematical proof. ACL2 is used in industry by AMD, IBM, and Rockwell-Collins, among others.”¹⁶⁹

5.6 *Explicit Induction in RRL, INKA, and OYSTER/CIAM*

RRL, the *Rewrite Rule Laboratory* [Kapur and Zhang, 1989], was initiated in 1982 and showed its main activity during its first dozen years. RRL is a system for proving the viability of many techniques related to term rewriting. Besides other forms of induction, RRL includes *cover-set induction*, which has eager induction-hypothesis generation, but is restricted to syntactic term orderings.

Interesting work on explicit induction was realized along the line of the INKA Induction (Karlsruhe) systems. We have to mention here Christoph Walther’s (*1950) elegant treatment of automated termination proofs for recursive function definitions [Walther, 1988; 1994b], and his theoretically outstanding work on the generation of step cases with eager induction-hypothesis generation [Walther, 1992; 1993]; moreover, there is Dieter Hutter’s (*1959) invention of rippling (in parallel to the same invention by Alan Bundy, cf. § 6.2), and Martin Protzen’s (*1962) profound work on patching of faulty conjectures and on breaking out of the imagined cage of explicit induction by “lazy induction” [Protzen, 1994; 1995; 1996].

The INKA project started in 1984 as a part of the Collaborative Research Center SFB 314 “Artificial Intelligence”, which was financed by the German Research Community (DFG) to overcome a backwardness in artificial intelligence in Germany of more than a dozen years compared to the research in Edinburgh and the US.

While the INKA systems proved the executability of several new concepts, they were never competitive with their contemporary Boyer–Moore theorem provers (though INKA 5.0 [Autexier *et al.*, 1999] was competitive in speed with NQTHM)¹⁷⁰ and the development of INKA was discontinued for this reason in the year 2000.

¹⁶⁹For the complete text of the citation of Boyer, Moore, and Kaufmann see <http://awards.acm.org/citation.cfm?id=4797627&aw=149>.

Three INKA system descriptions were presented at the CADE conference series: [Biundo *et al.*, 1986], [Hutter and Sengler, 1996], [Autexier *et al.*, 1999].

From the beginning, the INKA project, starting from [Boyer and Moore, 1979], ignored that Boyer and Moore had actually come from resolution theorem proving and abandoned this form of “random search” for their inductive theorem provers for very good reasons (cf. § 5.2). The problematic (many-sorted) resolution and paramodulation approach of the early INKA system was given up only close to the end of the project [Autexier *et al.*, 1999].

In principle, the INKA systems offered full first-order predicate logic, but typically via the poor operationalization of Skolemization as a standard preprocessing in resolution theorem proving. Besides interfaces to users and other systems, and the integration of logics, specifications, and results of other theorem provers, the essentially induction-relevant additions of INKA as compared to the system described in [Boyer and Moore, 1979] are the following: In [Biundo *et al.*, 1986] there is an existential quantification where the system tries to find witnesses for the existentially quantified variables by interactive program synthesis. In [Hutter and Sengler, 1996], there is rippling (cf. § 6.2).

The OYSTER/CIAM system was developed at the University of Edinburgh in the late 1980s¹⁷¹ and the 1990s by a large team led by Alan Bundy.¹⁷² OYSTER is a reimplementaion of NUPRL [Constable *et al.*, 1985], a proof editor for Martin-Löf constructive type theory with rules for structural induction in the style of Peano — a logic that is not well-suited for inductive proof search, as discussed in § 3.6. OYSTER is based on tactics with specifications in a meta-level language which provides a complete representation of the object level, but with a search space much better suited for inductive proof search. CIAM is a proof planner (cf. § 6.1) which guides OYSTER, based on proof search in the meta-language, which includes rippling (cf. § 6.2).

OYSTER/CIAM proved the viability of many concepts, but **it is the slowest system explicitly mentioned in this article,**¹⁷⁰ partly because of its constructive logic. Besides approaches in its line of development more or less addressing theorem proving in general, such as rippling (cf. § 6.2) and **the productive use of failure** [Ireland and Bundy, 1994], most interesting from the aspect of induction is the extension of recursion analysis to ripple analysis, which is sketched already in [Bundy *et al.*, 1989, § 7], and which is nicely presented in [Bundy, 1999, § 7.10].

¹⁷⁰This can roughly be concluded from the results of the inductive theorem proving contest at the 16th Int. Conf. on Automated Deduction (CADE), Trento, Italy, 1999 (the design of which is described in [Hutter and Bundy, 1999]), where the following systems competed with each other (in interaction with the following humans): NQTHM (Laurence Pierre), INKA 5.0 (Dieter Hutter), OYSTER/CIAM (Alan Bundy), and a first prototype of QUODLIBET (Ulrich Kühler). Only OYSTER/CIAM turned out to be significantly slower than the other systems, but all participating systems would have been left far behind ACL2 if it had participated.

¹⁷¹The system description [Bundy *et al.*, 1990] of OYSTER/CIAM appeared already in summer 1990 at the CADE conference series (with a submission in winter 1989/1990); so the development must have started before the 1990s, contrary to what is stated in § 11.4 of [Bundy, 1999].

¹⁷²For Alan Bundy see also Note 6.