# AUTOMATION OF MATHEMATICAL INDUCTION AS PART OF THE HISTORY OF LOGIC*

J Strother Moore, Claus-Peter Wirth

## 1 A SNAPSHOT OF A DECISIVE MOMENT IN HISTORY

The automation of mathematical theorem proving for deductive *first-order logic* started in the 1950s, and it took about half a century to develop systems that are sufficiently strong and general to be successfully applied outside the community of automated theorem proving.[1]

Surprisingly, the development of such strong systems for restricted logic languages was not achieved much earlier — for neither the *purely equational fragment* nor *propositional logic*.[2] Moreover, automation of theorem proving for *higher-order logic* has started becoming generally useful only during the last ten years.[3]

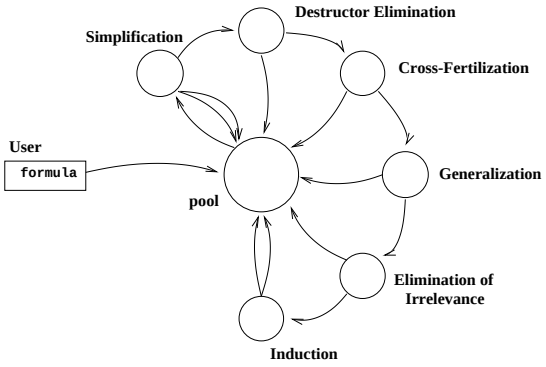


Figure 1. The Boyer–Moore Waterfall

Note that a formula falls back to the center pool after each successful application of one of the stages in the circle.

*Second Readers: Alan Bundy, Bernhard Gramlich

[1] The currently (i.e. in 2012) most successful first-order automated theorem prover is VAMPIRE, cf. e.g. [Riazanov and Voronkov, 2001].

[2] The most successful automated theorem prover for purely equational logic is WALDMEISTER, cf. e.g. [Buch and Hillenbrand, 1996], [Hillenbrand and Löchner, 2002]. For deciding propositional validity (i.e. sentential validity) (or its dual: propositional satisfiability) (which is decidable, but NP-complete), a breakthrough toward industrial strength was the SAT solver CHAFF, cf. e.g. [Moskewicz *et al.*, 2001].

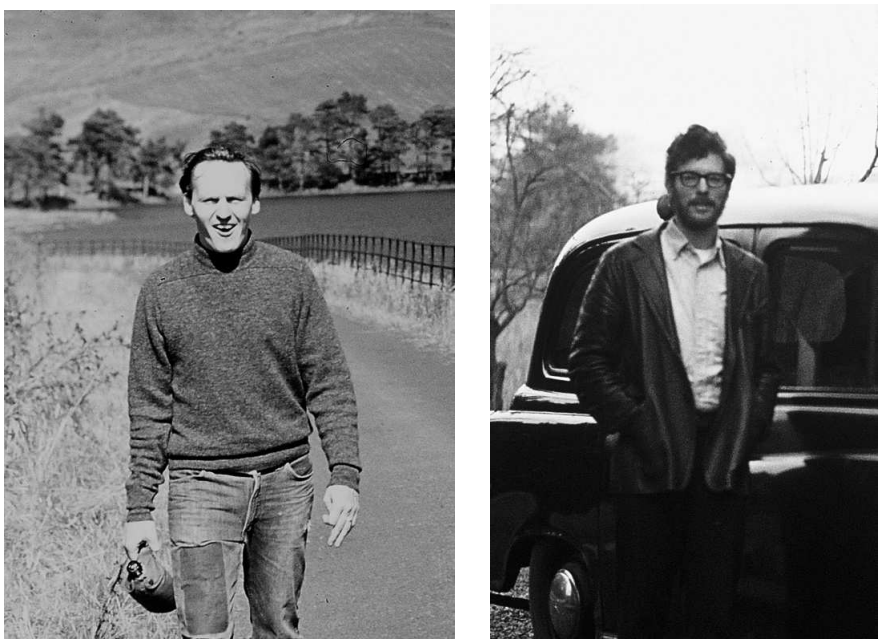

Figure 2. Robert S. Boyer (1971) (l.) and J Strother Moore (1972?) (r.)

In this context, it is surprising that for the field of quantifier-free first-order *inductive* theorem proving based on recursive functions, most of the progress toward general usefulness took place within the 1970s and that usefulness was clearly demonstrated by 1986.[4]

In this article we describe how this giant step took place, and sketch the further development of automated inductive theorem proving.

The work on this breakthrough in the automation of inductive theorem proving was started in September 1972, by Robert S. Boyer and J Strother Moore, in Edinburgh, Scotland. Unlike earlier work on theorem proving, Boyer and Moore chose to make induction the focus of their work. Most of the crucial steps and their synergetic combination in the "waterfall"[5] of their now famous theorem provers were developed in the span of a single year and implemented in their "Pure LISP

[3]Driving forces in the automation of higher-order theorem proving are the TPTP-competition-winning systems Leo-II (cf. e.g. [Benzmüller *et al.*, 2008]) and Satallax (cf. e.g. [Brown, 2012]).

[4]See the last paragraph of § 6.4.

[5]See Figure 1 for the Boyer–Moore waterfall. See [Bell and Thayer, 1976] for the probably first occurrence of "waterfall" as a term in software engineering. Boyer and Moore, however, were inspired not by this metaphor from software engineering, but again by a real waterfall, as can be clearly seen from [Boyer and Moore, 1979, p. 89]:

> "A good metaphor for the organization of these heuristics is an initially dry waterfall. One pours out a clause at the top. It trickles down and is split into pieces. Some pieces evaporate as they are proved. Others are further split up and simplified. Eventually at the bottom a pool of clauses forms whose conjunction suffices to prove the original formula."

Theorem Prover", presented at IJCAI in Stanford (CA) in August 1973,[6] and documented in Moore's PhD thesis [1973], defended in November 1973.

Readers who take a narrow view on the automation of inductive theorem proving might be surprised that we discuss the waterfall. It is impossible, however, to build a good inductive theorem prover without considering how to transform the induction conclusion into the hypothesis (or, alternatively, how to recognize that a legitimate induction hypothesis can dispatch a subgoal). So we take the expansive view and discuss not just the induction principle and its heuristic control, but also the waterfall architecture that is effectively an integral part of the success.

Boyer and Moore had met in August 1971, a year before the induction work started, when Boyer took up the position of a post-doctoral research fellow at the Metamathematics Unit of the University of Edinburgh. Moore was at that time starting the second year of his PhD studies in "the Unit". Ironically, they were both from Texas and they had both come to Edinburgh from the MIT. Boyer's PhD supervisor, W. W. Bledsoe, from The University of Texas at Austin, spent 1970–71 on sabbatical at the MIT, and Boyer accompanied him and completed his PhD work there. Moore got his bachelor's degree at the MIT (1966–70) before going to Edinburgh for his PhD.

Being "warm blooded Texans", they shared an office in the Metamathematics Unit at 9 Hope Park Square, Meadow Lane. The 18th century buildings at Hope Park Square were the center of Artificial Intelligence research in Britain at a time when the promises of AI were seemingly just on the horizon.[7] In addition to main-

[6] Cf. [Boyer and Moore, 1973].

[7] The Metamathematics Unit of the University of Edinburgh was renamed into "Dept. of Computational Logic" in late 1971, and was absorbed into the new "Dept. of Artificial Intelligence" in Oct. 1974. It was founded and headed by Bernard Meltzer. In the early 1970s, the University of Edinburgh hosted most remarkable scientists, of which the following are relevant in our context:

| | Univ. Edinburgh (time, Dept.) | PhD (year, advisor) | life time (birth–death) |
|---|---|---|---|
| Donald Michie | (1965–1984, MI) | (1953, unknown) | (1923–2007) |
| Bernard Meltzer | (1965–1978, CL) | (1953, Fürth) | (1916?–2008) |
| Robin J. Popplestone | (1965–1984, MI) | (no PhD) | (1938–2004) |
| Rod M. Burstall | (1965–2000, MI & Dept. AI) | (1966, Dudley) | (*1934) |
| Robert A. Kowalski | (1967–1974, CL) | (1970, Meltzer) | (*1941) |
| Pat Hayes | (1967–1973, CL) | (1973, Meltzer) | (*1944) |
| Gordon Plotkin | (1968–today, CL & LFCS) | (1972, Burstall) | (*1946) |
| J Strother Moore | (1970–1973, CL) | (1973, Burstall) | (*1947) |
| Mike J. C. Gordon | (1970–1978, MI) | (1973, Burstall) | (*1948) |
| Robert S. Boyer | (1971–1973, CL) | (1971, Bledsoe) | (*1946) |
| Alan Bundy | (1971–today, CL) | (1971, Goodstein) | (*1947) |
| Robin Milner | (1973–1979, LFCS) | (no PhD) | (1934–2010) |

CL = Metamathematics Unit (founded and headed by Bernard Meltzer)
(new name from late 1971 to Oct. 1974: Dept. of Computational Logic)
(new name from Oct. 1974: Dept. of Artificial Intelligence)

MI = Experimental Programming Unit (founded and headed by Donald Michie)
(new name from 1966 to Oct. 1974: Dept. for Machine Intelligence and Perception)
(new name from Oct. 1974: Machine Intelligence Unit)

LFCS = Laboratory for Foundations of Computer Science

(Sources: [Meltzer, 1975], [Kowalski, 1988], etc.)

line work on mechanized reasoning by Rod M. Burstall, Robert A. Kowalski, Pat Hayes, Gordon Plotkin, J Strother Moore, Mike J. C. Gordon, Robert S. Boyer, Alan Bundy, and (by 1973) Robin Milner, there was work on new programming paradigms, program transformation and synthesis, natural language, machine vision, robotics, and cognitive modeling. Hope Park Square received a steady stream of distinguished visitors from around the world, including J. Alan Robinson, John McCarthy, W. W. Bledsoe, Dana S. Scott, and Marvin Minsky. An eclectic series of seminars were on offer weekly to complement the daily tea times, where all researchers gathered around a table and talked about their current problems.

Boyer and Moore initially worked together on structure sharing in resolution theorem proving. The inventor of resolution, J. Alan Robinson (*1930?), created and awarded them the "1971 Programming Prize" on December 17, 1971 — half jokingly, half seriously. The document, handwritten by Robinson, actually says in part:

> "In 1971, the prize is awarded, by unanimous agreement of the Board, to Robert S. Boyer and J Strother Moore for their idea, explained in [Boyer and Moore, 1971], of representing clauses as their own genesis. The Board declared, on making the announcement of the award, that this idea is '... bloody marvelous'."

Their structure-sharing representation of derived clauses in a linear resolution system is just a stack of resolution steps. This suggests the idea of resolution being a kind of "procedure call."[8] Exploiting structure sharing, Boyer and Moore implemented a declarative LISP-like programming language called "Baroque" [Moore, 1973], a precursor to Prolog.[9] They then implemented a LISP interpreter in Baroque and began to use their resolution engine to prove simple theorems about programs in LISP. Resolution was sufficient to prove such theorems as "there is a list whose length is 3", whereas the absence of a rule for induction prevented the proofs of more interesting theorems like the associativity of list concatenation.

So, in the summer of 1972, they turned their attention to a theorem prover designed explicitly to do mathematical induction — this at a time when uniform first-order proof procedures were all the rage. The fall of 1972 found them taking turns at the blackboard proving theorems about recursive LISP functions and articulating their reasons for each proof step. Only after several months of such proofs did they sit down together to write the code for the Pure LISP Theorem Prover.

Today's readers might have difficulty imagining the computing infrastructure in Scotland in the early 1970s. Boyer and Moore developed their software on an ICL–4130, with 64 kByte (128 kByte in 1972) core memory (RAM). Paper tape was used for archival storage. The machine was physically located in the Forrest Hill building of the University of Edinburgh, about 1 km from Hope Park Square. A rudimentary time-sharing system allowed several users at once to run

[8] Cf. [Moore, 1973, p. 68].

[9] For logic programming and Prolog see [Kowalski, 1974; 1988], [Clocksin and Mellish, 2003].

lightweight applications from teletype machines at Hope Park Square. The only high-level programming language supported was POP–2, a simple stack-based list-processing language with an ALGOL-like syntax.[10]

Programs were prepared with a primitive text editor modeled on a paper tape editor: a disk file could be copied through a one byte buffer to an output file. By halting the copying and typing characters into or deleting characters from the buffer one could edit a file — a process that usually took several passes. Memory limitations of the ICL–4130 prohibited storing large files in memory for editing. In their very early collaboration, Boyer and Moore solved this problem by inventing what has come to be called the "piece table ", whereby an edited document is represented by a linked list of "pieces" referring to the original file which remains on disk. Their "77-editor" [Boyer *et al.*, 1973] (written in 1971 and named for the disk track on which it resided) provided an interface like MIT's Teco, but with POP–2 as the command language.[11] It was thus with their own editor that Boyer and Moore wrote the code for the PURE LISP THEOREM PROVER.

During the day they worked at Hope Park Square, with frequent trips by foot or bicycle through The Meadows to Forrest Hill to make archival paper tapes or to pick up line-printer output. During the night — when they could often have the ICL–4130 to themselves — they often worked at Boyer's home where another teletype was available.

## 2 METHOD OF PROCEDURE AND PRESENTATION

In contrast to the excellent handbook articles [Walther, 1994a] and [Bundy, 1999] on the *automation of explicit induction*, our focus in this article is neither on current standards, nor on the engineering and research problems of the field, but on the *history of the automation of mathematical induction*.

It is always hard to see the past because we look through the lens of the present. Achieving the necessary detachment from the present is especially hard for the historian of recent history because the "lens of the present" is shaped so immediately by the events being studied.

We try to mitigate this problem by avoiding the standpoint of a disciple of the leading school of explicit induction. Instead, we put the historic achievements into a broad mathematical context and a space of time from the ancient Greeks to a possible future, based on a most general approach to *recursive definition* (cf. § 5), and on *descente infinie* as a general, implementation-neutral approach to mathematical induction (cf. § 4.7). Then we can see the great achievements in the field with the surprise they historically deserve — after all, until 1973 mathematical induction was considered too creative an activity to be automated.

[10] Cf. [Burstall *et al.*, 1971].

[11] The 77-editor was widely used by researchers at Hope Park Square until the ICL–4130 was decommissioned. When Moore went to Xerox PARC in Palo Alto (CA) (Dec. 1973), the Boyer–Moore representation [Moore, 1981] was adopted by Charles Simonyi (*1948) for the Bravo editor on the Alto and subsequently found its way into Microsoft Word, cf. [Verma, 2005?].

As a historiographical text, this article should be accessible to an audience that goes beyond the technical experts and programmers of the day, should use common mathematical language and representation, focus on the global and eternal ideas and their developments, and paradigmatically display the *historically most significant achievements*.

Because these achievements in the automation of inductive theorem proving manifest themselves mainly in the line of the Boyer–Moore theorem provers, we cannot avoid the confrontation of the reader with some more ephemeral forms of representation found in these software systems. In particular, we cannot avoid some small expressions in the list programming language LISP,[12] simply because the Boyer–Moore theorem provers we discuss in this article, namely the Pure LISP Theorem Prover, Thm, Nqthm, and ACL2, all have *logics* based on a subset of LISP.

Note that we do not necessarily refer to the *implementation language* of these software systems, but to the *logic language* used both for representation of formulas and for communication with the user!

For the first system in this line of development, Boyer and Moore had a free choice, but wrote:

> "We use a subset of LISP as our language because recursive list processing functions are easy to write in LISP and because theorems can be naturally stated in LISP; furthermore, LISP has a simple syntax and is universal in Artificial Intelligence. We employ a LISP interpreter to 'run' our theorems and a heuristic which produces induction formulas from information about how the interpreter fails. We combine with the induction heuristic a set of simple rewrite rules of LISP and a heuristic for generalizing the theorem being proved."[13]

Note that the choice of LISP was influenced by the rôle of the LISP interpreter in induction. LISP was important for another reason: Boyer and Moore were building a *computational-logic* theorem prover:

> "The structure of the program is remarkably simple by artificial intelligence standards. This is primarily because the control structure is embedded in the syntax of the theorem. This means that the system does not contain two languages, the 'object language', LISP, and the 'meta-language', predicate calculus. They are identified. This mix of computation and deduction was largely inspired by the view that the two processes are actually identical. Bob Kowalski, Pat Hayes, and the nature of LISP deserve the credit for this unified view."[14]

This view was prevalent in the Metamathematics Unit by 1972. Indeed, "the Unit" was by then officially renamed the Department of Computational Logic.[7]

[12] Cf. [McCarthy *et al.*, 1965]. Note that we use the historically correct capitalized "LISP" for general reference, but not for more recent, special dialects such as Common Lisp.

[13] Cf. [Boyer and Moore, 1973, p. 486, left column].

[14] Cf. [Moore, 1973, p. 207f.].

In general, inductive theorem proving with recursively defined functions requires a logic in which

> a *method of symbolic evaluation* can be obtained from an interpretation procedure by generalizing the ground terms of computation to terms with free variables that are implicitly universally quantified.

So candidates to be considered today (besides a subset of LISP or of $\lambda$-calculus) are the typed functional programming languages ML and HASKELL,[15] which, however, were not available in 1972. LISP and ML are to be preferred to HASKELL as the logic of an inductive theorem prover because of their innermost evaluation strategy, which gives preference to the constructor terms that represent the constructor-based data types, which again establish the most interesting domains in hard- and software verification and the major elements of mathematical induction.

Yet another candidate today would be the rewrite systems of [Wirth and Gramlich, 1994a] and [Wirth, 1991; 2009] with *constructor variables*[16] and *positive/negative-conditional equations*, designed and developed for the specification, interpretation, and symbolic evaluation of recursive functions in the context of inductive theorem proving in the domain of constructor-based data types. Neither this tailor-made theory, nor even the general theory of rewrite systems in which its development is rooted,[17] were available in 1972. And still today, the applicative subset of COMMON LISP that provides the logic language for ACL2 (= $(\text{ACL})^2$ = A Computational Logic for Applicative COMMON LISP) is again to preferred to these positive/negative-conditional rewrite systems for reasons of efficiency: The applications of ACL2 in hardware verification and testing require a performance that is still at the very limits of today's computing technology. This challenging efficiency demand requires, among other aspects, that the logic of the theorem prover is so close to its own programming language that — after certain side conditions have been checked — the theorem prover can defer the interpretation of ground terms to the analogous interpretation in its own programming language.

For most of our illustrative examples in this article, however, we will use the higher flexibility and conceptual adequacy of positive/negative-conditional rewrite systems. They are so close to standard logic that we can dispense their semantics to the reader's intuition,[18] and they can immediately serve as an intuitively clear replacement of the *Boyer–Moore machines.*[19]

[15] Cf. [Hudlak *et al.*, 1999] for HASKELL, [Paulson, 1996] for ML, which started as the metalanguage for implementations of LCF (the *Logic of Computable Functions* with a single undefined element $\perp$, invented by Scott [1993]) with structural induction over $\perp$, `0`, and `s`, but without original contributions to the automation of induction, cf. [Milner, 1972, p. 8], [Gordon, 2000].

[16] See § 5.4 of this article.

[17] See [Dershowitz and Jouannaud, 1990] for the theory in which the rewrite systems of [Wirth and Gramlich, 1994a], [Wirth, 1991; 2009] are rooted. One may try to argue that the paper that launched the whole field of rewrite systems, [Knuth and Bendix, 1970], was already out in 1972, but the relevant parts of rewrite theory for unconditional equations were developed only in the late 1970s and the 1980s. Especially relevant in the given context are [Huet, 1980] and [Toyama, 1988]. The rewrite theory of *positive/negative-conditional* equations, however, started to become an intensive area of research only with the burst of creativity at 1st Int. Workshop on Conditional Term Rewriting Systems (CTRS), Orsay (France), 1987; cf. [Kaplan and Jouannaud, 1988].

Moreover, the typed (many-sorted) approach of the positive/negative-conditional equations allows the presentation of formulas in a form that is much easier to grasp for human readers than the corresponding sugar-free LISP notation with its overhead of explicit type restrictions.

Another reason for avoiding LISP notation is that we want to make it most obvious that the achievements of the Boyer–Moore theorem provers are not limited to their LISP logic.

For the same reason, we also prefer examples from arithmetic to examples from list theory, which might be considered to be especially supported by the LISP logic. The reader can find the famous examples from list theory in almost any other publication on the subject.[20]

In general, we tend to present the challenges and their historical solutions with the help of small intuitive examples and refer the readers interested in the very details of the implementations of the theorem provers to the published and easily accessible documents on which our description is mostly based.

Nevertheless, small LISP expression cannot be completely avoided because we have to describe the crucial parts of the historically most significant implementations and ought to show some of the advantages of LISP's untypedness.[21] The readers, however, do not have to know more about LISP than the following: A LISP term is either a variable symbol, or a function call of the form $(f\ t_1\ \cdots\ t_n)$, where $f$ is a function symbol, $t_1, \ldots, t_n$ are LISP terms, and $n$ is one of the natural numbers, which we assume to include 0.

## 3 ORGANIZATION OF THIS ARTICLE

This article is further organized as follows.

§§ 4 and 5 offer a self-contained reference for the readers who are not familiar with the field of mathematical induction and its automation. In § 4 we introduce the essentials of mathematical induction. In § 5 we have to become more formal regarding recursive function definitions, their consistency, termination, and induction templates and schemes. The main part is § 6, where we present the historically most important systems in automated induction, and discuss the details of software systems for explicit induction, with a focus on the 1970s. After describing the application context in § 6.1, we describe the following Boyer–Moore theorem provers: the Pure LISP Theorem Prover (§ 6.2), Thm (§ 6.3), Nqthm (§ 6.4), and ACL2 (§ 6.5). The most noteworthy remaining explicit-induction systems are sketched in § 6.6. Alternative approaches to the automation of induction that do not follow the paradigm of explicit induction are discussed in § 7. After summarizing the lessons learned in § 8, we conclude with § 9.

[18] The readers interested into the precise details are referred to [Wirth, 2009].

[19] Cf. [Boyer and Moore, 1979, p. 165f.].

[20] Cf. e.g. [Moore, 1973], [Boyer and Moore, 1979; 1988b; 1998], [Walther, 1994a], [Bundy, 1999], [Kaufmann *et al.*, 2000a; 2000b].

[21] See the advantages of the untyped, type-restriction-free declaration of the shell `CONS` in § 6.3.

## 4 MATHEMATICAL INDUCTION

In this section, we introduce mathematical induction and clarify the difference between *descente infinie* and *Noetherian*, *structural*, and *explicit induction*.

According to Aristotle, *induction* means to go from the special to the general, and to realize the *general* from the memorized perception of particular cases. Induction plays a major rôle in the generation of conjectures in mathematics and the natural sciences. Modern scientists design experiments to falsify a conjectured law of nature, and they accept the law as a scientific fact only after many trials have all failed to falsify it. In the tradition of Euclid, mathematicians accept a mathematical conjecture as a theorem only after a rigorous proof has been provided. According to Kant, induction is *synthetic* in the sense that it properly extends what we think to know — in opposition to *deduction*, which is *analytic* in the sense that it cannot provide us with any information not implicitly contained in the initial judgments, though we can hardly be aware of all deducible consequences.

Surprisingly, in this well-established and time-honored terminology, *mathematical induction* is not induction, but a special form of deduction for which — in the 19th century — the term "induction" was introduced and became standard in German and English mathematics.[22] In spite of this misnomer, for the sake of brevity, the term "induction" will always refer to mathematical induction in what follows.

Although it received its current name only in the 19th century, mathematical induction has been a standard method of every working mathematician at all times. It has been conjectured[23] that Hippasus of Metapontum (ca. 550 B.C.) applied a form of mathematical induction, later named *descente infinie (ou indéfinie)* by Fermat. We find another form of induction, nowadays called *structural induction*, in a text of Plato (427–347 B.C.).[24] In Euclid's famous "Elements" [ca. 300 B.C.], we find several applications of *descente infinie* and in a way also of structural induction.[25] Structural induction was known to the Muslim mathematicians around the year 1000, and occurs in a Hebrew book of Levi ben Gerson (Orange and Avignon) (1288–1344).[26] Furthermore, structural induction was used by Francesco Maurolico (Messina) (1494–1575),[27] and by Blaise Pascal (1623–1662).[28] After an absence of more than one millennium (besides copying ancient proofs), *descente infinie* was reinvented by Pierre Fermat (160?–1665).[29]

[22] First in German (cf. Note 38), soon later in English (cf. [Cajori, 1918]).

[23] It is conjectured in [Fritz, 1945] that Hippasus has proved that there is no pair of natural numbers that can describe the ratio of the lengths of the sides of a pentagram and its enclosing pentagon. Note that this ratio, seen as an irrational number, is equal to the golden number, which, however, was conceptualized in entirely different terms in ancient Greek mathematics.

[24] Cf. [Acerbi, 2000].

[25] An example for *descente infinie* is Proposition 31 of Vol. VII of the Elements. Moreover, the proof in the Elements of Proposition 8 of Vol. IX seems to be sound according to mathematical standards; and so we can see it only as a proof by structural induction in a very poor linguistic and logical form. This is in accordance with [Freudenthal, 1953], but not with [Unguru, 1991] and [Acerbi, 2000]. See [Fowler, 1994] and [Wirth, 2010b, § 2.4] for further discussion.

[26] Cf. [Rabinovitch, 1970]. Also summarized in [Katz, 1998].

[27] Cf. [Bussey, 1917].

[28] Cf. [Pascal, 1954, p. 103].

### *4.1 Well-Foundedness and Termination*

A relation $<$ is *well-founded* if, for each proposition $Q(w)$ that is not constantly false, there is a $<$-minimal $m$ among the objects for which $Q$ holds, i.e. there is an $m$ with $Q(m)$, for which there is no $u < m$ with $Q(u)$. Writing "$\mathsf{Wellf}(<)$" for "$<$ is well-founded", we can formalize this *definition* as follows:

$$(\mathsf{Wellf}(<)) \qquad \forall Q.\ \Big(\ \exists w.\ Q(w) \quad \Rightarrow \quad \exists m.\ \big(Q(m) \wedge \neg\exists u{<}m.\ Q(u)\big)\ \Big)$$

Let $<^+$ denote the transitive closure of $<$, and $<^*$ the reflexive closure of $<^+$.

$<$ is an (irreflexive) *ordering* if it is an irreflexive and transitive relation. There is not much difference between a well-founded relation and a well-founded ordering:[30]

LEMMA 1. *$<$ is well-founded if and only if $<^+$ is a well-founded ordering.*

Closely related to the well-foundedness of a relation $<$ is the termination of its *reverse relation* $>$, given as $<^{-1} := \{\ (u,v)\ |\ (v,u) \in <\ \}$.

A relation $>$ is *terminating* if it has no non-terminating sequences, i.e. if there is no infinite sequence of the form $x_0 > x_1 > x_2 > x_3 \ldots$.

If $>$ has a non-terminating sequence, then this sequence, taken as a set, is a witness for the non-well-foundedness of $<$. The converse implication, however, is a weak form of the Axiom of Choice;[31] indeed, it allows us to pick a non-terminating sequence for $>$ from the set witnessing the non-well-foundedness of $<$.

So well-foundedness is slightly stronger than termination of the reverse relation, and the difference is relevant here because we cannot take the Axiom of Choice for granted in a discussion of the foundations of induction, as will be explained in § 4.3.

### *4.2 The Theorem of Noetherian Induction*

In its modern standard meaning, the method of mathematical induction is easily seen to be a form of deduction, simply because it can be formalized as the application of the *Theorem of Noetherian Induction*:

> A proposition $P(w)$ can be shown to hold (for all $w$) by *Noetherian induction* over a well-founded relation $<$ as follows: *Show (for every $v$) that $P(v)$ follows from the assumption that $P(u)$ holds for all $u < v$.*

Again writing "$\mathsf{Wellf}(<)$" for "$<$ is well-founded", we can formalize the *Theorem of Noetherian Induction* as follows:[32]

$$(\mathsf{N}) \qquad \forall P.\ \Bigg(\ \forall w.\ P(w) \quad \Leftarrow \quad \exists <.\ \Bigg(\ \begin{array}{ll} & \forall v.\big(\,P(v) \ \Leftarrow\ \forall u{<}v.\ P(u)\,\big) \\ \wedge & \mathsf{Wellf}(<) \end{array}\ \Bigg)\Bigg)$$

[29] There is no consensus on Fermat's year of birth. Candidates are 1601, 1607 ([Barner, 2007]), and 1608. Thus, we write "160?", following [Goldstein, 2008]. The best-documented example of Fermat's applications of *descente infinie* is the proof of the theorem: *The area of a rectangular triangle with positive integer side lengths is not the square of an integer*; cf. e.g. [Wirth, 2010b].

[30] Cf. Lemma 2.1 of [Wirth, 2004, § 2.1.1].

[31] See [Wirth, 2004, § 2.1.2, p. 18] for the equivalence to the Principle of Dependent Choice, found in [Rubin and Rubin, 1985, p.19], analyzed in [Howard and Rubin, 1998, p. 30, Form 43].

[32] When we write an implication $A \Rightarrow B$ in the reverse form of $B \Leftarrow A$, we do this to indicate that a proof attempt will typically start from $B$ and try to reduce it to $A$.

The today commonly used term "Noetherian induction" is a tribute to the famous female German mathematician Emmy Noether (1882–1935). It occurs as the "Generalized principle of induction (Noetherian induction)" in [Cohn, 1965, p. 20]. Moreover, it occurs as Proposition 7 ("Principle of Noetherian Induction") in [Bourbaki, 1968a, Chapter III, § 6.5, p. 190] — a translation of the French original in its second edition [Bourbaki, 1967, § 6.5], where it occurs as Proposition 7 ("principe de récurrence nœthérienne").[33] We do not know whether "Noetherian" was used as a name of an induction principle before 1965;[34] in particular, it does not occur in the first French edition [Bourbaki, 1956] of [Bourbaki, 1967].[35]

## *4.3 An Induction Principle Stronger than Noetherian Induction?*

Let us try to find a weaker replacement for the precondition of well-foundedness in Noetherian induction, in the sense that we try to replace "$\mathsf{Wellf}(<)$" in the Theorem of Noetherian Induction ($\mathsf{N}$) in § 4.2 with some weaker property, which we will designate with "$\mathsf{Weak}(<, P)$" (such that $\forall P.\ \mathsf{Weak}(<, P) \Leftarrow \mathsf{Wellf}(<)$). This would result in the formula

$$(\mathsf{N}') \qquad \forall P.\left(\ \forall w.\ P(w) \quad \Leftarrow \quad \exists <.\left(\begin{array}{ll} & \forall v.\big(\,P(v) \ \Leftarrow\ \forall u{<}v.\ P(u)\,\big) \\ \wedge & \mathsf{Weak}(<, P) \end{array}\right)\right).$$

If we assume ($\mathsf{N}'$), however, we get the converse $\forall P.\ \mathsf{Weak}(<, P) \ \Rightarrow\ \mathsf{Wellf}(<)$.[36] This means that a proper weakening is possible only w.r.t. *certain* $P$, and the Theorem of Noetherian Induction *is the strongest among those induction principles of the form* ($\mathsf{N}'$) *where* $\mathsf{Weak}(<, P)$ *does not depend on* $P$.

$C$ is a *$<$-chain* if $<^+$ is a total ordering on $C$. Let us write "$u{<}C$" for $\forall c \in C.\ u{<}c$, and "$\forall u{<}C.\ F$" as usual for $\forall u.(u{<}C \Rightarrow F)$. In [Geser, 1995], we find applications of an induction principle that roughly has the form ($\mathsf{N}'$) where $\mathsf{Weak}(<, P)$ is:

> For every non-empty $<$-chain $C$ [without a $<$-minimal element]:
> $$\exists v \in C.\ P(v) \quad \Leftarrow \quad \forall u{<}C.\ P(u).$$

The resulting induction principle can be given an elegant form: If we drop the part of $\mathsf{Weak}(<, P)$ given in optional brackets [...], then we can drop the conjunction in ($\mathsf{N}'$) together with its first element, because $\{v\}$ is a non-empty $<$-chain.

---

[33] The peculiar French spelling "nœthérienne" imitates the German pronunciation of "Noether", where the "oe" is to be pronounced neither as a long "o" (the default, as in "Itzehoe"), nor as two separate vowels as indicated by the diaeresis in "oë", but as an umlaut, typically written in German as the ligature "ö". Neither Emmy nor her father Max Noether (1844–1921) (mathematics professor as well) used this ligature, found however in some of their official German documents.

[34] In 1967, "Noetherian Induction" was not generally used as a name for the Theorem of Noetherian Induction yet: For instance, this theorem — instantiated with the ordering of the natural numbers — is called the *principle of complete induction* in [Schoenfield, 1967, p. 205], but more often called *course-of-values induction*, cf. e.g. `http://en.wikipedia.org/wiki/Mathematical_induction#Complete_induction`. "Complete induction", however, is a most confusing name hardly used in English. Indeed, "complete induction" is the literal translation of the German technical term "vollständige Induction", which traditionally means structural induction (cf. Note 38) — and these two kinds of mathematical induction are different from each other.

[35] Indeed, the main text of § 6.5 in the 1st edition [Bourbaki, 1956] ends (on Page 98) three lines before the text of Proposition 7 begins in the 2nd edition [Bourbaki, 1967] (on Page 76 of § 6.5).

Then the following equivalent is obtained by switching from proposition $P$ to its class of counterexamples $Q$: "If, for every non-empty $<$-chain $C \subseteq Q$, there is a $u \in Q$ with $u<C$, then $Q = \emptyset$." Under the assumption that $Q$ is a set, this is an equivalent of the Axiom of Choice (cf. [Geser, 1995], [Rubin and Rubin, 1985]).

This means that the axiomatic status of induction principles ranges from the Theorem of Noetherian Induction up to the Axiom of Choice. If we took the Axiom of Choice for granted, this difference in status between a theorem and an axiom would collapse and our discussion of the axiomatic status of mathematical induction would degenerate. So the care with which we distinguished termination of the reverse relation from well-foundedness in § 4.1 is justified.

## *4.4 The Natural Numbers*

The field of application of mathematical induction most familiar in mathematics is the domain of the natural numbers 0, 1, 2, .... Let us formalize the natural numbers with the help of two constructor function symbols, namely one for the constant zero and one for the direct successor of a natural number:

$$\mathsf{0} : \mathsf{nat}$$
$$\mathsf{s} : \mathsf{nat} \to \mathsf{nat}$$

Moreover, let us assume in this article that the variables $x$, $y$ always range over the natural numbers, and that free variables in formulas are implicitly universally quantified (as is standard in mathematics), such that, for example, a formula with the free variable $x$ can be seen as having the implicit outermost quantifier $\forall x : \mathsf{nat}$.

After the definition ($\mathsf{Wellf}(<)$) and the theorem ($\mathsf{N}$), let us now consider some standard *axioms* for specifying the natural numbers, namely that a natural number is either zero or a direct successor of another natural number ($\mathsf{nat1}$), that zero is not a successor ($\mathsf{nat2}$), that the successor function is injective ($\mathsf{nat3}$), and that the so-called *Axiom of Structural Induction over* $\mathsf{0}$ *and* $\mathsf{s}$ holds; formally:

($\mathsf{nat1}$) $\quad x = \mathsf{0} \quad \vee \quad \exists y.\ \big(\ x = \mathsf{s}(y)\ \big)$

($\mathsf{nat2}$) $\quad \mathsf{s}(x) \neq \mathsf{0}$

($\mathsf{nat3}$) $\quad \mathsf{s}(x) = \mathsf{s}(y) \quad \Rightarrow \quad x = y$

($\mathsf{S}$) $\quad \forall P.\ \Big(\ \forall x.\ P(x) \quad \Leftarrow \quad P(\mathsf{0}) \ \wedge\ \forall y.\ \big(\ P(\mathsf{s}(y)) \Leftarrow P(y)\ \big)\ \Big)$

[36] *Proof.* Let $<\restriction_A$ denote the range restriction of $<$ to $A$ (i.e. $u<\restriction_A v$ if and only if $u < v \in A$). Let us take $P(w)$ to be $\mathsf{Wellf}(<\restriction_{A(w)})$ for $A(w) := \{\ w' \mid w' <^* w\ \}$. Then the reverse implication follows from ($\mathsf{N}'$) because $P(v) \Leftarrow \forall u<v.\ P(u)$ holds for any $v$,[37] and $\forall w.\ P(w)$ implies $\mathsf{Wellf}(<)$.

[37] *Proof.* To show $P(v)$, it suffices to find, for an arbitrary, not constantly false proposition $Q$, an $m$ with $Q(m)$, for which, in case of $m \in A(v)$, there is no $m' < m$ with $Q(m')$.

If we have $Q(m)$ for some $m$ with $m \notin A(v)$, then we are done.

If we have $Q(u')$ for some $u < v$ and some $u' \in A(u)$, then, for $Q'(u'')$ being the conjunction of $Q(u'')$ and $u'' \in A(u)$, there is (because of the assumed $P(u)$) an $m$ with $Q'(m)$, for which there is no $m' < m$ with $Q'(m')$. Then we have $Q(m)$. If there were an $m' < m$ with $Q(m')$, then we would have $Q'(m')$. Thus, there cannot be such an $m'$, and so $m$ satisfies our requirements.

Otherwise, if none of these two cases is given, $Q$ can only hold for $v$. As $Q$ is not constantly false, we get $Q(v)$ and then $v \not< v$ (because otherwise the second case is given for $u := v$ and $u' := v$). Then $m := v$ satisfies our requirements.

Richard Dedekind (1831–1916) proved the Axiom of Structural Induction (S) for his model of the natural numbers in [Dedekind, 1888], where he states that the proof method resulting from the application of this axiom is known under the name "vollständige Induction".[38]

Now we can go on by defining — in two equivalent[39] ways — the destructor function $\mathsf{p} : \mathsf{nat} \to \mathsf{nat}$, returning the predecessor of a positive natural number:

$$(\mathsf{p1}) \qquad \mathsf{p}(\mathsf{s}(x)) = x$$

$$(\mathsf{p1}') \qquad \mathsf{p}(x') = x \;\Leftarrow\; x' = \mathsf{s}(x)$$

The definition via (p1) is in *constructor style*, where constructor terms may occur on the left-hand side of the positive/negative-conditional equation as arguments of the function being defined. The alternative definition via (p1′) is in *destructor style*, where only variables may occur as arguments on the left-hand side.

For both definition styles, the term on the left-hand side must be *linear* (i.e. all its variable occurrences must be distinct variables) and have the function symbol to be defined as the top symbol.

Let us define some recursive functions over the natural numbers, such as addition and multiplication $+, * : \mathsf{nat}, \mathsf{nat} \to \mathsf{nat}$, the irreflexive ordering of the natural numbers $\mathsf{lessp} : \mathsf{nat}, \mathsf{nat} \to \mathsf{bool}$ (see § 4.5 for the data type bool of Boolean values), and the Ackermann function $\mathsf{ack} : \mathsf{nat}, \mathsf{nat} \to \mathsf{nat}$:[40]

$$\begin{array}{ll|ll} (+1) & 0+y = y & (*1) & 0*y = 0 \\ (+2) & \mathsf{s}(x)+y = \mathsf{s}(x+y) & (*2) & \mathsf{s}(x)*y = y+(x*y) \end{array}$$

$$\begin{array}{ll} (\mathsf{lessp1}) & \mathsf{lessp}(x,0) = \mathsf{false} \\ (\mathsf{lessp2}) & \mathsf{lessp}(0,\mathsf{s}(y)) = \mathsf{true} \\ (\mathsf{lessp3}) & \mathsf{lessp}(\mathsf{s}(x),\mathsf{s}(y)) = \mathsf{lessp}(x,y) \end{array}$$

$$\begin{array}{ll} (\mathsf{ack1}) & \mathsf{ack}(0,y) = \mathsf{s}(y) \\ (\mathsf{ack2}) & \mathsf{ack}(\mathsf{s}(x),0) = \mathsf{ack}(x,\mathsf{s}(0)) \\ (\mathsf{ack3}) & \mathsf{ack}(\mathsf{s}(x),\mathsf{s}(y)) = \mathsf{ack}(x,\mathsf{ack}(\mathsf{s}(x),y)) \end{array}$$

[38] In the tradition of Aristotelian logic, the technical term "vollständige Induction" (in Latin: "inductio completa", cf. e.g. [Wolff, 1740, Part I, § 478, p. 369]) denotes a complete case analysis, cf. e.g. [Lambert, 1764, Dianoiologie, § 287; Alethiologie, § 190]. Its misuse as a designation of structural induction originates in [Fries, 1822, p. 46f.], and was perpetuated by Dedekind. Its literal translation "complete induction" is misleading, cf. Note 34. By the 1920s, "vollständige Induction" had become a very vague notion that is best translated as "mathematical induction", as done in [Heijenoort, 1971, p.130] and as it is standard today, cf. e.g. [Hilbert and Bernays, 2013, Note 23.4].

[39] For the equivalence transformation between constructor and destructor style see Example 15 in § 6.3.2.

[40] Rósza Péter (1905–1977) (a woman in the fertile community of Budapest mathematicians and, like most of them, of Jewish parentage) published a simplified version [1951] of the first recursive, but not primitive recursive function developed by Wilhelm Ackermann (1896–1962) [Ackermann, 1928]. It is actually Péter's version what is simply called "the Ackermann function" today.

The relation from a natural number to its direct successor can be formalized by the binary relation $\lambda x, y.\ (\mathsf{s}(x) = y)$. Then $\mathsf{Wellf}(\lambda x, y.\ (\mathsf{s}(x) = y))$ states the well-foundedness of this relation, which means according to Lemma 1 that its transitive closure — i.e. the irreflexive ordering of the natural numbers — is a well-founded ordering; so, in particular, we have $\mathsf{Wellf}(\lambda x, y.\ (\mathsf{lessp}(x, y) = \mathsf{true}))$.

Now the natural numbers can be specified up to isomorphism either by[41]

- (nat2), (nat3), and (S) — following Guiseppe Peano (1858–1932),

or else by

- (nat1) and $\mathsf{Wellf}(\lambda x, y.\ (\mathsf{s}(x) = y))$ — following Mario Pieri (1860–1913).[42]

Immediate consequences of the axiom (nat1) and the definition (p1) are the lemma (s1) and its flattened[43] version (s1′):

(s1) $\quad \mathsf{s}(\mathsf{p}(x')) = x' \quad \Leftarrow \quad x' \neq \mathsf{0}$

(s1′) $\quad \mathsf{s}(x) = x' \quad \Leftarrow \quad x' \neq \mathsf{0} \ \wedge \ x = \mathsf{p}(x')$

Moreover, on the basis of the given axioms we can most easily show

(lessp4) $\quad \mathsf{lessp}(x, \mathsf{s}(x)) \quad = \mathsf{true}$

(lessp5) $\quad \mathsf{lessp}(x, \mathsf{s}(x + y)) = \mathsf{true}$

by *structural induction on* $x$, i.e. by taking the predicate variable $P$ in the Axiom of Structural Induction (S) to be $\lambda x.\ (\mathsf{lessp}(x, \mathsf{s}(x)) = \mathsf{true})$ in case of (lessp4), and $\lambda x.\ \forall y.\ (\mathsf{lessp}(x, \mathsf{s}(x + y)) = \mathsf{true})$ in case of (lessp5).

Moreover — to see the necessity of doing induction on several variables in parallel — we will present[44] the more complicated proof of the strengthened transitivity of the irreflexive ordering of the natural numbers, i.e. of

(lessp7) $\quad \mathsf{lessp}(\mathsf{s}(x), z) = \mathsf{true} \quad \Leftarrow \quad \mathsf{lessp}(x, y) = \mathsf{true} \ \wedge \ \mathsf{lessp}(y, z) = \mathsf{true}$

We will also prove the commutativity lemma (+3)[45] and the simple lemma (ack4) about the Ackermann function:[46]

(+3) $\quad x + y = y + x,$

(ack4) $\quad \mathsf{lessp}(y, \mathsf{ack}(x, y)) = \mathsf{true}$

[41] Cf. [Wirth, 2004, § 1.1.2].

[42] Pieri [1908] stated these axioms informally and showed their equivalence to the version of the Peano axioms [Peano, 1889] given in [Padoa, 1913]. For a discussion and an English translation see [Marchisotto and Smith, 2007]. Pieri [1908] has also a version where, instead of the symbol 0, there is only the statement that there is a natural number, and where (nat1) is replaced with the weaker statement that there is at most one s-minimal element:

$$\neg\exists y_0.\ (x_0 = \mathsf{s}(y_0)) \wedge \neg\exists y_1.\ (x_1 = \mathsf{s}(y_1)) \ \Rightarrow \ x_0 = x_1.$$

That non-standard natural numbers cannot exist in Pieri's specification is easily shown as follows: For every natural number $x$ we can form the set of all elements that can be reached from $x$ by the reverse of the successor relation; by well-foundedness of s, this set contains the unique s-minimal element (0); thus, we have $x = \mathsf{s}^n(\mathsf{0})$ for some standard meta-level natural number $n$.

[43] *Flattening* is a logical equivalence transformation that replaces a subterm (here: $\mathsf{p}(x')$) with a fresh variable (here: $x$) and adds a condition that equates the variable with the subterm.

[44] We will prove (lessp7) twice: once in Example 3 in § 4.7, and again in Example 12 in § 6.2.6.

[45] We will prove (+3) twice: once in Example 2 in § 4.7, and again in Example 4 in § 4.8.1.

[46] We will prove (ack4) in Example 5 in § 4.9.

### *4.5 Standard Data Types*

As we are interested in the verification of hardware and software, more important for us than natural numbers are the standard data types of higher-level programming languages, such as lists, arrays, and records.

To clarify the inductive character of data types defined by constructors, and to show the additional complications arising from constructors with no or more than one argument, let us present the data types $\mathsf{bool}$ (of Boolean values) and $\mathsf{list}(\mathsf{nat})$ (of lists over natural numbers), which we also need for our further examples.

A special case is the data type $\mathsf{bool}$ of the Boolean values given by the two constructors $\mathsf{true}, \mathsf{false} : \mathsf{bool}$ without any arguments, for which we get only the following two axioms by analogy to the axioms for the natural numbers. We globally declare the variable $b : \mathsf{bool}$; so $b$ will always range over the Boolean values.

($\mathsf{bool}1$) $\quad b = \mathsf{true} \quad \vee \quad b = \mathsf{false}$

($\mathsf{bool}2$) $\quad \mathsf{true} \neq \mathsf{false}$

Note that the analogy of the axioms of Boolean values to the axioms of the natural numbers (cf. § 4.4) is not perfect: An axiom ($\mathsf{bool}3$) analogous to ($\mathsf{nat}3$) cannot exist because there are no constructors for $\mathsf{bool}$ that take arguments. Moreover, an axiom analogous to ($\mathsf{S}$) is superfluous because it is implied by ($\mathsf{bool}1$).

Furthermore, let us define the Boolean function $\mathsf{and} : \mathsf{bool}, \mathsf{bool} \to \mathsf{bool}$ :

($\mathsf{and}1$) $\quad \mathsf{and}(\mathsf{false}, b) \;\; = \mathsf{false}$

($\mathsf{and}2$) $\quad \mathsf{and}(b, \mathsf{false}) \;\; = \mathsf{false}$

($\mathsf{and}3$) $\quad \mathsf{and}(\mathsf{true}, \mathsf{true}) = \mathsf{true}$

Let us now formalize the data type of the (finite) lists over natural numbers with the help of the following two constructors: the constant symbol

$$\mathsf{nil} : \mathsf{list}(\mathsf{nat})$$

for the empty list, and the function symbol

$$\mathsf{cons} : \mathsf{nat}, \;\; \mathsf{list}(\mathsf{nat}) \to \mathsf{list}(\mathsf{nat}),$$

which takes a natural number and a list of natural numbers, and returns the list where the number has been added to the input list as a new first element. We globally declare the variables $k, l : \mathsf{list}(\mathsf{nat})$.

By analogy to natural numbers, the axioms of this data type are the following:

($\mathsf{list}(\mathsf{nat})1$) $\quad l = \mathsf{nil} \;\vee\; \exists y, k.\; \big(\; l = \mathsf{cons}(y, k) \;\big)$

($\mathsf{list}(\mathsf{nat})2$) $\quad \mathsf{cons}(x, l) \neq \mathsf{nil}$

($\mathsf{list}(\mathsf{nat})3_1$) $\quad \mathsf{cons}(x, l) = \mathsf{cons}(y, k) \;\Rightarrow\; x = y$

($\mathsf{list}(\mathsf{nat})3_2$) $\quad \mathsf{cons}(x, l) = \mathsf{cons}(y, k) \;\Rightarrow\; l = k$

($\mathsf{list}(\mathsf{nat})\mathsf{S}$) $\quad \forall P.\; \big(\forall l.\; P(l) \;\Leftarrow\; \big(P(\mathsf{nil}) \;\wedge\; \forall x, k.\; \big(P(\mathsf{cons}(x, k)) \Leftarrow P(k)\big)\big)\big)$

Moreover, let us define the recursive functions $\mathsf{length}, \mathsf{count} : \mathsf{list}(\mathsf{nat}) \to \mathsf{nat}$, returning the length and the size of a list:

($\mathsf{length}1$) $\quad \mathsf{length}(\mathsf{nil}) \;\;\;\;\;\;\;\;\;\; = \mathsf{0}$

($\mathsf{length}2$) $\quad \mathsf{length}(\mathsf{cons}(x, l)) = \mathsf{s}(\mathsf{length}(l))$

($\mathsf{count}1$) $\quad \mathsf{count}(\mathsf{nil}) \;\;\;\;\;\;\;\;\;\; = \mathsf{0}$

($\mathsf{count}2$) $\quad \mathsf{count}(\mathsf{cons}(x, l)) = \mathsf{s}(x + \mathsf{count}(l))$

Note that the analogy of the axioms of lists to the axioms of the natural numbers is again not perfect:

1. There is an additional axiom ($\mathsf{list}(\mathsf{nat})3_1$), which has no analog among the axioms of the natural numbers.
2. Neither of the axioms ($\mathsf{list}(\mathsf{nat})3_1$) and ($\mathsf{list}(\mathsf{nat})3_2$) is implied by the axiom ($\mathsf{list}(\mathsf{nat})1$) together with the axiom
$$\mathsf{Wellf}(\lambda l, k.\ \exists x.\ (\mathsf{cons}(x, l) = k)),$$
which is the analog to Pieri's second axiom for the natural numbers.[47]
3. The latter axiom is weaker than each of the two axioms
$$\mathsf{Wellf}(\lambda l, k.\ (\mathsf{lessp}(\mathsf{length}(l), \mathsf{length}(k)) = \mathsf{true})),$$
$$\mathsf{Wellf}(\lambda l, k.\ (\mathsf{lessp}(\mathsf{count}(l), \mathsf{count}(k)) = \mathsf{true})),$$
which state the well-foundedness of bigger[48] relations. In spite of their relative strength, the well-foundedness of these relations is already implied by the well-foundedness that Pieri used for his specification of the natural numbers.

Therefore, the lists of natural numbers can be specified up to isomorphism by a specification of the natural numbers up to isomorphism (see § 4.4), plus the axioms ($\mathsf{list}(\mathsf{nat})3_1$) and ($\mathsf{list}(\mathsf{nat})3_2$), plus one of the following sets of axioms:

- ($\mathsf{list}(\mathsf{nat})2$), ($\mathsf{list}(\mathsf{nat})\mathsf{S}$) — in the style of Peano,
- ($\mathsf{list}(\mathsf{nat})1$), $\mathsf{Wellf}(\lambda l, k.\ \exists x.\ (\mathsf{cons}(x, l) = k))$ — in the style of Pieri,[49]
- ($\mathsf{list}(\mathsf{nat})1$), ($\mathsf{length}1$–$2$) — refining the style of Pieri.[50]

Today it is standard to avoid higher-order axioms in the way exemplified in the last of these three items,[51] and to get along with one second-order axiom for the natural numbers, or even with the first-order instances of that axiom.

[47] See § 4.4 for Pieri's specification of the natural numbers. The axioms ($\mathsf{list}(\mathsf{nat})3_1$) and ($\mathsf{list}(\mathsf{nat})3_2$) are not implied because all axioms besides ($\mathsf{list}(\mathsf{nat})3_1$) or ($\mathsf{list}(\mathsf{nat})3_2$) are satisfied in the structure where both natural numbers and lists are isomorphic to the standard model of the natural numbers, and where lists differ only in their sizes.

[48] Indeed, in case of $\mathsf{cons}(x, l) = k$, we have $\mathsf{lessp}(\mathsf{length}(l), \mathsf{length}(k)) =$
$= \mathsf{lessp}(\mathsf{length}(l), \mathsf{length}(\mathsf{cons}(x, l))) = \mathsf{lessp}(\mathsf{length}(l), \mathsf{s}(\mathsf{length}(l))) = \mathsf{true}$ because of ($\mathsf{lessp}4$), and we also have $\mathsf{lessp}(\mathsf{count}(l), \mathsf{count}(k)) = \mathsf{lessp}(\mathsf{count}(l), \mathsf{count}(\mathsf{cons}(x, l))) =$
$\mathsf{lessp}(\mathsf{count}(l), \mathsf{s}(x + \mathsf{count}(l))) = \mathsf{true}$ because of (+3) and ($\mathsf{lessp}5$).

[49] This option is essentially the choice of the "shell principle" of [Boyer and Moore, 1979, p.37ff.]: The one but last axiom of item (1) of the shell principle means ($\mathsf{list}(\mathsf{nat})2$) in our formalization, and guarantees that item (6) implies $\mathsf{Wellf}(\lambda l, k.\ \exists x.\ (\mathsf{cons}(x, l) = k))$.

[50] Although ($\mathsf{list}(\mathsf{nat})2$) follows from ($\mathsf{length}1$–$2$) and ($\mathsf{nat}2$), it should be included in this standard specification because of its frequent applications.

[51] For this avoidance, however, we have to admit the additional function $\mathsf{length}$. The same can be achieved with $\mathsf{count}$ instead of $\mathsf{length}$, which is only possible, however, for lists over element types that have a mapping into the natural numbers.

Moreover, as some of the most natural functions on lists, let us define the destructors $\mathsf{car} : \mathsf{list}(\mathsf{nat}) \to \mathsf{nat}$ and $\mathsf{cdr} : \mathsf{list}(\mathsf{nat}) \to \mathsf{list}(\mathsf{nat})$, both in constructor and destructor style. Furthermore, let us define the recursive member predicate $\mathsf{mbp} : \mathsf{nat},\ \mathsf{list}(\mathsf{nat}) \to \mathsf{bool}$, and $\mathsf{delfirst} : \mathsf{list}(\mathsf{nat}) \to \mathsf{list}(\mathsf{nat})$, a recursive function that deletes the first occurrence of a natural number in a list:

$$(\mathsf{car1}) \qquad \mathsf{car}(\mathsf{cons}(x,l)) = x$$

$$(\mathsf{cdr1}) \qquad \mathsf{cdr}(\mathsf{cons}(x,l)) = l$$

$$(\mathsf{car1}') \qquad \mathsf{car}(l') = x \ \Leftarrow\ l' = \mathsf{cons}(x,l)$$

$$(\mathsf{cdr1}') \qquad \mathsf{cdr}(l') = l \ \Leftarrow\ l' = \mathsf{cons}(x,l)$$

$$(\mathsf{mbp1}) \qquad \mathsf{mbp}(x,\mathsf{nil}) = \mathsf{false}$$
$$(\mathsf{mbp2}) \qquad \mathsf{mbp}(x,\mathsf{cons}(y,l)) = \mathsf{true} \ \Leftarrow\ x = y$$
$$(\mathsf{mbp3}) \qquad \mathsf{mbp}(x,\mathsf{cons}(y,l)) = \mathsf{mbp}(x,l) \ \Leftarrow\ x \neq y$$

$$(\mathsf{delfirst1}) \qquad \mathsf{delfirst}(x,\mathsf{cons}(y,l)) = l \ \Leftarrow\ x = y$$
$$(\mathsf{delfirst2}) \qquad \mathsf{delfirst}(x,\mathsf{cons}(y,l)) = \mathsf{cons}(y,\mathsf{delfirst}(x,l)) \ \Leftarrow\ x \neq y$$

Immediate consequences of the axiom $(\mathsf{list}(\mathsf{nat})1)$ and the definitions $(\mathsf{car1})$ and $(\mathsf{cdr1})$ are the lemma $(\mathsf{cons1})$ and its flattened version $(\mathsf{cons1}')$:

$$(\mathsf{cons1}) \qquad \mathsf{cons}(\mathsf{car}(l'),\mathsf{cdr}(l')) = l' \ \Leftarrow\ l' \neq \mathsf{nil}$$

$$(\mathsf{cons1}') \qquad \mathsf{cons}(x,l) = l' \ \Leftarrow\ l' \neq \mathsf{nil} \ \wedge\ x = \mathsf{car}(l') \ \wedge\ l = \mathsf{cdr}(l')$$

Furthermore, let us define the Boolean function $\mathsf{lexless} : \mathsf{list}(\mathsf{nat}), \mathsf{list}(\mathsf{nat}) \to \mathsf{bool}$, which lexicographically compares lists according to the ordering of the natural numbers, and $\mathsf{lexlimless} : \mathsf{list}(\mathsf{nat}), \mathsf{list}(\mathsf{nat}), \mathsf{nat} \to \mathsf{bool}$, which further restricts the length of the first argument to be less than the number given as third argument:

$$(\mathsf{lexless1}) \qquad \mathsf{lexless}(l,\mathsf{nil}) = \mathsf{false}$$
$$(\mathsf{lexless2}) \qquad \mathsf{lexless}(\mathsf{nil},\mathsf{cons}(y,k)) = \mathsf{true}$$
$$(\mathsf{lexless3}) \qquad \mathsf{lexless}(\mathsf{cons}(x,l),\mathsf{cons}(y,k)) = \mathsf{lexless}(l,k) \ \Leftarrow\ x = y$$
$$(\mathsf{lexless4}) \qquad \mathsf{lexless}(\mathsf{cons}(x,l),\mathsf{cons}(y,k)) = \mathsf{lessp}(x,y) \ \Leftarrow\ x \neq y$$

$$(\mathsf{lexlimless1}) \qquad \mathsf{lexlimless}(l,k,x) = \mathsf{and}(\mathsf{lexless}(l,k),\mathsf{lessp}(\mathsf{length}(l),x))$$

Such lexicographic combinations play an important rôle in well-foundedness arguments of induction proofs, because they combine given well-founded orderings into new well-founded orderings, provided there is an upper bound for the length of the list:[52]

$$(\mathsf{lexlimless2}) \qquad \mathsf{Wellf}(\lambda l,k.\ (\mathsf{lexlimless}(l,k,x) = \mathsf{true}))$$

Finally note that analogous axioms can be used to specify any other data type generated by constructors, such as pairs of natural numbers or binary trees over such pairs.

[52] The length limit is required because otherwise we have the following counterexample to termination: $(\mathsf{s}(0))$, $(0,\mathsf{s}(0))$, $(0,0,\mathsf{s}(0))$, $(0,0,0,\mathsf{s}(0))$, .... Note that the need to compare lists of different lengths typically arises in mutual induction proofs where the two induction hypotheses have a different number of free variables at measured positions. See [Wirth, 2004, § 3.2.2] for a nice example.

### *4.6 The Standard High-Level Method of Mathematical Induction*

In general, the intuitive and procedural aspects of a mathematical proof method are not completely captured by its logic formalization. For actually finding and automating proofs by induction, we also need effective heuristics.

In the everyday mathematical practice of an advanced theoretical journal, the common inductive arguments are hardly ever carried out explicitly. Instead, the proof reads something like "by structural induction on $n$, q.e.d." or "by (Noetherian) induction on $(x, y)$ over $<$, q.e.d.", expecting that the mathematically educated reader could easily expand the proof if in doubt. In contrast, difficult inductive arguments, sometimes covering several pages,[53] require considerable ingenuity and have to be carried out in the journal explicitly.

In case of a proof on natural numbers, the experienced mathematician might engineer his proof roughly according to the following pattern:

> He starts with the conjecture and simplifies it by case analysis, typically based on the axiom (nat1). When he realizes that the current goal is similar to an instance of the conjecture, he applies the instantiated conjecture just like a lemma, but keeps in mind that he has actually applied an induction hypothesis. Finally, using the free variables of the conjecture, he constructs some ordering whose well-foundedness follows from the axiom $\mathsf{Wellf}(\lambda x, y.\ (\mathsf{s}(x) = y))$ and in which all instances of the conjecture applied as induction hypotheses are smaller than the original conjecture.

The hard tasks of a proof by mathematical induction are thus:

**(Induction-Hypotheses Task)**
to find the numerous induction hypotheses;[54] and

**(Induction-Ordering Task)**
to construct an *induction ordering* for the proof, i.e. a well-founded ordering that satisfies the ordering constraints of all these induction hypotheses in parallel.[55]

The above induction method can be formalized as an application of the Theorem of Noetherian Induction. For non-trivial proofs, mathematicians indeed prefer the the axioms of Pieri's specification in combination with the Theorem of Noetherian Induction (N) to Peano's alternative with the Axiom of Structural Induction (S), because the instances for $P$ and $<$ in (N) are often easier to find than the instances for $P$ in (S) are.

[53] Such difficult inductive arguments are the proofs of Hilbert's *first $\varepsilon$-theorem* [Hilbert and Bernays, 1970], Gentzen's *Hauptsatz* [Gentzen, 1935], and confluence theorems such as the ones in [Gramlich and Wirth, 1996], [Wirth, 2009].

[54] As, e.g., in the proof of Gentzen's Hauptsatz on Cut-elimination.

[55] For instance, this was the hard part in the elimination of the $\varepsilon$-formulas in the proof of the 1st $\varepsilon$-theorem in [Hilbert and Bernays, 1970], and in the proof of the consistency of arithmetic by the $\varepsilon$-substitution method in [Ackermann, 1940].

## *4.7 Descente Infinie*

The soundness of the induction method of § 4.6 is most easily seen when the argument is structured as a proof by contradiction, assuming a counterexample. For Fermat's historic reinvention of the method, it is thus just natural that he developed the method in terms of assumed counterexamples.[56] Here is Fermat's Method of *Descente Infinie* in modern language, very roughly speaking:

> A proposition $P(w)$ can be proved by *descente infinie* as follows: *Show that for each assumed counterexample $v$ of $P$ there is a smaller counterexample $u$ of $P$ w.r.t. a well-founded relation $<$, which does not depend on the counterexamples.*

If this method is executed successfully, we have proved $\forall w.\ P(w)$ because no counterexample can be a $<$-minimal one, and so the well-foundedness of $<$ implies that there are no counterexamples at all.

Nowadays every logician immediately realizes that a formalization of the method of *descente infinie* is obtained from the Theorem of Noetherian Induction (N) (cf. § 4.2) simply by replacing

$$P(v) \ \Leftarrow\ \forall u{<}v.\ P(u)$$

with its contrapositive

$$\neg P(v) \ \Rightarrow\ \exists u{<}v.\ \neg P(u).$$

It was very hard for Fermat to obtain a positive version of his counterexample method.[57] The difference between an implication and its contrapositive, however, is irrelevant in our context here, which is the one of the 19th and 20th centuries and which is based on classical logic. What matters for us is the heuristic task of finding proofs. Therefore, we take *descente infinie* in this article[58] as a synonym for the modern standard high-level method of mathematical induction described in § 4.6.

Let us now prove the lemmas (+3) and (lessp7) of § 4.4 (in the axiomatic context of § 4.4) by *descente infinie*, seen as the standard high-level method of mathematical induction described in § 4.6.

---

[56] Cf. [Fermat, 1891ff.], [Mahoney, 1994], [Bussotti, 2006], [Wirth, 2010b].

[57] Fermat reported in his letter for Christiaan Huygens (1629–1695) that he had had problems applying the Method of *Descente Infinie* to positive mathematical statements. See [Wirth, 2010b, p. 11] and the references there, in particular [Fermat, 1891ff., Vol. II, p. 432].

Moreover, a natural-language presentation via *descente infinie* (such as Fermat's representation in Latin) is often simpler than a presentation via the Theorem of Noetherian Induction, because it is easier to speak of one counterexample $v$ and to find one smaller counterexample $u$, than to administrate the dependences of universally quantified variables.

[58] In general, in the tradition of [Wirth, 2004], *descente infinie* is nowadays taken as a synonym for the standard high-level method of mathematical induction as described in § 4.6. This way of using the term *"descente infinie"* is found in [Brotherston and Simpson, 2007; 2011], [Voicu and Li, 2009], [Wirth, 2005a; 2010a; 2013; 2012c].

If, however, the historical perspective before the 19th century is taken, then this identification is not appropriate because a more fine-grained differentiation is required, such as found in [Bussotti, 2006], [Wirth, 2010b].

EXAMPLE 2 (Proof of (+3) by *descente infinie*).

By application of the Theorem of Noetherian Induction (N) (cf. § 4.2) with $P$ set to $\lambda x, y.\ (x + y = y + x)$, and the variables $v$, $u$ renamed to $(x, y)$, $(x'', y'')$, respectively, the conjectured lemma (+3) reduces to

$$\exists <.\ \left( \begin{array}{ll} & \forall (x, y).\ \big( (x + y = y + x) \ \Leftarrow\ \forall (x'', y'') < (x, y).\ (x'' + y'' = y'' + x'') \big) \\ \wedge & \mathsf{Wellf}(<) \end{array} \right).$$

Let us focus on the sub-formula $x + y = y + x$. Based on axiom (nat1) we can reduce this task to the two cases $x = 0$ and $x = \mathsf{s}(x')$ with the two goals

$$0 + y = y + 0; \qquad \mathsf{s}(x') + y = y + \mathsf{s}(x');$$

respectively. They simplify by (+1) and (+2) to

$$y = y + 0; \qquad \mathsf{s}(x' + y) = y + \mathsf{s}(x');$$

respectively. Based on axiom (nat1) we can reduce each of these goals to the two cases $y = 0$ and $y = \mathsf{s}(y')$, which leaves us with the four open goals

$$0 = 0 + 0; \qquad \mathsf{s}(x' + 0) = 0 + \mathsf{s}(x');$$
$$\mathsf{s}(y') = \mathsf{s}(y') + 0; \qquad \mathsf{s}(x' + \mathsf{s}(y')) = \mathsf{s}(y') + \mathsf{s}(x').$$

They simplify by (+1) and (+2) to

$$0 = 0; \qquad \mathsf{s}(x' + 0) = \mathsf{s}(x');$$
$$\mathsf{s}(y') = \mathsf{s}(y' + 0); \qquad \mathsf{s}(x' + \mathsf{s}(y')) = \mathsf{s}(y' + \mathsf{s}(x'));$$

respectively. Now we instantiate the induction hypothesis that is available in the context[59] given by our above formula in four different forms, namely we instantiate $(x'', y'')$ with $(x', 0)$, $(0, y')$, $(x', \mathsf{s}(y'))$, and $(\mathsf{s}(x'), y')$, respectively. Rewriting with these instances, the four goals become:

$$0 = 0; \qquad \mathsf{s}(0 + x') = \mathsf{s}(x');$$
$$\mathsf{s}(y') = \mathsf{s}(0 + y'); \qquad \mathsf{s}(\mathsf{s}(y') + x') = \mathsf{s}(\mathsf{s}(x') + y');$$

which simplify by (+1) and (+2) to

$$0 = 0; \qquad \mathsf{s}(x') = \mathsf{s}(x');$$
$$\mathsf{s}(y') = \mathsf{s}(y'); \qquad \mathsf{s}(\mathsf{s}(y' + x')) = \mathsf{s}(\mathsf{s}(x' + y')).$$

Now the first three goals follow directly from the reflexivity of equality, whereas the last goal needs also an application of our induction hypothesis: This time we have to instantiate $(x'', y'')$ with $(x', y')$.

Finally, we instantiate our induction ordering $<$ to the lexicographic combination of length less than 3 of the ordering of the natural numbers. If we read our pairs as two-element lists, i.e. $(x'', y'')$ as $\mathsf{cons}(x'', \mathsf{cons}(y'', \mathsf{nil}))$, then we can set $<$ to $\lambda l, k.\ (\mathsf{lexlimless}(l, k, \mathsf{s}(\mathsf{s}(0)))) = \mathsf{true})$, which is well-founded according to (lexlimless2) (cf. § 4.5). Then it is trivial to show that $(\mathsf{s}(x'), \mathsf{s}(y'))$ is greater than each of $(x', 0)$, $(0, y')$, $(x', \mathsf{s}(y'))$, $(\mathsf{s}(x'), y')$, $(x', y')$.

This completes the proof of our conjecture by *descente infinie*. □

[59] On how this availability can be understood formally, see [Autexier, 2005].

EXAMPLE 3 (Proof of (lessp7) by *descente infinie*).

In the previous proof in Example 2 we made the application of the Theorem of Noetherian Induction most explicit, and so its presentation was rather formal w.r.t. the underlying logic.

Contrary to this, let us now proceed more in the vernacular of a working mathematician. Moreover, instead of $p = \mathsf{true}$, let us just write $p$.

To prove the strengthened transitivity of lessp as expressed in lemma (lessp7) in the axiomatic context of § 4.4, we have to show

$$\mathsf{lessp}(\mathsf{s}(x), z) \;\Leftarrow\; \mathsf{lessp}(x, y) \wedge \mathsf{lessp}(y, z).$$

Let us reduce the last literal. To this end, we apply the axiom (nat1) once to $y$ and once to $z$. Then, after reduction with (lessp1), the two base cases have an atom false in their conditions, abbreviating $\mathsf{false} = \mathsf{true}$, which is false according to (bool2), and so the base cases are true (*ex falso quodlibet*). The remaining case, where we have both $y = \mathsf{s}(y')$ and $z = \mathsf{s}(z')$, reduces with (lessp3) to

$$\mathsf{lessp}(x, z') \Leftarrow \mathsf{lessp}(x, \mathsf{s}(y')) \wedge \mathsf{lessp}(y', z')$$

If we apply the induction hypothesis instantiated via $\{y \mapsto y',\ z \mapsto z'\}$ to match the last literal, then we obtain the two goals

$$\mathsf{lessp}(x, z') \Leftarrow \mathsf{lessp}(x, \mathsf{s}(y')) \wedge \mathsf{lessp}(y', z') \wedge \mathsf{lessp}(\mathsf{s}(x), z')$$

$$\mathsf{lessp}(x, y') \vee \mathsf{lessp}(\mathsf{s}(x), z') \vee \mathsf{lessp}(x, z') \Leftarrow \mathsf{lessp}(x, \mathsf{s}(y')) \wedge \mathsf{lessp}(y', z')$$

By elimination of irrelevant literals, the first goal can be reduced to the valid conjecture $\mathsf{lessp}(x, z') \Leftarrow \mathsf{lessp}(\mathsf{s}(x), z')$, but we cannot obtain a lemma simpler than our initial conjecture (lessp7) by generalization and elimination of irrelevant literals from the second goal. This means that the application of the given instantiation of the induction hypothesis is useless.

Thus, instead of induction-hypothesis application, we had better apply the axiom (nat1) also to $x$, obtaining the cases $x = \mathsf{0}$ and $x = \mathsf{s}(x')$ with the two goals — after reduction with (lessp2) and (lessp3) —

$$\mathsf{lessp}(\mathsf{0}, z') \;\Leftarrow\; \mathsf{lessp}(y', z')$$

$$\mathsf{lessp}(\mathsf{s}(x'), z') \;\Leftarrow\; \mathsf{lessp}(x', y') \wedge \mathsf{lessp}(y', z'),$$

respectively. The first is trivial by (lessp1), (lessp2) after another application of the axiom (nat1) to $z'$. The second is just an instance of the induction hypothesis via $\{x \mapsto x',\ y \mapsto y',\ z \mapsto z'\}$. As the induction ordering we can select any of the variables of the original conjecture w.r.t. the irreflexive ordering on the natural numbers or w.r.t. the successor relation.

This completes the proof of the conjecture by *descente infinie*.

Note that we also have made clear that the given proof can only be successful with an induction hypotheses where all variables are instantiated with predecessors. It is actually possible to show that this simple example — *ceteris paribus* — requires an induction hypothesis resulting from an instance $\{x{\mapsto}x'',\ y{\mapsto}y'',\ z{\mapsto}z''\}$ where, for some meta-level natural number $n$, we have

$$x = \mathsf{s}^{n+1}(x'') \ \wedge\ y = \mathsf{s}^{n+1}(y'') \ \wedge\ z = \mathsf{s}^{n+1}(z''). \qquad \square$$

## *4.8 Explicit Induction*

### *4.8.1 From the Theorem of Noetherian Induction to Explicit Induction*

To admit the realization of the standard high-level method of mathematical induction as described in § 4.6, a proof calculus should have an explicit concept of an induction hypothesis. Moreover, it has to cope in some form with the second-order variables $P$ and $<$ in the Theorem of Noetherian Induction ($\mathsf{N}$) (cf. § 4.2), and with the second-order variable $Q$ in the definition of well-foundedness ($\mathsf{Wellf}(<)$) (cf. § 4.1).

Such an implementation needs special care regarding the calculus and its heuristics. For example, the theorem provers for higher-order logic with the strongest automation today[3] are yet not able to prove standard inductive theorems by just adding the Theorem of Noetherian Induction, which immediately effects an explosion of the search space. It is a main obstacle to practical usefulness of higher-order theorem provers that they are still poor in the automation of induction.

Therefore, it is probable that — on the basis of the logic calculi and the computer technology of the 1970s — Boyer and Moore would also have failed to implement induction via these human-oriented and higher-order features and were wise to confine the concept of an induction hypothesis to the internals of single reductive inference steps — namely the applications of the so-called *induction rule* — and to restrict all other inference steps to quantifier-free first-order deductive reasoning.

Described in terms of the Theorem of Noetherian Induction, this *induction rule* immediately instantiates the higher-order variables $P$ and $<$ with first-order predicates. This is rather straightforward for the predicate variable $P$, which simply becomes the (properly simplified and generalized) quantifier-free first-order conjecture that is to be proved by induction, and the tuple of the free first-order variables of this conjecture takes the place of the single argument of $P$; cf. Example 4 below.

The instantiation of the higher-order variable $<$ is more difficult: Instead of a simple instantiation, the whole context of its two occurrences is transformed. For the first occurrence, namely the one in the sub-formula $\forall u{<}v.\ P(u)$, the whole sub-formula is replaced with a conjunction of instances of $P(u)$, for which $u$ is known to be smaller than $v$ in some lexicographic combination of given orderings that are already known to be well-founded. As a consequence, the second occurrence of $<$, i.e. the one in $\mathsf{Wellf}(<)$, simplifies to true, and so we can drop the conjunction that contains it.

At a first glance, it seems highly unlikely that there could be any framework of proof-search heuristics in which such an induction rule could succeed in implementing all applications of the Theorem of Noetherian Induction, simply because this rule has to solve the two hard tasks of an induction proof, namely the Induction-Hypotheses Task and the Induction-Ordering Task (cf. § 4.6), right at the beginning of the proof attempt, before the proof has been sufficiently developed to exhibit its structural difficulties.

Most surprisingly, but as a matter of fact, the induction rule has proved to be most successful in realizing all applications of the Theorem of Noetherian Induction required within the proof-search heuristics of the Boyer–Moore waterfall (cf. Figure 1). Essential for this success is the relatively weak quantifier-free first-order logic:

- No new symbols have to be introduced during the proof, such as the ones of quantifier elimination. Therefore, the required instances of the induction hypothesis can already be denoted when the induction rule is applied.[60]

- A general peculiarity of induction,[61] namely that the formulation of lemmas often requires the definition of new recursive functions, is aggravated by the weakness of the logic; and the user is actually required to provide further guidance for the induction rule via these new function definitions.[62]

Moreover, this success crucially depends on the possibility to generate additional lemmas that are proved by subsequent inductions, which is best shown by an example.

EXAMPLE 4 (Proof of (+3) by explicit induction).
Let us prove (+3) in the context of § 4.4, just as we have done already in Example 2 (cf. § 4.7), but now with the induction rule as the only way to apply the Theorem of Noetherian Induction.

As the conjecture is already properly simplified and concise, we instantiate $P(w)$ in the Theorem of Noetherian Induction again to the whole conjecture and reduce this conjecture by application of the Theorem of Noetherian Induction again to

$$\exists <.\ \left(\begin{array}{ll} & \forall(x,y).\ \big((x+y=y+x)\ \Leftarrow\ \forall(x'',y'')<(x,y).\ (x''+y''=y''+x'')\big) \\ \wedge & \mathsf{Wellf}(<) \end{array}\right).$$

Based, roughly speaking, on a termination analysis for the function $+$, the heuristic of the induction rule of explicit induction suggests to instantiate $<$ to $\lambda(x'',y''),(x,y).\ (\mathsf{s}(x'')=x)$.

[60] Cf. Note 64.

[61] See item 2 of § 4.10.

[62] Cf. § 9.

As this relation is known to be well-founded, the induction rule reduces the task based on axiom (nat1) to two goals, namely the base case

$$\mathsf{0}+y=y+\mathsf{0};$$

and the step case

$$(\mathsf{s}(x')+y=y+\mathsf{s}(x')) \;\Leftarrow\; (x'+y=y+x').$$

This completes the application of the induction rule. Thus, instances of the induction hypothesis can no longer be applied in the further proof (except the ones that have been added explicitly as conditions of step cases by the induction rule).

The induction rules of the Boyer–Moore theorem provers are not able to find the many instances we applied in the proof of Example 2. This is different for a theoretically more powerful induction rule suggested by Christoph Walther (*1950), which actually finds the proof of Example 2.[63] In general, however, for harder conjectures, a simulation of *descente infinie* by the induction rule of explicit induction would require an arbitrary look-ahead into the proofs, depending on the size of the structure of these proofs; thus, because the induction rule is understood to have a limited look-ahead into the proofs, such a simulation would not fall under the paradigm of explicit induction any more. Indeed, the look-ahead of induction rules into the proofs is typically not more than a single unfolding of a single occurrence of a recursive function symbol, for each such occurrence in the conjecture.

Note that the two above goals of the base and the step case can also be obtained by reducing the input conjecture with an instance of axiom (S) (cf. § 4.4), i.e. with the Axiom of Structural Induction over 0 and s. Nevertheless, the induction rule of the Boyer–Moore theorem provers is, in general, able to produce much more complicated base and step cases than those that can be obtained by reduction with the axiom (S).

Now the first goal is simplified again to $y=y+\mathsf{0}$, and then another application of the induction rule results in two goals that can be proved without further induction.

The second goal is simplified to

$$(\mathsf{s}(x'+y)=y+\mathsf{s}(x')) \;\Leftarrow\; (x'+y=y+x').$$

Now we use the condition from *left* to right for rewriting only the *left*-hand side of the conclusion and then we throw away the condition completely, with the intention to obtain a stronger induction hypothesis in a subsequent induction proof. This is the famous *"cross-fertilization"* of the Boyer–Moore waterfall (cf. Figure 1). By this, the simplified second goal reduces to

$$\mathsf{s}(y+x')=y+\mathsf{s}(x').$$

[63] See [Walther, 1993, p. 99f.]. On Page 100, the most interesting step case computed by Walther's induction rule is (rewritten to constructor-style):

$$\mathsf{s}(x)+\mathsf{s}(y)=\mathsf{s}(y)+\mathsf{s}(x) \;\Leftarrow\; (\; x+\mathsf{s}(y)=\mathsf{s}(y)+x \;\wedge\; \forall z.\,(z+y=y+z)\;).$$

In practice, however, Walther's induction rule has turned out to be overall less successful when applied within a heuristic framework similar to the Boyer–Moore waterfall (cf. Figure 1).

Now the induction rule triggers a structural induction on $y$, which is successful without further induction.

All in all, although the induction rule of the Boyer–Moore theorem provers does not find the more complicated induction hypotheses of the *descente infinie* proof of Example 2 in § 4.7, it is well able prove our original conjecture with the help of the additional lemmas $y = y + 0$ and $\mathsf{s}(y + x') = y + \mathsf{s}(x')$.

It is crucial here that the heuristics of the Boyer–Moore waterfall discover these lemmas automatically, and that this is also typically the case in general.

From a logical viewpoint, these lemmas are redundant because they follow from the original conjecture and the definition of $+$. From a heuristic viewpoint, however, they are more useful than the original conjecture, because — oriented for rewriting from right to left — their application tends to terminate in the context of the overall simplification by symbolic evaluation, which constitutes the first stage of the waterfall. □

Although the two proofs of the very simple conjecture (+3) given in Examples 2 and 4 can only give a very rough idea on the advantage of *descente infinie* for hard induction proofs,[64] these two proofs nicely demonstrate how the induction rule of explicit induction manages to prove simple theorems very efficiently and with additional benefits for the further performance of the simplification procedure.

Moreover, for proving very hard theorems for which the overall waterfall heuristic fails, the user can state hints and additional lemmas with additional notions in any Boyer–Moore theorem prover, except the Pure LISP Theorem Prover.

### *4.8.2 Theoretical Viewpoint on Explicit Induction*

From a theoretical viewpoint, we have to be aware of the possibility that the intended models of specifications in *explicit-induction systems* may also include non-standard models.

[64] For some of the advantages of *descente infinie*, see Example 12 in § 6.2.6, and especially the more difficult, complete formal proof of Max H. A. Newman's famous lemma in [Wirth, 2004, § 3.4], where the reverse of a well-founded relation is shown to be confluent in case of local confluence — *by induction w.r.t. this well-founded relation itself*. The induction rule of explicit induction cannot be applied here because an eager induction hypothesis generation is not possible: The required instances of the induction hypothesis contain $\delta$-variables that can only be generated later during the proof by quantifier elimination.

Though confluence is the Church–Rosser property, the Newman Lemma has nothing to do with the Church–Rosser Theorem stating the confluence of the rewrite relation of $\alpha\beta$-reduction in untyped $\lambda$-calculus, which has actually been verified with a Boyer–Moore theorem prover in the first half of the 1980s by Shankar [1988] (see the last paragraph of § 6.4 and Note 175) following the short Tait/Martin-Löf proof found e.g. in [Barendregt, 2012, p. 59ff.]. Unlike the Newman Lemma, Shankar's proof proceeds by structural induction on the $\lambda$-terms, not by Noetherian induction w.r.t. the reverse of the rewrite relation; indeed, untyped $\lambda$-calculus is not terminating.

For the natural numbers, for instance, there may be **Z**-chains in addition to the natural numbers **N**, whereas the higher-order specifications of Peano and Pieri specify exactly the natural numbers **N** up to isomorphism.[65] This is indeed the case for the case of the Boyer–Moore theorem provers as explained in Note 138. These **Z**-chains cannot be excluded because the inference rules realize only first-order deductive reasoning, except for the induction rule to which all applications of the Theorem of Noetherian Induction are confined and which does not use any higher-order properties, but only well-founded orderings that are defined in the first-order logic of the explicit-induction system.

### *4.8.3 Practical Viewpoint on Explicit Induction*

Note that the application of the induction rule of explicit induction is not implemented via a reference to the Theorem of Noetherian Induction, but directly handles the following practical tasks and their heuristic decisions.

In general, the *induction stage* of the Boyer–Moore waterfall (cf. Figure 1) applies the induction rule once to its input formula, which results in a conjunction — or conjunctive set — of base and step cases to which the input conjecture reduces, i.e. whose validity implies the validity of the input conjecture.

Therefore, a working mathematician would expect that the induction rule of explicit induction solves the following two tasks:

1. Choose some of the variables in the conjecture as *induction variables*, and split the conjecture into several base and step cases, based on the induction variables' demand on which governing conditions and constructor substitutions[66] have to be added to be able to unfold — without further case analysis — some of the recursive function calls that contain the induction variables as direct arguments.

2. Eagerly generate the induction hypotheses for the step cases.

The actual realization of these tasks in the induction rule, however, is quite different from these expectations: Induction variables play only a very minor rôle toward the end of the procedure (in the deletion of flawed induction schemes, cf. § 6.3.8), the focus is on complete step cases including eagerly generated induction hypotheses, and the complementing bases case are generated only at the very end.[67]

[65] Contrary to the **Z**-chains (which are structures similar to the integers **Z**, injectively generated from an arbitrary element via $\mathsf{s}$ and its inverse, where every element is greater than every standard natural number), "$\mathsf{s}$-circles" cannot exist because it is possible to show by structural induction on $x$ the two lemmas $\mathsf{lessp}(x, x) = \mathsf{false}$ and $\mathsf{lessp}(x, \mathsf{s}^{n+1}(x)) = \mathsf{true}$ for each standard meta-level natural number $n$.

[66] This adding of constructor substitutions refers to the application of axioms like (nat1) (cf. § 4.4), and is required whenever constructor style either is found in the recursive function definitions or is to be used for the step cases. In the Pure LISP Theorem Prover, only the latter is the case. In Thm, none is the case.

[67] See, e.g., Example 10 of § 5.8.

## *4.9 Generalization*

Contrary to merely deductive, analytic theorem proving, an input conjecture for a proof by induction is not only a task (as induction conclusion) but also a tool (as induction hypothesis) in the proof attempt. Therefore, a stronger conjecture is often easier to prove because it supplies us with a stronger induction hypothesis during the proof attempt.

Such a step from a weaker to a stronger input conjecture is called *generalization*.

Generalization is to be handled with great care because it is an *unsafe* reduction step in the sense that it may reduce a valid conjecture to an invalid one; such a reduction is called *over-generalization*.

Generalization of input conjectures directly supplied by humans is rarely helpful because stating sufficiently general theorems is part of the standard mathematical training in induction. As we have seen in Example 4 of § 4.8.1, however, explicit induction often has to start another induction during the proof, and then the secondary, machine-generated input conjecture often requires generalization.

The two most simple syntactical generalizations are the replacement of terms with fresh universal variables and the removal of irrelevant side conditions.

In the vernacular of Boyer–Moore theorem provers, the first is simply called "generalization" and the second is called "elimination of irrelevance". They are dealt with in two consecutive stages of these names in the Boyer–Moore waterfall, which come right before the induction stage.

The removal of irrelevant side conditions is intuitively clear. For formulas in clausal form, it simply means to remove irrelevant literals. More interesting are the heuristics of its realization, which we discuss in § 6.3.5.

The less clear process of generalization typically proceeds by the replacement of all occurrences of a non-variable[68] term with a fresh variable.

This is especially promising for a subsequent induction if the same non-variable term has multiple occurrences in the conjecture, and becomes even more promising if these occurrences are found on both sides of the same positive equation or in literals of different polarity, say in a conclusion and a condition of an implication.

To avoid *over-generalization*, subterms are to be preferred to their super-terms,[69] and one should never generalize a term of any of the following forms: a constructor term, a top level term, a term with a logical operator (such as implication or equality) as top symbol, a direct argument of a logical operator, or the first argument of a conditional (`IF`). Indeed, for any of these forms, the information loss by generalization is typically so high that the generalization results in an invalid conjecture.

How powerful generalization can be is best seen by the multitude of its successful automatic applications, which often surprise humans. Here is one of these:

[68] Besides the replacement of (typically all) the occurrences of a non-variable term, there is also the possibility of replacing some — *but not all* — occurrences of a variable with a fresh variable. This is a very delicate process, but heuristics for it were discussed very early, namely in [Aubin, 1976, § 3.3].

[69] This results in a weaker conjecture and the stronger one remains available by generalization.

EXAMPLE 5 (Proof of (ack4) by Explicit Induction and Generalization).
Let us prove (ack4) in the context of § 4.4 by explicit induction. It is obvious that such a proof has to follow the definition of ack in the three cases (ack1), (ack2), (ack3), using the termination ordering of ack, which is just the lexicographic combination of its arguments. So the induction rule of explicit induction reduces the input formula (ack4) to the following goals:[70]

$$\mathsf{lessp}(y, \mathsf{ack}(0, y)) = \mathsf{true};$$

$$\mathsf{lessp}(0, \mathsf{ack}(\mathsf{s}(x'), 0)) = \mathsf{true} \;\Leftarrow\; \mathsf{lessp}(\mathsf{s}(0), \mathsf{ack}(x', \mathsf{s}(0))) = \mathsf{true};$$

$$\mathsf{lessp}(\mathsf{s}(y'), \mathsf{ack}(\mathsf{s}(x'), \mathsf{s}(y'))) = \mathsf{true} \;\Leftarrow\; \left(\begin{array}{ll} & \mathsf{lessp}(y', \mathsf{ack}(\mathsf{s}(x'), y')) = \mathsf{true} \\ \wedge & \mathsf{lessp}(\mathsf{ack}(\mathsf{s}(x'), y'), \mathsf{ack}(x', \mathsf{ack}(\mathsf{s}(x'), y'))) = \mathsf{true} \end{array}\right).$$

After simplifying with (ack1), (ack2), (ack3), respectively, we obtain:

$$\mathsf{lessp}(y, \mathsf{s}(y)) = \mathsf{true};$$

$$\mathsf{lessp}(0, \mathsf{ack}(x', \mathsf{s}(0))) = \mathsf{true} \;\Leftarrow\; \mathsf{lessp}(\mathsf{s}(0), \mathsf{ack}(x', \mathsf{s}(0))) = \mathsf{true};$$

$$\mathsf{lessp}(\mathsf{s}(y'), \mathsf{ack}(x', \mathsf{ack}(\mathsf{s}(x'), y'))) = \mathsf{true} \;\Leftarrow\; \left(\begin{array}{ll} & \mathsf{lessp}(y', \mathsf{ack}(\mathsf{s}(x'), y')) = \mathsf{true} \\ \wedge & \mathsf{lessp}(\mathsf{ack}(\mathsf{s}(x'), y'), \mathsf{ack}(x', \mathsf{ack}(\mathsf{s}(x'), y'))) = \mathsf{true} \end{array}\right).$$

Now the base case is simply an instance of our lemma (lessp4). Let us simplify the two step cases by introducing variables for their common subterms:

$$\mathsf{lessp}(0, z) = \mathsf{true} \;\Leftarrow\; \big(\; \mathsf{lessp}(\mathsf{s}(0), z) = \mathsf{true} \;\wedge\; z = \mathsf{ack}(x', \mathsf{s}(0)) \;\big);$$

$$\mathsf{lessp}(\mathsf{s}(y'), z_2) = \mathsf{true} \;\Leftarrow\; \left(\begin{array}{lll} & \mathsf{lessp}(y', z_1) = \mathsf{true} & \wedge\; \mathsf{lessp}(z_1, z_2) = \mathsf{true} \\ \wedge & z_1 = \mathsf{ack}(\mathsf{s}(x'), y') & \wedge\; z_2 = \mathsf{ack}(x', z_1) \end{array}\right).$$

Now the first follows from applying (nat1) to $z$. Before we can prove the second by another induction, however, we have to generalize it to the lemma (lessp7) of § 4.4 by deleting the last two literals from the condition. □

In combination with explicit induction, generalization becomes especially powerful in the invention of new lemmas of general interest, because the step cases of explicit induction tend to have common occurrences of the same term in their conclusion and their condition. Indeed, the lemma (lessp7), which we have just discovered in Example 5, is one of the most useful lemmas in the theory of natural numbers.

It should be noted that all Boyer–Moore theorem provers except the PURE LISP THEOREM PROVER are able to do this whole proof completely automatically and invent the lemma (lessp7) by generalization of the second step case; and they do this even when they work with an arithmetic theory that was redefined, so that no decision procedures or other special knowledge on the natural numbers can be used by the system. Moreover, as shown in § 3.3 of [Wirth, 2004], in a slightly richer logic, these heuristics can actually synthesize the lower bound in the first argument of lessp from the weaker input conjecture $\exists z.\ (\mathsf{lessp}(z, \mathsf{ack}(x, y)) = \mathsf{true})$, simply because lessp does not contribute to the choice of the base and step cases.

[70] See Example 10 of § 5.8 on how these step cases are actually found in explicit induction.

## *4.10 Proof-Theoretical Peculiarities of Mathematical Induction*

The following two proof-theoretical peculiarities of induction compared to first-order deduction may be considered noteworthy:[71]

1. A calculus for arithmetic cannot be complete, simply because the theory of the arithmetic of natural numbers is not enumerable.[72]

2. According to Gentzen's Hauptsatz,[73] a proof of a first-order theorem can always be restricted to the "sub"-formulas of this theorem. In contrast to lemma application in a deductive proof tree, however, the application of induction hypotheses and lemmas inside an inductive reasoning cycle cannot generally be eliminated in the sense that the "sub"-formula property could be obtained.[74] As a consequence, in first-order inductive theorem proving, "creativity" cannot be restricted to finding just the proper instances, but may require the invention of new lemmas and notions.[75]

## *4.11 Conclusion*

In this section, after briefly presenting the induction method in its rich historical context, we have offered a formalization and a first practical description. Moreover, we have explained why we can take Fermat's term *"descente infinie"* in our modern context as a synonym for the standard high-level method of mathematical induction. Finally, we have introduced explicit induction and generalization.

Noetherian induction requires domains for its well-founded orderings; and these domains are typically built-up by constructors. Therefore, the discussion of the method of induction required the introduction of some paradigmatic data types, such as natural numbers and lists.

To express the relevant notions on these data types, we need *recursion*, a method of definition, which we have often used in this section intuitively. We did not discuss its formal admissibility requirements yet. We will do so in § 5, with a focus on modes of recursion that admit an effective consistency test, including termination aspects such as induction templates and schemes.

---

[71] Note, however, that these peculiarities of induction do not make a difference to first-order deductive theorem proving *in practice*. See Notes 72 and 75.

[72] This theoretical result is given by Gödel's first incompleteness theorem [1931]. In practice, however, it does not matter whether our proof attempt fails because our theorem will not be enumerated ever, or will not be enumerated before doomsday.

[73] Cf. [Gentzen, 1935].

[74] Cf. [Kreisel, 1965].

[75] In practice, however, proof search for harder theorems often requires the introduction of lemmas, functions, and relations, and it is only a matter of degree whether we have to do this for principled reasons (as in induction) or for tractability (as required in first-order deductive theorem proving, cf. [Baaz and Leitsch, 1995]).

# 5 RECURSION, TERMINATION, AND INDUCTION

## 5.1 Recursion and the Rewrite Relation on Ground Terms

*Recursion* is a form of programming or definition where a newly defined notion may even occur in its *definientia*. 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 *definientia* (*right-hand sides*)), reduction with *recursive* definitions may run forever.

We have already seen some recursive function definitions in §§ 4.4 and 4.5, such as the ones of $+$, $\mathsf{lessp}$, $\mathsf{length}$, and $\mathsf{count}$, 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 § 4.4.

The steps of rewriting with recursive definitions can be formalized as a binary relation on terms, namely as 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.[76]

A *ground* term is a term without variables. We can restrict our considerations here to rewrite relations *on ground terms*.

## 5.2 Confluence

The restriction that is to be required for every recursive function definition is the *confluence*[77] of this rewrite relation on ground terms.

The confluence restriction guarantees that no distinct objects of the data types can be equated by the recursive function definitions.[78]

This is essential for consistency if we assume axioms such as ($\mathsf{nat}2$–3) (cf. § 4.4) or ($\mathsf{list}(\mathsf{nat})2$–3) (cf. § 4.5).

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 $\mathsf{p}(x) = \mathsf{0}$ as a further defining equation to ($\mathsf{p}1$), then we would get $\mathsf{s}(\mathsf{0}) = \mathsf{p}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) = \mathsf{0}$, in contradiction to the axiom ($\mathsf{nat}2$) of § 4.4.

---

[76] For 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).

[77] A 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: $\overset{+}{\longleftarrow} \circ \overset{+}{\longrightarrow} \subseteq \overset{*}{\longrightarrow} \circ \overset{*}{\longleftarrow}$. Here $\circ$ denotes the concatenation of binary relations; for the further notation see § 4.1.

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

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,[79] and that all variables occurring on the right-hand side of the definition also occur on the left-hand side of the defining equation.[80]

These two restrictions are an immediate consequence of the general definition 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.[81]

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

```
(PLUS X Y) = (IF (ZEROP X)
                 (FIX Y)
                 (ADD1 (PLUS (SUB1 X) Y)))
```

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 § 4.4. 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 § 4.4, 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:

$$(+1') \qquad x + y = y \qquad \Leftarrow \quad x = \mathsf{0}$$
$$(+2') \qquad x + y = \mathsf{s}(\mathsf{p}(x) + y) \quad \Leftarrow \quad x \neq \mathsf{0}$$

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. □

[79] Cf. 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].

[80] Cf. item (*c*) of the "definition principle" of [Boyer and Moore, 1979, p. 44f.].

If we considered the recursive equation (+2) together with the alternative recursive equation (+2′), then we could rewrite $\mathsf{s}(x)+y$ on the one hand with (+2) into $\mathsf{s}(x+y)$, and, on the other hand, with (+2′) into $\mathsf{s}(\mathsf{p}(\mathsf{s}(x))+y)$. 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. The only known 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].[82] 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.

## *5.3 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 that contain a defined function symbol w.r.t. the rewrite relation.

The requirement of termination should be intuitively clear; we will further discuss it in § 5.5.

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 $\mathsf{p}$ we did not specify any value for $\mathsf{p}(\mathsf{0})$; so $\mathsf{p}(\mathsf{0})$ 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.[83]

[81] See [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.

[82] The 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. § 5.4), 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.

[83] Cf. item (*b*) of the "definition principle" of [Boyer and Moore, 1979, p. 44f.].

## *5.4 Constructor Variables*

These restrictions of reducibility and termination of the rewrite relation are not essential; neither for the semantics of recursive function definitions with data types given by constructors,[84] nor for confluence and consistency.[85]

Note that these two restrictions imply that only *total recursive* functions[86] 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.

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 *ground* terms.

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

$$(\mathsf{russell1}) \quad \mathsf{russell}(b) = \mathsf{false} \;\Leftarrow\; \mathsf{russell}(b) = \mathsf{true}$$
$$(\mathsf{russell2}) \quad \mathsf{russell}(b) = \mathsf{true} \;\Leftarrow\; \mathsf{russell}(b) = \mathsf{false}$$

Then it is claimed that this function definition would result in an inconsistent specification on the basis of the axioms (bool1–2) of § 4.5.

This inconsistency, however, arises only if the variable $b$ of the axiom (bool1) can be instantiated with the term $\mathsf{russell}(b)$, 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*[87] 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 $\mathsf{russell}(b)$ can denote *junk objects* different from true and false, and no inconsistency arises.[88]

Note that these constructor variables are implicitly part of the LISP semantics with its innermost evaluation strategy. For instance, in Example 6 of § 5.2, 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

[84] Cf. [Wirth and Gramlich, 1994b].

[85] Cf. [Wirth, 2009].

[86] You may follow the explicit reference to [Schoenfield, 1967] as the basis for the logic of the Pure LISP Theorem Prover on Page 93 of [Moore, 1973].

[87] Such *constructor variables* were formally introduced for the first time 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 [Schmidt-Samoa, 2006a; 2006b; 2006c].

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

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` reads[89]

$$\exists \texttt{Z (IF X (IF Y T NIL) NIL) = Z},$$

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 § 5.5, 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).

## *5.5 Termination and General Induction Templates*

In addition to the restricted style of recursive definition that is found in LISP and that guarantees reducibility of terms with defined function symbols and confluence as described in §§ 5.3 and 5.4, 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,[90] *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 $f_k$.[91]

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 induction 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.

[89] In 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 tacitly write "`IF`" here and in what follows for every "`COND`" of the Pure LISP Theorem Prover.

[90] Note 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.

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

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*[92] 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 $f_k(t_1, \ldots, t_{n_k})$, 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 the case of constructor-style definitions, and just sufficient to guarantee proper destructor applications in the case of destructor-style definitions.
   Together they form the triple $(f_k(t_1, \ldots, t_{n_k}), R, C)$, and a set containing such a triple for each such defining equation forms the relational description.
   For our definition of + via (+1), (+2) in § 4.4, there is only one recursive equation and only one relevant relational description, namely the following one with an empty case condition:
   $$\{\ (\ \mathsf{s}(x)+y,\ \ \{x+y\},\ \ \emptyset\ )\ \}.$$
   Also for our definition of + with (+1′), (+2′) in Example 6, there is only one recursive equation and only one relevant relational description, namely
   $$\{\ (\ x+y,\ \ \{\mathsf{p}(x)+y\},\ \ \{x \neq \mathsf{0}\}\ )\ \}.$$

2. For each $k \in K$, a variable-free weight term $w_{f_k}$ in which the position numbers $(1), \ldots, (n_k)$ 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 $\lambda x, y.\ (\mathsf{lessp}(x, y) = \mathsf{true})$, is appropriate.

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

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

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

[92] The name "relational description" comes from [Walther, 1992; 1993].

EXAMPLE 7 (Two Induction Templates with different Measured Positions).
For the ordering predicate lessp as defined by (lessp1–3) of § 4.4, we get two appropriate induction templates with the sets of measured positions $\{1\}$ and $\{2\}$, respectively, both with the relational description

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

and both with the well-founded ordering $\lambda x, y.\ (\mathsf{lessp}(x,y) = \mathsf{true})$. 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 § 4.4. □

EXAMPLE 8 (One Induction Template with Two Measured Positions).
For the Ackermann function ack as defined by (ack1–3) of § 4.4, we get only one appropriate induction template. The set of its measured positions is $\{1,2\}$, because of the weight function $\mathsf{cons}((1),\mathsf{cons}((2),\mathsf{nil}))$, which we will abbreviate in the following with $[(1),(2)]$. The well-founded relation is the lexicographic ordering $\lambda l, k.\ (\mathsf{lexlimless}(l,k,\mathsf{s}(\mathsf{s}(0)))) = \mathsf{true})$. The relational description has two elements: For the equation (ack2) we get

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

and for the equation (ack3) we get

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

The validity of the template is expressed in the three equations

$$\begin{array}{llll}
\mathsf{lexlimless}([x,\mathsf{s}(0)], & [\mathsf{s}(x),0], & \mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) & = \quad \mathsf{true};\\
\mathsf{lexlimless}([\mathsf{s}(x),y], & [\mathsf{s}(x),\mathsf{s}(y)], & \mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) & = \quad \mathsf{true};\\
\mathsf{lexlimless}([x,\mathsf{ack}(\mathsf{s}(x),y)], & [\mathsf{s}(x),\mathsf{s}(y)], & \mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) & = \quad \mathsf{true};
\end{array}$$

which follow deductively from (lessp4), (lexlimless1), (lexless2–4), (length1–2). □

For valid induction templates of destructor-style definitions see Examples 18 and 19 in § 6.3.7.

## *5.6 Termination of the Rewrite Relation on Ground Terms*

Let us prove 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 after addition of the new positive/negative-conditional equations for the $f_k$, assuming any arbitrary model $\mathcal{M}$ of the old equations to be given.

For an *argumentum ad absurdum*, suppose that there is an infinite sequence of rewrite steps on ground terms. Consider each term in this sequence to be replaced with the multiset that contains, for each occurrence of a function call $f_k(t_1,\ldots,t_{n_k})$ with $k \in K$, the value of its weight term $w_{f_k}\{(1)\mapsto t_1,\ldots,(n_k)\mapsto t_{n_k}\}$ in $\mathcal{M}$.

Then the rewrite steps with instances of the *old* equations of previous function definitions (of symbols not among the $f_k$) can change the multiset only by deleting some elements for the following two reasons: Instances that do not contain any new function symbol have no effect on the values in $\mathcal{M}$, because $\mathcal{M}$ is a model of the old equations. There are no other instances because the new function symbols do not occur in the old equations, and because we consider all our variables to be constructor variables as explained in § 5.4.[93]

Moreover, a rewrite step with a *new* equation reduces the multiset in a well-founded relation, namely the multiset extension of the well-founded relation of the template in the assumed model $\mathcal{M}$. This follows from the fulfilledness of the conditions of the equation and the validity of the template.

Thus, in each rewrite step, 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.

## *5.7 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 induction templates are hardly helpful for finding proofs involving non-trivially mutually recursive functions.[94]

Moreover, in principle, users can always encode mutually recursive functions $f_k(\ldots)$ by means of a single recursive function $f(k,\ldots)$. Via such an encoding, humans tend to provide additional heuristic information relevant for induction templates, 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 § 5.5 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.[95]

*Changeable* are those measured positions $i$ of the template which sometimes change in the recursion, i.e. for which there is a triple $(f(t_1,\ldots,t_n),R,C)$ in the relational description of the template, and an $f(t'_1,\ldots,t'_n) \in R$ 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).

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

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

[93] Among the old equations here, we may even admit projective equations with *general* variables, such as for destructors and the conditional function $\mathsf{IfThenElse_{nat}} : \mathsf{bool},\mathsf{nat},\mathsf{nat} \to \mathsf{nat}$:

$$\mathsf{p}(\mathsf{s}(X)) = X \quad \Big| \quad \begin{array}{l}\mathsf{car}(\mathsf{cons}(X,L)) = X \\ \mathsf{cdr}(\mathsf{cons}(X,L)) = L\end{array} \quad \Big| \quad \begin{array}{l}\mathsf{IfThenElse_{nat}}(\mathsf{true},X,Y) = X \\ \mathsf{IfThenElse_{nat}}(\mathsf{false},X,Y) = Y\end{array}$$

for general variables $X,Y : \mathsf{nat}$, $L : \mathsf{list}(\mathsf{nat})$, ranging over general terms (instead of constructor terms only). Moreover, we can drop all typing restrictions because they are irrelevant here.

[94] See, however, [Kapur and Subramaniam, 1996] for explicit-induction heuristics applicable to simple forms of mutual recursion.

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

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.[96] For templates of constructor-style equations we additionally have to require here that the first element $f(t_1, \ldots, t_n)$ of each triple of the relational description of the template matches $(f(t''_1, \ldots, t''_n))\xi$ for some *constructor substitution* $\xi$ that may replace the variables of $f(t''_1, \ldots, t''_n)$ 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 § 4.4. 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 $\mathsf{lessp}(\mathsf{s}(x), \mathsf{s}(y))$ cannot be matched to $(\mathsf{lessp}(y, \mathsf{ack}(x, y)))\xi$ for any constructor substitution $\xi$. □

## 5.8 Induction Schemes

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

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

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 formula $A[f(t''_1, \ldots, t''_n)]$ given as input to the induction rule of explicit induction. Let $\sigma$ be the substitution whose domain are the variables of $f(t_1, \ldots, t_n)$ and which matches the first element $f(t_1, \ldots, t_n)$ of the triple to $(f(t''_1, \ldots, t''_n))\xi$ for some constructor substitution $\xi$ whose domain are the variables of $f(t''_1, \ldots, t''_n)$, 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 $i \in \{1, \ldots, n\}$.

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

$$\Big(\ \big(A[f(t''_1, \ldots, t''_n)]\big)\xi \ \ \Leftarrow\ \ \bigwedge\nolimits_{j \in J} \big(A[f(t''_1, \ldots, t''_n)]\big)\mu_j\ \Big) \ \ \Leftarrow\ \ \bigwedge C\sigma$$

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

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 $\mu_j$ 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 $f(t''_1, \ldots, t''_n)$, and we can satisfy the constraint simply by defining $t''_i\mu_j := t'_{j,i}\sigma$.

On the remaining variables of $f(t''_1,\ldots,t''_n)$, we define $\mu_j$ 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 $(A[f(t''_1,\ldots,t''_n)])\mu_j$ after unfolding $f(t''_1,\ldots,t''_n)\xi$ in $(A[f(t''_1,\ldots,t''_n)])\xi$. Note that such an eager instantiation is required in explicit induction unless the logic admits one of the following: existential quantification, existential variables,[97] 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 $f(t''_1,\ldots,t''_n)$ in $A[f(t''_1,\ldots,t''_n)]$. 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 $f(t''_1,\ldots,t''_n)$.
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
$$\big(\ \xi,\ \ \{\,\mu_j \mid j\in J\,\},\ \ C\sigma\ \big).$$
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*[98] of all substitutions $\mu_j$ with $j\in J$ given by
$$\frac{|\{\ (j,i)\in J\times\{1,\ldots,n\}\ \mid\ t''_i\mu_j = t'_{j,i}\sigma\ \}|}{|J\times\{1,\ldots,n\}|},$$
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,[99] but — for providing a sound induction formula — they still have to be complemented by base cases, which may be analogously described by triples $(\xi,\emptyset,C)$, 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.

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

[97] Existential 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].

[98] We newly introduce this name here in the hope that it helps the readers to remember that this ratio measures how well the induction hypotheses hit the recursive calls.

[99] Well-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.

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:

$$(\mathsf{ack4}) \,/\, 1.1.2 \quad = \quad \mathsf{ack}(x, y).$$

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*

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

that is given as follows.

The first triple of the relational description, namely

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

(obtained from the equation (ack2)) is replaced with $(\ \xi_1,\ \ \{\mu_{1,1}\},\ \ \emptyset\ )$, where $\xi_1 = \{x{\mapsto}\mathsf{s}(x'),\ \ y{\mapsto}0\}$ and $\mu_{1,1} = \{x{\mapsto}x',\ \ y{\mapsto}\mathsf{s}(0)\}$. This can be seen as follows. The substitution called $\sigma$ in the above discussion — which has to match the first element of the triple to $((\mathsf{ack4})/1.1.2)\xi_1$ — has to satisfy $(\mathsf{ack}(\mathsf{s}(x), 0))\sigma = (\mathsf{ack}(x, y))\xi_1$. Taking $\xi_1$ 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 $(\mathsf{ack4})/1.1.2$ to the $\sigma$-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

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

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

The hitting ratio for the three constraints on the two arguments of $(\mathsf{ack4})/1.1.2$ is $\frac{6}{6} = 1$. This is optimal: the induction hypotheses are 100% identical to the expected recursive calls.

To achieve completeness of the substitutions $\xi_k$ 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 § 4.9. □

# 6 AUTOMATED EXPLICIT INDUCTION

## *6.1 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.

It is almost never the case that the product of the possible initial states, input threads, and schedulings of a computing system is a smaller number. Otherwise, however, even the most carefully chosen test series cannot cover the often very huge or even infinite number of possible cases; and then, no matter how many bugs have been found by testing, there can never be certainty that none remain.

Therefore, 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.[100]

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. Good candidates are the areas of central processing units (CPUs) in standard processors and of security protocols.

To reduce the costs of verification, we can hope to automate it with automated theorem-proving systems. This automation has to include mathematical induction because induction is essential for the verification of the properties of most data types used in digital design (such as natural numbers, arrays, lists, and trees), for the repetition in processing (such as loops), and for parameterized systems (such as a generic $n$-bit adder). Decision methods (many of them exploiting finiteness, e.g. the use of 32-bit data paths) allow automatic verification of some modules,

[100] See, for example, [Davis, 2009].

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.

## *6.2 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 zero[101] in the summer 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 [Boyer and Moore, 1973], which took place in Stanford (CA) in August 1973, and it is best documented in Part II of Moore's PhD thesis [1973], defended in November 1973.

The Pure LISP Theorem Prover was 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".

[101] No 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, besides interactively invoked induction in resolution theorem proving (e.g. by starting a resolution proof for the two clauses resulting from Skolemization of $(\mathsf{P}(\mathsf{0}) \wedge \neg\mathsf{P}(x)) \Rightarrow \exists y.\ (\mathsf{P}(y) \wedge \neg\mathsf{P}(\mathsf{s}(y)))$ [Darlington, 1968]), 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 $\mathsf{0}$ and $\mathsf{s}$ (cf. § 4.4) on a randomly picked, universally quantified variable of type nat.

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).

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 of 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. 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. In [Moore, 1973, p.172] 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."

Boyer and Moore report[102] 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. 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]

[102] Cf. [Wirth, 2012d].

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,[103] but, also in this aspect, the major influence on Boyer and Moore is again W. W. Bledsoe.[104]

The source code of the Pure LISP Theorem Prover was written in the programming language POP–2.[105] 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.[106] This was considered fast at the time, compared to the search-dominated proofs by resolution systems. Moore explains the speed:

> "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.[107]

[103] Cf. e.g. the quotation from [Burstall, 1969] in Note 101.

[104] Cf. e.g. [Bledsoe *et al.*, 1972].

[105] Cf. [Burstall *et al.*, 1971].

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

> "Despite theses 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."

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 (IF[108] $a$ $d$ $d$) and (CDR (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. Note that such a lazy form of evaluation is sound w.r.t. the given logic only if each eager call terminates and returns a constructor ground term, simply because all functions are meant to be defined in terms of constructor variables (cf. § 5.4).[109]

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 applicative[110] 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:

| | | | | | | |
|---|---|---|---|---|---|---|
| (CAR (CONS $a$ $d$)) | = | $a$ | | (CAR NIL) | = | NIL |
| (CDR (CONS $a$ $d$)) | = | $d$ | | (CDR NIL) | = | NIL |

As standard in LISP, every term of the form (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.[111] 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 (CONS NIL $d$).[112]

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.

### *6.2.1 Simplification in the* Pure LISP Theorem Prover

The first stage of the Boyer–Moore waterfall — "simplification" in Figure 1 — is called "normalation"[113] in the Pure LISP Theorem Prover. It applies the following simplification procedures to LISP expressions until the result does not change any more: "evaluation", "normalization", and "reduction".

[107] Note 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 § 5.2) confluence is sufficient here.
[108] Cf. Note 89.
[109] There is a work-around for projective functions as indicated in Note 93 and in [Wirth, 2009].
[110] Cf. Note 81.
[111] Cf. 2nd paragraph of Page 86 of [Moore, 1973].
[112] Cf. 2nd paragraph of Page 87 of [Moore, 1973].
[113] During 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."

*"Normalization"* tries to 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

`(IF` $c$ $p$ $n$`)`

we can simplify occurrences of $c$ in $p$ to `(CONS (CAR` $c$`) (CDR` $c$`))`, and in $n$ to `NIL`. The replacement with `(CONS (CAR` $c$`) (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`.

2. If $c$ is of the form `(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;[114] 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".[115]

*"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

[114] Actually, this ground term (i.e. a term without variables) here is always a *constructor* ground term (i.e. a term built-up exclusively from constructor function symbols) because the previously applied "evaluation" procedure has reduced any ground term to a constructor ground term, provided that the termination *proviso* is satisfied.

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

terminates because of the assumed termination of recursive function definitions (cf. § 5.5). The reason for this non-termination is the following: Because of the LISP definition style via *un*conditional 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 § 5.2, a rewrite step with the definition of `PLUS` can always be executed, whereas a rewrite step with $(+1')$ or $(+2')$ requires $x = \mathtt{0}$ 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.[116]

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 stages, symbolic evaluation must be prevented from unfolding function definitions unless the context admits us to expect an effect of simplification.[117]

Because the main function of "evaluation" — only to be found in this first one of the Boyer–Moore theorem 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.[118] [119]

---

[116] It 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.

[117] In 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].

[118] Actually, "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.

"Evaluation" provides a crucial 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 like?" 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.

### *6.2.2 Destructor Elimination in the* Pure LISP Theorem Prover

There is no such stage in the Pure LISP Theorem Prover.[120]

### *6.2.3 (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.

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 § 4.8.1.

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.

[119] The 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.

[120] See, however, Note 118 and the discussion of the Pure LISP Theorem Prover in § 6.3.2.

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: the resulting formula is equipped with a link to the deleted equation.

### *6.2.4 Generalization in the* PURE LISP THEOREM PROVER

Generalization in the PURE LISP THEOREM PROVER works as described in § 4.9. 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.[121]

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

### *6.2.5 Elimination of Irrelevance in the* PURE LISP THEOREM PROVER

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

### *6.2.6 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 § 4.8. 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,[122] the induction rule of the PURE LISP THEOREM PROVER is based solely on the information provided by "evaluation" as described in § 6.2.1.

[121] See § 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.

[122] Cf. § 5.8.

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 $\mathsf{p}$ takes over the rôle of the destructor `CDR`.

EXAMPLE 11 (Induction Rule in the Explicit Induction Proof of ($\mathsf{ack}4$)).
Let us see how the induction rule of the Pure LISP Theorem Prover proceeds w.r.t. the proof of ($\mathsf{ack}4$) that we have seen in Example 5 of § 4.9. The substitutions $\xi_1$, $\xi_2$ computed as instances for the induction conclusion in Example 10 of § 5.8 suggest an overall case analysis with a base case given by $\{x \mapsto \mathsf{0}\}$, and two step cases given by $\xi_1 = \{x \mapsto \mathsf{s}(x'),\ y \mapsto \mathsf{0}\}$ and $\xi_2 = \{x \mapsto \mathsf{s}(x'),\ y \mapsto \mathsf{s}(y')\}$. The Pure LISP Theorem Prover requires the axioms ($\mathsf{ack}1$), ($\mathsf{ack}2$), ($\mathsf{ack}3$) to be in destructor instead of constructor style:

$$\begin{array}{lll}
(\mathsf{ack}1') & \mathsf{ack}(x,y) = \mathsf{s}(y) & \Leftarrow\ x = \mathsf{0} \\
(\mathsf{ack}2') & \mathsf{ack}(x,y) = \mathsf{ack}(\mathsf{p}(x),\mathsf{s}(\mathsf{0})) & \Leftarrow\ x \neq \mathsf{0}\ \wedge\ y = \mathsf{0} \\
(\mathsf{ack}3') & \mathsf{ack}(x,y) = \mathsf{ack}(\mathsf{p}(x),\mathsf{ack}(x,\mathsf{p}(y))) & \Leftarrow\ x \neq \mathsf{0}\ \wedge\ y \neq \mathsf{0}
\end{array}$$

"Evaluation" does not rewrite the input conjecture with this definition, but writes a "fault description" for the permanent occurrences of $\mathsf{p}$ as arguments of the three occurrences of $\mathsf{ack}$ on the right-hand sides, essentially consisting of the following three "pockets": $(\mathsf{p}(x))$, $(\mathsf{p}(x),\mathsf{p}(y))$, and $(\mathsf{p}(y))$, respectively. Similarly, the pockets gained from the fault descriptions of rewriting the input conjecture with the definition of $\mathsf{lessp}$ essentially consists of the pocket $(\mathsf{p}(y),\mathsf{p}(\mathsf{ack}(x,y)))$. Similar to the non-applicability of the induction template for $\mathsf{lessp}$ in Example 9 of § 5.7, this fault description does not suggest any induction because one of the arguments of $\mathsf{p}$ 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 $\mathsf{p}$ 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 $\{x \mapsto \mathsf{0}\}$ and $\{y \mapsto \mathsf{0}\}$, and a step case given by $\{x \mapsto \mathsf{s}(x'),\ y \mapsto \mathsf{s}(y')\}$.[123] This turns the first step case of the proof of Example 5 into a base case. The Pure LISP Theorem Prover finally fails (contrary to all other Boyer–Moore theorem provers, see Examples 5, 10, and 21) with the step case it actually generates:

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

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

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

EXAMPLE 12 (Proof of (lessp7) by Explicit Induction with Merging).
Let us write $T(x, y, z)$ for (lessp7) of § 4.4. From the proof of (lessp7) in Example 3 of § 4.7 we can learn the following: The proof becomes simpler when we take $T(\mathsf{0}, \mathsf{s}(y'), \mathsf{s}(z'))$ as base case (besides say $T(x, y, \mathsf{0})$ and $T(x, \mathsf{0}, \mathsf{s}(z'))$), instead of any of $T(\mathsf{0}, y, \mathsf{s}(z'))$, $T(\mathsf{0}, \mathsf{s}(y'), z)$, $T(\mathsf{0}, y, z)$. The crucial lesson from Example 3, however, is that the step case of explicit induction has to be

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

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 all axioms to be in destructor style, so instead of (lessp1), (lessp2), (lessp3), we have to take:

$$\begin{array}{lll}
(\mathsf{lessp1}') & \mathsf{lessp}(x, y) = \mathsf{false} & \Leftarrow\ y = \mathsf{0} \\
(\mathsf{lessp2}') & \mathsf{lessp}(x, y) = \mathsf{true} & \Leftarrow\ y \neq \mathsf{0} \ \wedge\ x = \mathsf{0} \\
(\mathsf{lessp3}') & \mathsf{lessp}(x, y) = \mathsf{lessp}(\mathsf{p}(x), \mathsf{p}(y)) & \Leftarrow\ y \neq \mathsf{0} \ \wedge\ x \neq \mathsf{0}
\end{array}$$

"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 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 $((\mathsf{p}(z)))$, $((\mathsf{p}(x), \mathsf{p}(y)))$, and $((\mathsf{p}(y), \mathsf{p}(z)))$, respectively. The Pure LISP Theorem Prover merges these three fault descriptions to the single one $((\mathsf{p}(x), \mathsf{p}(y), \mathsf{p}(z)))$, and so suggests the proper step case indeed, although it suggests the base case $T(\mathsf{0}, y, z)$ instead of $T(\mathsf{0}, \mathsf{s}(y'), \mathsf{s}(z'))$, which requires some extra work, but does not result in a failure. □

### *6.2.7 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 § 6.2.6, 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.[124] In the PURE LISP THEOREM PROVER, all information on this structure that is taken into account by the induction rule comes from the fault descriptions of previous applications of "evaluation", which store only a small part of the 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 § 4.5), 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 § 6.2.

Especially the crucial link between symbolic evaluation and the induction rule of explicit induction described at the end of § 6.2.1 may be crucial for the success of the entire development of recursion analysis and explicit induction.

## *6.3* 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.[125] To the best of our knowledge, there is no similarly broad treatment of heuristics in theorem proving.

[124] There 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."

Except for ACL2, Boyer and Moore never gave names to their theorem provers.[126] 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.[127] 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,[128] 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.[129] 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. § 6.2). 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 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.[130]

[125] In [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."

[126] The 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.

[127] Cf. [Boyer, 2012].

[128] For 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].

[129] Cf. [Boyer, 2012].

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 § 5.5. 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*,[131] 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.[132] Each argument of the shell can be accessed[133] 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 recognizes[133] 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.[134] `T` and `F` now abbreviate the nullary function calls `(TRUE)` and `(FALSE)`, respectively, which are the only Boolean values. Any argument with Boolean intention besides `F` is taken to be `T` (including `NIL`).

---

[130] In 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."

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

[132] Note 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.

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

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`,
   - destructor[133] `SUB1` with default value `0`, and
   - type predicate[133] `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 ADD1 | Axioms Generated by Shell CONS |
|---|---|---|
| 0.1 | (NUMBERP X) = T ∨ (NUMBERP X) = F | (LISTP X) = T ∨ (LISTP X) = F |
| 0.2 | (NUMBERP (ADD1 X1)) = T | (LISTP (CONS X1 X2)) = T |
| 0.3 | (NUMBERP 0) = T | |
| 0.4 | (NUMBERP T) = F | (LISTP T) = F |
| 0.5 | (NUMBERP F) = F | (LISTP F) = F |
| 0.6 | | (LISTP X) = F ∨ (NUMBERP X) = F |
| 1 | (ADD1 (SUB1 X)) = X ⇐ X ≠ 0 ∧ (NUMBERP X) = T | (CONS (CAR X) (CDR X)) = X ⇐ (LISTP X) = T |
| 2 | (ADD1 X1) ≠ 0 | |
| 3 | (SUB1 (ADD1 X1)) = X1 ⇐ (NUMBERP X1) = T | (CAR (CONS X1 X2)) = X1<br>(CDR (CONS X1 X2)) = X2 |
| 4 | (SUB1 0) = 0 | |
| 5.1 | (SUB1 X) = 0 ⇐ (NUMBERP X) = F | (CAR X) = 0 ⇐ (LISTP X) = F<br>(CDR X) = 0 ⇐ (LISTP X) = F |
| 5.2 | (SUB1 (ADD1 X1)) = 0 ⇐ (NUMBERP X1) = F | |
| L1[135] | (ADD1 X) = (ADD1 0) ⇐ (NUMBERP X) = F | |
| L2[136] | (NUMBERP (SUB1 X)) = T | |

[134] There 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 `(CAR NIL) = 0`, the shell `PACK` has no base object, and `NIL` just abbreviates `(PACK (CONS 78 (CONS 73 (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 § 4.4, 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),[134] 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 § 4.5 — 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 § 5 we have already discussed the *definition principle*[137] 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 § 5.5. To this end, Thm can apply its additional axiom of the well-foundedness of the irreflexive ordering LESSP on the natural numbers,[138] and the theorem of the well-foundedness of the lexicographic combination of two well-founded orderings.

Just as in § 6.2, we will now 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.

[135] Proof 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.

[136] Proof 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).

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

### *6.3.1 Simplification in* THM

We discussed simplification in the PURE LISP THEOREM PROVER in § 6.2.1. 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.[139] 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 § 5.2)*
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:

```
(FLATTEN X) = (IF (LISTP X)
                  (APPEND (FLATTEN (CAR X)) (FLATTEN (CDR X)))
                  (CONS X NIL))
```
□

[138] See 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 42.

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 § 4.1 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 42), it is not possible to replace the reflexive and transitive closure of the successor relation `S` with the THM-definable predicate $\{\ \mathtt{Y} \mid \mathtt{(NUMBERP\ Y)} = \mathtt{T} \ \wedge\ (\mathtt{(LESSP\ Y\ X)} = \mathtt{T} \ \vee\ \mathtt{Y} = \mathtt{X})\ \}$, because (by the THM-analog of axiom (lessp2′) of Example 12 in § 6.2.6) 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`.

[139] See 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.[140]

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 $(t \neq \texttt{F})$. The free universal variables of a clause have to be treated as constants during contextual rewriting.[141]

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,[142] it is moved to the top position (outside-in), by replacing each case with itself in the `IF`'s context, such that the literal $C[\texttt{(IF}\ t_0\ t_1\ t_2\texttt{)}]$ is intermediately replaced with $\texttt{(IF}\ t_0\ C[t_1]\ C[t_2]\texttt{)}$, and then this literal splits its clause in two: one with the two literals $\texttt{(NOT}\ t_0\texttt{)}$ and $C[t_1]$ in place of the old one, and one with $t_0$ and $C[t_2]$ instead.

THM eagerly removes variables in solved form: If the variable `X` does not occur in the term $t$, but the literal $(\texttt{X} \neq t)$ 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

[140] Cf. § 6.2.1.

[141] This 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.

[142] These rewrite rules whose soundness depends on termination are `(IF X Y Y) = Y`; `(IF X X F) = X`; and for Boolean `X`: `(IF X T F) = X`; tested for applicability in the given order.

universally quantified and so $(\mathtt{X} \neq t)$ must be false because it implies $(t \neq t)$. Alternatively, the removal can be seen as a resolution step with the axiom of reflexivity.

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 $P(\mathsf{s}(x))) \Leftarrow P(x)$, THM uses "destructors in the hypothesis", such as $(P(x) \Leftarrow P(\mathsf{p}(x))) \Leftarrow x \neq 0$. 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. 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
   $$C[\mathsf{lessp}(x, \mathsf{s}(y))]$$
   will be expanded by ($\mathsf{lessp2}'$), ($\mathsf{lessp3}'$), and ($\mathsf{p1}$) into the clauses
   $$x \neq 0, \;\; C[\mathsf{true}]$$
   and
   $$x = 0, \;\; C[\mathsf{lessp}(\mathsf{p}(x), y)]$$
   — even if $\mathsf{p}(x)$ is a newly occurring subterm! — because the second argument position of lessp is such a set of measured positions according to Example 18 of § 6.3.7.[143]

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

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 non-exclusive 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 limited framework of very abstract control), 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 then 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.[144] 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.[145] 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.[146]

Some collections of rewrite lemmas can cause THM's rewriter not to terminate.[147] For permutative rules like commutativity, however, termination is assured by simple term ordering heuristics.[148]

[144] Matt Kaufmann and J Strother Moore added support for "forcing" and "case split" annotations to ACL2 in the mid-1990s.

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

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

[147] Non-termination of rewriting caused the Boyer–Moore theorem provers to run forever or exhaust the LISP stack or heap — except ACL2, which maintains its own user-adjustable stack size and gives a coherent error on stack overflow without crashing the LISP system. NQTHM introduced special tools to track down the rewriting process via the rewrite call stack (namely `BREAK-REWRITE`, after setting `(MAINTAIN-REWRITE-PATH T)`) and to count the applications of a rewrite rule (namely `ACCUMULATED-PERSISTENCE`), so the problematic rules can easily be detected and the user can disable them. See § 12 of [Boyer and Moore, 1988b; 1998] for the details.

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

### *6.3.2 Destructor Elimination in* THM

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

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 ($\mathsf{ack3}$) of § 4.4:

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

Introduce (delete) the solved variables $x'$ and $y'$ for the constructor terms $\mathsf{s}(x)$ and $\mathsf{s}(y)$ occurring on the left-hand side, respectively, and add (delete) two further conditions by applying the definition ($\mathsf{p1}'$) (cf. § 4.4) twice.

$$\mathsf{ack}(\mathsf{s}(x),\mathsf{s}(y)) = \mathsf{ack}(x,\mathsf{ack}(\mathsf{s}(x),y)) \;\Leftarrow\; \left(\begin{array}{lllll} & x' = \mathsf{s}(x) & \wedge & \mathsf{p}(x') = x \\ \wedge & y' = \mathsf{s}(y) & \wedge & \mathsf{p}(y') = y \end{array}\right).$$

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

$$\mathsf{ack}(x',y') = \mathsf{ack}(x,\mathsf{ack}(x',y)) \;\Leftarrow\; \left(\begin{array}{lllll} & x' = \mathsf{s}(x) & \wedge & \mathsf{p}(x') = x \\ \wedge & y' = \mathsf{s}(y) & \wedge & \mathsf{p}(y') = y \end{array}\right).$$

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

$$\mathsf{ack}(x',y') = \mathsf{ack}(\mathsf{p}(x'),\mathsf{ack}(x',\mathsf{p}(y'))) \;\Leftarrow\; \left(\begin{array}{lllll} & x' = \mathsf{s}(x) & \wedge & \mathsf{p}(x') = x \\ \wedge & y' = \mathsf{s}(y) & \wedge & \mathsf{p}(y') = y \end{array}\right).$$

Add (Delete) two conditions by applying axiom ($\mathsf{nat2}$) twice.

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

Delete (Introduce) the leftmost equations of the condition by applying lemma ($\mathsf{s1}'$) (cf. § 4.4) twice, and delete (introduce) the solved variables $x$ and $y$ for the destructor terms $\mathsf{p}(x')$ and $\mathsf{p}(y')$ occurring in the left-hand side of the equation in the conclusion, respectively.

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

Up to renaming of the variables, this is the destructor-style rule ($\mathsf{ack3}'$) of Example 11 (cf. § 6.2.6). □

Our data types are defined inductively over constructors.[149] 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 applications.

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 `(CONS X Y)` and a subterm of it, whereas `(CDR Z)` is a sub-structure of `Z` in case of `(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.[150]

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

$$\big(\ P(x) \ \Leftarrow\ P(\mathsf{p}(x))\ \big) \ \Leftarrow\ x \neq \mathsf{0}.$$

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 $\mathsf{delfirst}$ defined in § 4.5. To verify the correctness of a deletion-sort algorithm based on $\mathsf{delfirst}$, a useful step case for an induction proof is of the form[151]

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

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. □

[149] Here 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 $\mathsf{car}$, $\mathsf{cdr}$ of the list of natural numbers $\mathsf{list}(\mathsf{nat})$ of § 4.5, we co-inductively obtain the data type of infinite streams of natural numbers.

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

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

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 all the terms at the positions of the proper destructor arguments are removed.

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 $\mathsf{lessp}(r, y) = \mathsf{true}$. The related generalized destructors are the quotient $\mathsf{div}(x, y)$ of $x$ by $y$, and its remainder $\mathsf{rem}(x, y)$. 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 § 4.4 play in Example 15 (and which the definitions (car1′), (cdr1′) and the lemma (cons1′) of § 4.5 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 $\lambda q, r.\ ((q * y) + r)$:

$$(\mathsf{div1}') \quad \mathsf{div}(x, y) = q \Leftarrow y \neq 0 \wedge (q * y) + r = x \wedge \mathsf{lessp}(r, y) = \mathsf{true}$$
$$(\mathsf{rem1}') \quad \mathsf{rem}(x, y) = r \Leftarrow y \neq 0 \wedge (q * y) + r = x \wedge \mathsf{lessp}(r, y) = \mathsf{true}$$
$$(+9') \quad (q * y) + r = x \Leftarrow y \neq 0 \wedge q = \mathsf{div}(x, y) \wedge r = \mathsf{rem}(x, y)$$

If we have a clause with the literal $y = 0$, in which the destructor terms $\mathsf{div}(x, y)$ or $\mathsf{rem}(x, y)$ occur, we can — just as in the of Example 15 (reverse direction) — introduce the new literals $\mathsf{div}(x, y) \neq q$ and $\mathsf{rem}(x, y) \neq r$ 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 $\mathsf{div}(x, y)$, $\mathsf{rem}(x, y)$, and $x$ are gone.[152] □

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 $y_0, \ldots, y_n$, 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 $(y_0 \neq t_0^{\mathrm{d}})$, $\ldots$, $(y_n \neq t_n^{\mathrm{d}})$, for distinct (generalized) destructor terms $t_0^{\mathrm{d}}, \ldots, t_n^{\mathrm{d}}$.[153]

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 $t_0^{\mathrm{d}}, \ldots, t_n^{\mathrm{d}}$ occur in the given clause as subterms, then rewrite all their occurrences with $(y_0 \neq t_0^{\mathrm{d}})$, $\ldots$, $(y_n \neq t_n^{\mathrm{d}})$ from right to left and then use the first literal of the elimination lemma from right to left for further normalization.[154]

[152] For 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].

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

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

by the elimination lemma (s1) into the clause

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

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

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

and another one.[155]

Looping could result from eliminating the destructor introduced by simplification (such as it is actually the case for our destructor 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.[156]

---

[153] Thm 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 $y_0, \ldots, y_n$.

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 $t_i^{\mathrm{d}}$, and then removing the literal $(y_i \neq t_i^{\mathrm{d}})$. Thus, Thm would accept only the non-flattened versions of our elimination lemmas, such as (s1) instead of (s1′) (cf. § 4.4), and such as (cons1) instead of (cons1′) (cf. § 4.5).

[154] If 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.

[155] The latter step is given in more detail in the context of the second of the two less important criteria of § 6.3.1 for unblocking the unfolding of $\mathsf{lessp}(x, \mathsf{s}(y))$.

[156] See 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].

### *6.3.3 (Cross-) Fertilization in* THM

This stage has already been described in § 6.2.3. There is no noticeable difference between the PURE LISP THEOREM PROVER and THM here, besides some heuristic fine tuning.[157]

### *6.3.4 Generalization in* THM

THM adds only one new rule to the universally applicable heuristic rules for generalization on a term $t$ mentioned in § 4.9:

> "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.[158]

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 (say the bit expressing `NUMBERP`), then, for the respective type predicate, 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 $T[t]$.[159] In general, for the addition of such a literal `(NOT` $t'$`)`, a proper subterm $t'$ of a generalization lemma must match $t$.[160]

[157] See Page 149 of [Boyer and Moore, 1979].
[158] See Page 156f. of [Boyer and Moore, 1979].
[159] Cf. Note 121.
[160] Moreover, 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.

### *6.3.5 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.[161]

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 constructor functions (i.e. shells and base objects) and functions with explicit definitions, 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,[162] the justification for this heuristic test is the following: The bit-vector of this literal must contain the singleton type of `F` (containing only the term `F`, cf. § 6.3.1); 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.

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

[162] This 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. Moreover, users sometimes supply such predicates in order to suggest a particular induction ordering. For example, if we want to supply the function `sqrtio` of § 6.3.9 to Thm, then we have to provide a complete definition, typically given by setting `sqrtio` to be `T` in all other cases. Luckily, such nonsense functions will typically not occur in any proof.

### *6.3.6 Induction in* Thm *as compared to the* Pure LISP Theorem Prover

As we have seen in § 6.2.6, 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[163] 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.[164]

### *6.3.7 Induction Templates generated by Definition-Time Recursion Analysis*

The first part of recursion analysis in Thm consists of 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 — to be able to generate a plenitude of sound sets of step formulas later — 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. This information will later be used to guarantee the soundness and improve 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 § 5.5.[165] 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 § 5.5, but now for the destructor-style definitions of lessp and ack, because only destructor-style definitions are admissible in Thm.

[163] See the discussion in § 6.2.7 on Example 11 from § 6.2.6.

[164] Note 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 § 4.8.1.

EXAMPLE 18 (Two Induction Templates with different Measured Positions).
For the ordering predicate $\mathsf{lessp}$ as defined by ($\mathsf{lessp}1'$–$3'$) in Example 12 of § 6.2.6, we get two induction templates with the sets of measured positions $\{1\}$ and $\{2\}$, respectively, both for the well-founded ordering $\lambda x, y.\ (\mathsf{lessp}(x, y) = \mathsf{true})$. The first template has the weight term (1) and the relational description

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

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

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

EXAMPLE 19 (One Induction Template with Two Measured Positions).
For the Ackermann function $\mathsf{ack}$ as defined by ($\mathsf{ack}1'$–$3'$) in Example 11 of § 6.2.6, we get only one appropriate induction template. The set of its measured positions is $\{1, 2\}$, because of the weight function $\mathsf{cons}((1), \mathsf{cons}((2), \mathsf{nil}))$ (in Thm actually: `(CONS x y)`) in the well-founded lexicographic ordering

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

The relational description has two elements: For the equation ($\mathsf{ack}2'$) we get

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

and for the equation ($\mathsf{ack}3'$) we get

$$(\ \mathsf{ack}(x, y),\ \ \{\mathsf{ack}(x, \mathsf{p}(y)),\ \mathsf{ack}(\mathsf{p}(x), \mathsf{ack}(x, \mathsf{p}(y)))\},\ \ \{x \neq \mathsf{0},\ \ y \neq \mathsf{0}\}\ ).$$ □

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

`(LESSP (COUNT (SUB1 X)) (COUNT X))` ⇐ `(NOT (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 $\mathsf{count}$ from § 4.5.[166]

[165] Those 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.

### *6.3.8 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 § 5.8. 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 § 4.4. 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 $\mathsf{ack}(x, y)$. □

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) $\quad \mathsf{lessp}(y, \mathsf{ack}(x, y)) = \mathsf{true}$
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

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

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

This can be seen as follows: The substitution called $\xi$ in the discussion of § 5.8 can be chosen to be the identity substitution ${}_{\{x,y\}}\restriction\mathrm{id}$ 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 $\sigma$ in § 5.8 (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 $\sigma = \{x \mapsto y,\ \ y \mapsto \mathsf{ack}(x, y)\}$. So the constraints for $\mu_1$ (which tries to match (ack4)/1.1 to the $\sigma$-instance of the second element of the triple) are: $y\mu_1 = \mathsf{p}(y)$ for the first (measured) position of lessp; and $\mathsf{ack}(x, y)\mu_1 = \mathsf{p}(\mathsf{ack}(x, y))$ 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 = $y$.

[166] For 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

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

(generated by the equation (ack2′)) is replaced with $\big(\ {}_{\{x,y\}}\!\restriction\!\mathrm{id},\ \ \{\mu'_{1,1}\},\ \ \{x\neq\mathsf{0}\}\ \big)$, where $\mu'_{1,1} = \{x{\mapsto}\mathsf{p}(x),\ \ y{\mapsto}\mathsf{s}(0)\}$. The triple

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

(generated by (ack3′)) is replaced with $\big(\ {}_{\{x,y\}}\!\restriction\!\mathrm{id},\ \ \{\mu'_{2,1},\ \ \mu'_{2,2}\},\ \ \{x\neq\mathsf{0},\ \ y\neq\mathsf{0}\}\ \big)$, where $\mu'_{2,1} = \{x{\mapsto}x,\ \ y{\mapsto}\mathsf{p}(y)\}$, and $\mu'_{2,2} = \{x{\mapsto}\mathsf{p}(x),\ \ y{\mapsto}\mathsf{ack}(x,\mathsf{p}(y))\}$.

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

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

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 § 5.8 and Example 21), for each step case $(\mathrm{id}, \{\ \mu_j \mid j\in J\ \}, C)$ mapped to $(\mathrm{id}, \{\ \mu'_j \mid j\in J{\uplus}J'\ \}, C')$, we have $C \subseteq C'$, and the set of substitutions $\{\ \mu_j \mid j\in J\ \}$ can be injectively[167] mapped to $\{\ \mu'_j \mid j\in J\uplus J'\ \}$ (w.l.o.g. say $\mu_i$ to $\mu'_i$ for $i\in J$), such that, for each $j\in J$ and $x\in\mathrm{dom}(\mu_j)$: $\ x\in\mathrm{dom}(\mu'_j);\ \ x\mu_j{=}x$ 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 $\{y\neq\mathsf{0}\}$ is a subset of the case condition $\{x\neq\mathsf{0},\ y\neq\mathsf{0}\}$, 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 $\frac{3}{2}$. □

[167] From 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. § 6.3.9.) 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 § 6.3.9, only practical testing of the heuristics is what matters here. See also Note 168.

In Example 12 of § 6.2.6 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 $\mu_1$ and $\mu_2$ are [*non-trivially*] *mergeable* if $x\mu_1 = x\mu_2$ for each $x \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ [and there is a $y \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ with $y\mu_1 \neq y$].

Two triples $({}_{V_1}\restriction\mathrm{id}, A_1, C_1)$ and $({}_{V_2}\restriction\mathrm{id}, A_2, C_2)$ of two step-case descriptions of two induction schemes, each with domain $V_k = \mathrm{dom}(\mu_k)$ 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 $\mu_1$ and $\mu_2$ are [non-trivially] mergeable. The result of their merging is $\big(\ {}_{V_1 \cup V_2}\restriction\mathrm{id},\ \ m(A_1, A_2),\ \ C_1 \cup C_2\ \big)$, 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 $\mu_1$ and $\mu_2$ are mergeable as well as all substitutions ${}_{V_1 \setminus V_2}\restriction\mathrm{id} \cup \mu_2$ with $\mu_2 \in A_2$ for which there is no substitution $\mu_1 \in A_1$ such that $\mu_1$ and $\mu_2$ are mergeable.

Two induction schemes are *mergeable* if the step-case description of the first induction scheme can be injectively[168] 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.[169]

| | pos. set | ind. var.s | step-case description | hit. ratio |
|---|---|---|---|---|
| 1 | $\{1\}$ | $\{x\}$ | $\{\ (\ {}_{\{x,z\}}\restriction\mathrm{id},\ \ \{\mu_1\},\ \ \{x \neq 0\}\ )\ \}$ | 1 |
| 2 | $\{2\}$ | $\{x\}$ | $\{\ (\ {}_{\{x,y\}}\restriction\mathrm{id},\ \ \{\mu_2\},\ \ \{x \neq 0\}\ )\ \}$ | 1 |
| 3 | $\{2\}$ | $\{y\}$ | $\{\ (\ {}_{\{x,y\}}\restriction\mathrm{id},\ \ \{\mu_2\},\ \ \{y \neq 0\}\ )\ \}$ | 1 |
| 4 | $\{3\}$ | $\{y\}$ | $\{\ (\ {}_{\{y,z\}}\restriction\mathrm{id},\ \ \{\mu_3\},\ \ \{y \neq 0\}\ )\ \}$ | 1 |
| 5 | $\{3\}$ | $\{z\}$ | $\{\ (\ {}_{\{y,z\}}\restriction\mathrm{id},\ \ \{\mu_3\},\ \ \{z \neq 0\}\ )\ \}$ | 1 |
| 6 | $\{2\}$ | $\{x, y\}$ | $\{\ (\ {}_{\{x,y\}}\restriction\mathrm{id},\ \ \{\mu_2\},\ \ \{x \neq 0,\ \ y \neq 0\}\ )\ \}$ | 2 |
| 7 | $\{3\}$ | $\{y, z\}$ | $\{\ (\ {}_{\{y,z\}}\restriction\mathrm{id},\ \ \{\mu_3\},\ \ \{y \neq 0,\ \ z \neq 0\}\ )\ \}$ | 2 |
| 8 | $\{2, 3\}$ | $\{x, y, z\}$ | $\{\ (\ {}_{\{x,y,z\}}\restriction\mathrm{id},\ \ \{\mu_4\},\ \ \{x \neq 0,\ \ y \neq 0,\ \ z \neq 0\}\ )\ \}$ | 4 |
| 9 | $\{1, 2, 3\}$ | $\{x, y, z\}$ | $\{\ (\ {}_{\{x,y,z\}}\restriction\mathrm{id},\ \ \{\mu_4\},\ \ \{x \neq 0,\ \ y \neq 0,\ \ z \neq 0\}\ )\ \}$ | 5 |

$\mu_1 = \{x \mapsto \mathsf{p}(x),\ \ z \mapsto \mathsf{p}(z)\}$, $\mu_2 = \{x \mapsto \mathsf{p}(x),\ \ y \mapsto \mathsf{p}(y)\}$,
$\mu_3 = \{y \mapsto \mathsf{p}(y),\ \ z \mapsto \mathsf{p}(z)\}$, and $\mu_4 = \{x \mapsto \mathsf{p}(x),\ \ y \mapsto \mathsf{p}(y),\ \ z \mapsto \mathsf{p}(z)\}$.
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 $p = \mathsf{true}$ 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

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

Then the Boyer–Moore waterfall sends this clause through three rounds of reduction between destructor elimination and simplification as discussed at the end of § 6.3.2, 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".[170]

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.[171] □

[168] From a logical viewpoint, it is again not clear why an 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 § 6.3.9, only practical testing of the heuristics is what matters here. See also Note 167.

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

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

[171] The 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.

### *6.3.9 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 requirements[172] 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.[173]

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

[172] Cf. Notes 167 and 168.

[173] While 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]: Heuristics cannot be "bugged" or "have serious flaws", unless this would mean that they turn out to be inferior to others w.r.t. a standard corpus. A "rational reconstruction" or a "meta-theoretic analysis" may help to guess even superior heuristics, but they may not have any epistemological value *per se*.

$$
\begin{aligned}
x &= \overline{AE} \\
y &= \overline{FE} = \overline{BE} = \overline{AF} \\
\mathsf{div}(x, \mathsf{s}(\mathsf{s}(0))) &= \overline{AC} = \overline{CF} \\
x - y &= \overline{AB} = \overline{BD} = \overline{BG} = \overline{GF} \\
\mathsf{s}(\mathsf{s}(0)) * (x - y) &= \overline{AD} \\
(\mathsf{s}(\mathsf{s}(0)) * y) - x &= \overline{AG} = \overline{GD}
\end{aligned}
$$

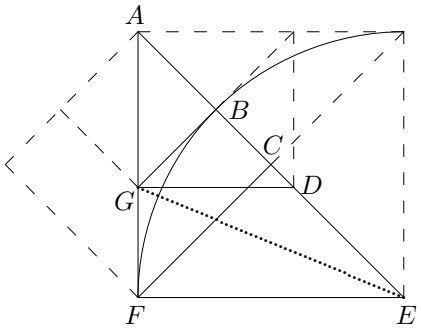


Figure 4. Four possibilities to descend with rational representations of $\sqrt{2}$: From the triangle with right angle at $F$ to those at $C$, $G$, or $B$.

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 of 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 an advanced 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:

$$
\begin{aligned}
(\mathsf{sqrtio1}) \quad & \mathsf{sqrtio}(x, y) \\
& = \mathsf{and}(\mathsf{sqrtio}(y, \mathsf{div}(x, \mathsf{s}(\mathsf{s}(0)))), \\
& \quad \mathsf{and}(\mathsf{sqrtio}(\mathsf{s}(\mathsf{s}(0)) * (x - y), (\mathsf{s}(\mathsf{s}(0)) * y) - x), \\
& \qquad \mathsf{sqrtio}((\mathsf{s}(\mathsf{s}(0)) * y) - x, x - y))) \\
& \qquad\qquad \Leftarrow \quad x * x = \mathsf{s}(\mathsf{s}(0)) * y * y \ \wedge \ y \neq 0
\end{aligned}
$$

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.

## *6.4* 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 $\varepsilon_0$, i.e. up to $\omega^{\omega^{\cdot^{\cdot^{\cdot}}}}$, 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 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-arithmetic[174] 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).

[174] Linear 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,[175] 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,[176] and the verification of the Berkeley C String Library.[177] Many more examples are listed in [Boyer and Moore, 1998].

## *6.5* 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.

---

[175] Cf. [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."

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

[177] Via 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."[178]

## *6.6 Explicit Induction in* RRL*,* INKA*, and* OYSTER/CLAM

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) development of rippling (in parallel to a similar development of rippling by Alan Bundy, cf. § 7.2), 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 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.

[178] For the complete text of the citation of Boyer, Moore, and Kaufmann see `http://awards.acm.org/citation.cfm?id=4797627&aw=149`.

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),[179] and the development of INKA was discontinued for this reason in the year 2000.

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. § 6.2). 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. § 7.2).

The OYSTER/CLAM system was developed at the University of Edinburgh in the late 1980s[180] and the 1990s by a large team led by Alan Bundy.[181] OYSTER is a reimplementation 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 § 4.6. 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. CLAM is a proof planner (cf. § 7.1) which guides OYSTER, based on proof search in the meta-language, which includes rippling (cf. § 7.2).

OYSTER/CLAM proved the viability of many concepts, but it is the slowest system explicitly mentioned in this article,[179] partly because of its constructive logic.

[179] This 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/CLAM (Alan Bundy), and a first prototype of QUODLIBET (Ulrich Kühler). Only OYSTER/CLAM 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.

[180] The system description [Bundy *et al.*, 1990] of OYSTER/CLAM 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].

[181] For Alan Bundy see also Note 7.

Besides approaches in its line of development more or less addressing theorem proving in general, such as rippling (cf. § 7.2) 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].

## 7 ALTERNATIVE APPROACHES BESIDES EXPLICIT INDUCTION

In this section we will discuss the approaches to the automation of mathematical induction that do not strictly follow the method of explicit induction as we have described it. In general, these approaches are not disjoint from explicit induction. To the contrary, *proof planning* and *rippling* have until now been applied mostly to systems more or less based on explicit induction, but they are not exclusively related to induction and they are not following Boyer–Moore's method of explicit induction in every detail. Even systems for *implicit induction* may include many features of explicit induction and some of them actually do, such as RRL (cf. § 6.6) and QUODLIBET (cf. § 7.4).

### *7.1 Proof Planning*

Suggestions on how to overcome an envisioned dead end in automated theorem proving were summarized in the end of the 1980s under the keyword *proof planning*. Besides its human-science aspects,[182] the main idea[183] of proof planning is to extend a theorem-proving system — on top of the *low-level search space* of the logic calculus of a proof checker — with a *higher-level search space*, which is typically smaller or better organized w.r.t. searching, more abstract, and more human-oriented.

The extensive and sophisticated subject of proof planning is not especially related to induction, but addresses automated theorem proving in general. We cannot cover it here and have to refer the reader to the standard publications on the subject.[184]

[182] Cf. [Bundy, 1989].

[183] Cf. [Bundy, 1988], [Dennis *et al.*, 2005].

[184] In addition to [Bundy, 1989; 1988] and [Dennis *et al.*, 2005], see also [Dietrich, 2011], [Melis *et al.*, 2008], [Jamnik *et al.*, 2003], and the references there.

## *7.2 Rippling*

*Rippling* is a technique for augmenting rewrite rules with information that helps to find a way to rewrite one expression (*goal*) into another (*target*), more precisely to reduce the difference between the goal and the target by rewriting the goal.

Although rippling is not restricted to inductive theorem proving, it was first used by Aubin [1976] in the context of the description of heuristics for the automation of mathematical induction[185] and found most of its applications there.

We have already mentioned rippling in § 6.6 several times, but this huge and well-documented area of research cannot be covered here, and we have to refer the reader to the monograph [Bundy *et al.*, 2005].[186]

Let us explain here, however, why rippling can be most helpful in the automation of simple inductive proofs.

Roughly speaking, the remarkable success in proving *simple* theorems by induction automatically, can be explained as follows: If we look upon the task of proving a theorem as reducing it to a tautology, then we have more heuristic guidance when we know that we probably have to do it by mathematical induction: Tautologies can have arbitrary subformulas, but the induction hypothesis we are going to apply can restrict the search space tremendously.

In a cartoon of Alan Bundy's, the original theorem is pictured as a zigzagged mountainscape and the reduced theorem after the unfolding of recursive operators according to recursion analysis (goal) is pictured as the reflection of the mountainscape on the surface of a lake with ripples. To apply the induction hypothesis (target), instead of the uninformed search for an arbitrary tautology, we have to *get rid of the ripples* to be able to apply an instance of the theorem as induction hypothesis to the mountainscape mirrored by the calmed surface of the lake.

A crucial advantage of rippling in the area of automated induction is that it can also be used to suggest missing lemmas as described in [Ireland and Bundy, 1994].

Until today, rippling was applied to the automation of induction only within explicit induction, whereas it is clearly not limited to explicit induction, and we actually expect it to be more useful in areas of automated theorem proving with bigger search spaces and, in particular, in *descente infinie*.

[185] The verb "to ripple up" is used in §§ 3.2 and 3.4 of [Aubin, 1976] — not as a technical term, but just as an informal term for motivating some heuristics. The formalizers of rippling give explicit credit to Aubin [1976] for their inspiration in [Bundy *et al.*, 2005, § 1.10, p. 21], although Aubin does not mention the term at any other place in his publications [Aubin, 1976; 1979]. Note, however, that instead of today's name "rippling out", Aubin actually used "rippling up".

[186] Historically important are also the following publications on rippling: [Hutter, 1990], [Bundy *et al.*, 1991], [Ireland and Bundy, 1994], [Basin and Walsh, 1996].

### *7.3 Implicit Induction*

The further approaches to mechanize mathematical induction *not* subsumed by explicit induction, however, are united under the name "implicit induction".

Triggered[187] by the success of Boyer and Moore [1979], publication on these alternative approaches started already in the year 1980 in purely equational theories.[188] A sequence of papers on technical improvements[189] was topped by [Bachmair, 1988], which gave rise to a hope to develop the method into practical usefulness, although it was still restricted to purely equational theories. Inspired by this paper, in the late 1980s and the first half of the 1990s several researchers tried to understand more clearly what implicit induction means from a theoretical point of view and whether it could be useful in practice.[190]

While it is generally accepted that [Bachmair, 1988] is about implicit induction and [Boyer and Moore, 1979] is about explicit induction, there are the following three different viewpoints on what the essential aspect of implicit induction actually is.

**Proof by Consistency:**[191] Systems for proof by consistency run some Knuth–Bendix[192] or superposition[193] completion procedure which produces a huge number of irrelevant inferences under which the ones relevant for establishing the induction steps can hardly be made explicit. A proof attempt is successful when the prover has drawn all necessary inferences and stops without having detected any inconsistency.

[187] Although it is obvious that in the relatively small community of artificial intelligence and computer science in the 1970s, the success of [Boyer and Moore, 1979] triggered the publication of papers on induction in the term rewriting community, we can document the influence of Boyer and Moore's work here only with the following facts: [Boyer and Moore, 1975; 1979] are both cited in [Huet and Hullot, 1980]. [Boyer and Moore, 1977b] is cited in [Musser, 1980] as one of the "important sources of inspiration". Moreover, Lankford [1980] constitutively refers to a personal communication with Robert S. Boyer in 1979. Finally, Goguen [1980] avoids a direct reference to Boyer and Moore, but cites only the PhD thesis [Aubin, 1976] of Raymond Aubin, following their work in Edinburgh.

[188] Cf. [Goguen, 1980], [Huet and Hullot, 1980], [Lankford, 1980], [Musser, 1980].

[189] Cf. [Göbel, 1985], [Jouannaud and Kounalis, 1986], [Fribourg, 1986], [Küchlin, 1989].

[190] Cf. e.g. [Zhang *et al.*, 1988], [Kapur and Zhang, 1989], [Bevers and Lewi, 1990], [Reddy, 1990], [Gramlich and Lindner, 1991], [Ganzinger and Stuber, 1992], [Bouhoula and Rusinowitch, 1995], [Padawitz, 1996].

[191] The name "proof by consistency" was coined in the title of [Kapur and Musser, 1987], which is the later published forerunner of its outstanding improved version [Kapur and Musser, 1986].

[192] See UNICOM [Gramlich and Lindner, 1991] for such a system, following [Bachmair, 1988] with several improvements. See [Knuth and Bendix, 1970] for the Knuth–Bendix completion procedure.

[193] See [Ganzinger and Stuber, 1992] for such a system.

Proof by consistency has shown to perform far worse than any other known form of mechanizing mathematical induction; mainly because it requires the generation of far too many superfluous inferences, and because its runs are typically infinite, and its admissibility conditions are too restrictive for most applications. Roughly speaking, the conceptual flaw in proof by consistency is that, instead of finding a sufficient set of reasonable inferences, the research follows the idea of ruling out as many irrelevant inferences as possible.

**Implicit Induction Ordering:** In the early implicit-induction systems,[194] induction proceeds over a syntactical term ordering, which typically cannot be made explicit in the sense that there would be some predicate term in the logical syntax that denotes this ordering in the intended models of the specification. The semantical orderings of explicit induction, however, cannot depend on the precise syntactical term structure of a weight $w$, but only on the value of $w$ under an evaluation in the intended models.
Contrary to rudimentary inference systems that turned out to be useless in practice (such as the one of [Bachmair, 1988] for inductive completion in unconditional specifications), more powerful human-oriented inference systems (such as the one of QuodLibet) are considerably restrained by the constraint to be sound also for induction orderings that depend on the precise syntactical structure of terms (beyond their values).[195]
The early implicit-induction systems needed such sophisticated term orderings,[196] because they started from the induction conclusion and every inference step reduced the formulas w.r.t. the induction ordering again and again, but an application of an induction hypothesis was admissible to greater formulas only. This deterioration of the ordering information with every inference step was overcome by the introduction of explicit weight terms in [Wirth and Becker, 1995], which obviate the former need for syntactical term orderings as induction orderings.

**Descente Infinie ("Lazy Induction"):** Contrary to explicit induction, where induction is introduced into an otherwise merely deductive inference system only by the explicit application of induction axioms in the induction rule, the cyclic arguments and their well-foundedness in implicit induction need not be confined to single inference steps.[197] The induction rule of explicit induction generates all induction hypotheses in a single inference step. To the

[194] See [Gramlich and Lindner, 1991] and [Ganzinger and Stuber, 1992] for such systems.

[195] This soundness constraint, which was still observed in [Wirth, 1997], was dropped during the further development of QuodLibet in [Kühler, 2000], because it turned out to be unintuitive and superfluous.

[196] Cf. e.g. [Bachmair, 1988], [Steinbach, 1988; 1995], [Geser, 1996].

[197] For this reason, the funny name "inductionless induction" was originally coined for implicit induction in the titles of [Lankford, 1980; 1981] as a short form for "induction without induction rule". See also the title of [Goguen, 1980] for a similar phrase.

contrary, in implicit induction, the inference system "knows" what an induction hypothesis is, i.e. it includes inference rules that provide or apply induction hypotheses, given that certain ordering conditions resulting from these applications can be met by an induction ordering. Because this aspect of implicit induction can facilitate the human-oriented induction method described in § 4.6, the name *descente infinie* was coined for it (cf. § 4.7). Researchers introduced to this aspect by [Protzen, 1994] (entitled "Lazy Generation of Induction Hypotheses") sometimes speak of "lazy induction" instead of *descente infinie*.

The entire handbook article [Comon, 2001] (with corrections in [Wirth, 2005a]) is dedicated to the two aspects of *proof by consistency* and *implicit induction orderings*. Today, however, the interest in these two aspects tends to be historical or theoretical, especially because these aspects can hardly be combined with explicit induction.

To the contrary, *descente infinie* synergetically combines with explicit induction, as witnessed by the QUODLIBET system, which we will discuss in § 7.4.

## 7.4 QUODLIBET

In the last years of the Collaborative Research Center SFB 314 "Artificial Intelligence" (cf. § 6.6), after extensive experiments with several inductive theorem proving systems,[198] such as the explicit-induction systems NQTHM (cf. § 6.4) and INKA (cf. § 6.6), the implicit-induction system UNICOM [Gramlich and Lindner, 1991], and the mixed system RRL (cf. § 6.6), Claus-Peter Wirth (*1963) and Ulrich Kühler (*1964) came to the conclusion that — in spite of the excellent interaction concept of UNICOM[199] — *descente infinie* was actually the only aspect of implicit induction that deserved further investigation. Moreover, the coding of recursive functions in *unconditional* equations in UNICOM turned out to be most inadequate for inductive theorem proving in practice, where positive/negative-conditional equations were in demand for specification, as well as clausal logic for theorem proving.[200]

Therefore, a new system had to be created, which was given the name QUODLIBET (Latin for "as you like it"), because it should enable its users to avoid overspecification by admitting partial function specifications, and to execute proofs whose crucial proof steps mirror exactly the intended ones.[201]

[198] Cf. [Kühler, 1991].

[199] For the assessment of UNICOM's interaction concept see [Kühler, 1991, p. 134ff.].

[200] See [Kühler, 1991, pp. 134, 138].

A concept for partial function specification instead of the totality requirement of explicit induction was easily obtained by elaborating the first part of [Wirth, 1991] into the framework for positive/negative-conditional rewrite systems of [Wirth and Gramlich, 1994a]. After inventing constructor variables in [Wirth *et al.*, 1993], the monotonicity of validity w.r.t. consistent extension of the partial specifications was easily achieved [Wirth and Gramlich, 1994b], so that the induction proofs did not have to be re-done after such an extension of a partially defined function.

Although the efficiently decidable confluence criterion that defines admissibility of function definitions in QuodLibet and guarantees their (object-level) consistency (cf. § 5.2) was very hard to prove and was presented completely and in an appropriate form not before [Wirth, 2009], the essential admissibility requirements were already clear in 1996.[202]

The weak admissibility conditions of QuodLibet — mutually recursive functions, possibly partially defined because of missing cases or non-termination — are of practical importance. Although humans can code mutually recursive functions into non-mutually recursive functions,[203] they will hardly be able to understand complicated formulas where these encodings occur, and so they will have severe problems in assisting the proving system in the construction of hard proofs. Partiality due to non-termination essentially occurs in interpreters with undecidable domains. Partiality due to missing cases of the definition can often be avoided by overspecification in theory, but not in practice where the unintended results of overspecification may complicate matters considerably.

[201] We cannot claim that QuodLibet is actually able to execute proofs whose crucial proof steps mirror exactly the ones intended by its human users, simply because this was not scientifically investigated in terms of cognitive psychology. Users, however, considered it to be more appropriate that other systems in this aspect, mostly due to the direct support for partial and mutual function specification, cf. [Löchner, 2006]. Moreover, the four dozens of elementary rules of QuodLibet's inference machine were designed to mirror the way human's organize their proofs (cf. [Wirth, 1997], [Kühler, 2000]); so a user has to deal with one natural inference step where Oyster may have hundreds of intuitionistic steps. The appropriateness of our calculus for interchanging information with humans deteriorated, however, after adding inference rules for the efficient implementation of Presburger Arithmetic, as we will explain below. Note that the calculus is only the lowest logic level a user of a theorem-proving system may have to deal with; from our experience with many such systems we came to the firm conviction, however, that the automation of proof search will always fail on the lowest logic level from time to time, such that human-oriented state-of-the-art logic calculi are are essential for the acceptance of automated, interactive theorem provers by their users.

[202] See [Kühler and Wirth, 1996] for the first publication of the object-level consistency of the specifications that are admissible and supported with strong induction heuristics in QuodLibet. In [Kühler and Wirth, 1996], a huge proof from the original 1995 edition of [Wirth, 2005b] guaranteed the consistency. Moreover, the most relevant and appropriate one of the seven inductive validities of [Wirth and Gramlich, 1994b] is chosen for QuodLibet in [Kühler and Wirth, 1996] (no longer the initial or free models typical for implicit induction!).

[203] See the first paragraph of § 5.7.

For instance, Bernd Löchner (*1967) (a user, not a developer of QuodLibet) concludes in [Löchner, 2006, p. 76]:

> "The translation of the different specifications into the input language of the inductive theorem prover QuodLibet [Avenhaus *et al.*, 2003] was straightforward. We later realized that this is difficult or impossible with several other inductive provers as these have problems with mutual recursive functions and partiality" ...

Based on the *descente infinie* inference system for clausal first-order logic of [Wirth and Kühler, 1995];[204] the system development of QuodLibet in Common Lisp (cf. § 6.5), mostly by Kühler and Tobias Schmidt-Samoa (*1973), lasted from 1995 to 2006. The system was described and demonstrated at the 19th Int. Conf. on Automated Deduction (CADE), Miami Beach (FL), 2003 [Avenhaus *et al.*, 2003]. The extension of the *descente infinie* inference systems of QuodLibet to the full [modal] higher-order logic of [Wirth, 2004; 2013] has not been implemented yet.

To the best of our knowledge, QuodLibet is the first theorem prover whose proof state is an and-or-tree (of clauses); actually, a forest of such trees, so that in a mutual induction proof each conjecture providing induction hypotheses has its own tree [Kühler, 2000]. An extension of the recursion analysis of [Boyer and Moore, 1979] for constructor-style specifications (cf. § 5.5) was developed by writing and testing tactics in QuodLibet's Pascal-like[205] meta-language Qml [Kühler, 2000]. To achieve an acceptable run-time performance (but not competitive with ACL2, of course), Qml tactics are compiled before execution.

In principle, termination proofs are not required, simply because termination is not an admissibility restriction in QuodLibet. Instead, definition-time recursion analysis uses induction lemmas (cf. § 6.3.7) to prove lemmas on function domains by induction.[206]

At proof time, recursion analysis is used by the standard tactic only to determine the induction variables from the induction templates: As seen in Example 3 of § 4.7 w.r.t. the strengthened transitivity of lessp (as compared to the explicit-induction proof in Example 12 of § 6.2.6 and Example 23 of § 6.3.8), subsumption and merging of schemes are not required in *descente infinie*.[207]

[204] Later improvements of this inference system are found in [Wirth, 1997], [Kühler, 2000], and [Schmidt-Samoa, 2006b].

[205] See [Wirth, 1971] for the programming language Pascal. The critical decision for an imperative instead of a functional tactics language turned out to be most appropriate during the ten years of using Qml.

[206] While domain lemmas for totally defined functions are usually found without interaction and total functions do not provide relevant overhead in QuodLibet, the user often has to help in case of partial function definitions by providing *domain lemmas* such as

$$\text{Def } \mathsf{delfirst}(x, l), \quad \mathsf{mbp}(x, l) \neq \mathsf{true},$$

for delfirst defined via (delfirst1–2) of § 4.5.

An enormous speed-up of QuodLibet and an extension of its automatically provable theorems was achieved by Schmidt-Samoa during his PhD work with the system in 2004–2006. He developed a marking concept for the tagging of rewrite lemmas (cf. § 6.3.1), where the elements of a clause can be marked as Forbidden, Mandatory, Obligatory, and Generous, to control the recursive relief of conditions in contextual rewriting [Schmidt-Samoa, 2006b; 2006c]. Moreover, a very simple, but most effective reuse mechanism analyzes during a proof attempt whether it actually establishes a proof of some sub-clause, and uses this knowledge to crop conjunctive branches that do not contribute to the actual goal [Schmidt-Samoa, 2006b]. Finally, an even closer integration of linear arithmetic (cf. Note 174) with excellent results [Schmidt-Samoa, 2006a; 2006b] questioned one of the basic principles of QuodLibet, namely the idea that the prover does not try to be clever, but stops early if there is no progress visible, and presents the human user the proof state in a nice graphical tree representation: The expanded highly-optimized formulation of arithmetic by means of special functions for the decidable fragment of Presburger Arithmetic results in clauses that do not easily admit human inspection anymore. We did not find means to overcome this, because we did not find a way to fold theses clauses to achieve a human-oriented higher level of abstraction.

QuodLibet is, of course, able to do all[208] *descente infinie* proofs of our examples automatically. Moreover, QuodLibet finds all proofs for the irrationality of the square root of two indicated in Figure 4 (sketched in § 6.3.9) automatically and without explicit hints on the induction ordering (say, via newly defined nonsensical functions, such as the one given in (sqrtio1) of § 6.3.9) — provided that the required lemmas are available.

All in all, QuodLibet has proved that *descente infinie* ("lazy induction") goes well together with explicit induction and that we have reason to hope that eager induction-hypotheses generation can be overcome for theorems with difficult induction proofs, sacrificing neither efficiency nor the usefulness of the excellent heuristic knowledge developed in explicit induction. Why *descente infinie* and human-orientedness should remain on the agenda for induction in mathematics assistance systems is explained in the manifesto [Wirth, 2012c].

[207] Although it is not a must and not part of the standard tactic, induction hypotheses may be generated eagerly in QuodLibet to enhance generalization as in Example 5 of § 4.9, in which case subsumption and merging of induction schemes as described in § 6.3.8 are required. Moreover, the concept of flawed induction schemes of QuodLibet (taken over from Thm as well, cf. § 6.3.8) depends on the mergeability of schemes. Furthermore, QuodLibet actually applies some merging techniques to plan case analyses optimized for induction [Kühler, 2000, § 8.3.3]. The question why QuodLibet adopts the great ideas of recursion analysis from Thm, but does not follow them precisely, has two answers: First, it was necessary to extend the heuristics of Thm to deal with constructor-style definitions. The second answer was already given in § 6.3.9: Testing is the only judge on heuristics.

[208] These three *descente infinie* proofs are presented as Examples 2 and 3 of § 4.7, and Example 5 of § 4.9.

## 8 LESSONS LEARNED

What lessons can we draw from the history of the automation of induction?

- Do not be too inclined to follow the current fads. Choose a hard problem, give thought to the "right" foundations, and then pursue its solution with patience and perseverance.

- Another piece of oft-repeated advice to the young researcher: start simply.

  From the standpoint of formalizing microprocessors, investing in a theorem prover supporting only `NIL` and `CONS` is clearly inadequate. From the standpoint of understanding induction and simplification, however, it presents virtually all the problems, and its successors then gradually refined and elaborated the techniques. The four key provers discussed here — the PURE LISP THEOREM PROVER, THM, NQTHM, and ACL2 — are clearly "of a kind". The lessons learned from one tool directly informed the design of the next.

- If you are interested in building an inductive theorem prover, do not make the mistake of focusing merely on an induction principle and the heuristics for controlling it. A successful inductive theorem prover must be able to simplify and generalize. Ideally, it should be able to invent new concepts to express inductively provably theorems.

- If theorems and proofs are simple and obvious for humans, a good automatic theorem prover ought not to struggle with them. If it takes a lot of time and machinery to prove obvious theorems, then truly interesting theorems are out of reach.

- Do not be too eager to add features that break old ones. Instead, truly explore the extent to which new problems can be formalized within the existing framework so as to exploit the power of the existing system.

  Had Boyer and Moore adopted higher-order logic initially or attempted to solve the problem solely by exhaustive searching in a general purpose logic calculus, the discovery of many powerful techniques would have been delayed.

- We strongly recommend collecting all your successful proofs into a regression suite and re-running your improved provers on this suite regularly. It is remarkably easy to "improve" a theorem prover such that it discovers a new proof at the cost of failing to re-discover old ones.

  The ACL2 regression suite, which is used as the acceptance test that any suggested possible improvement has to pass, contains over 90,000 `DEFTHM` commands, i.e. conjectures to be proved. It is an invaluable resource to Kaufmann and Moore when they explore new heuristics.

- Finally, Boyer and Moore did not give names to their provers before ACL2, and so they became most commonly known under the name *the Boyer–Moore theorem prover*.

  So here is some advice to young researchers who want to become well-known: Build a good system, but do not give it a name, so that people have to attach your name to it!

## 9 CONCLUSION

> "One of the reasons our theorem prover is successful is that we trick the user into telling us the proof. And the best example of that, that I know, is: If you want to prove that there exists a prime factorization — that is to say a list of primes whose product is any given number — then the way you state it is: You define a function that takes a natural number and delivers a list of primes, and then you prove that it does that. And, of course, the definition of that function is everybody else's proof. The absence of quantifiers and the focus on constructive, you know, recursive definitions forces people to do the work. And so then, when the theorem prover proves it, they say 'Oh what wonderful theorem prover!', without even realizing they sweated bullets to express the theorem in that impoverished logic."

said Moore, and Boyer agreed laughingly.[209]

## ACKNOWLEGDGEMENTS

We would like to thank Fabio Acerbi, Klaus Barner, Anne O. Boyer, Robert S. Boyer, Alan Bundy, Catherine Goldstein, Bernhard Gramlich, Warren A. Hunt, Matt Kaufmann, Ulrich Kühler, Klaus Madlener, Jo Moore, Peter Padawitz, Marianeh Rezaei, Tobias Schmidt-Samoa, and Judith Stengel.

[209] [Wirth, 2012d].

## BIBLIOGRAPHY

[Abrahams *et al.*, 1980] Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors. *Conference Record of the 7th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Las Vegas (NV), 1980.* ACM Press, 1980. `http://dl.acm.org/citation.cfm?id=567446`.

[Acerbi, 2000] Fabio Acerbi. Plato: Parmenides 149a7–c3. A proof by complete induction? *Archive for History of Exact Sciences*, 55:57–76, 2000.

[Ackermann, 1928] Wilhelm Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928. Received Jan. 20, 1927.

[Ackermann, 1940] Wilhelm Ackermann. Zur Widerspruchsfreiheit der Zahlentheorie. *Mathematische Annalen*, 117:163–194, 1940. Received Aug. 15, 1939.

[Aït-Kaci and Nivat, 1989] Hassan Aït-Kaci and Maurice Nivat, editors. *Proc. of the Colloquium on Resolution of Equations in Algebraic Structures (CREAS), Lakeway (TX), 1987.* Academic Press (Elsevier), 1989.

[Anon, 2005] Anon. Advanced Architecture MicroProcessor 7 Government (AAMP7G) microprocessor. Rockwell Collins, Inc. WWW only: `http://www.rockwellcollins.com/sitecore/content/Data/Products/Information_Assurance/Cryptography/AAMP7G_Microprocessor.aspx`, 2005.

[Armando *et al.*, 2008] Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors. *4th Int. Joint Conf. on Automated Reasoning (IJCAR), Sydney (Australia), 2008*, number 5195 in Lecture Notes in Artificial Intelligence. Springer, 2008.

[Aubin, 1976] Raymond Aubin. *Mechanizing Structural Induction.* PhD thesis, Univ. Edinburgh, 1976. Short version is [Aubin, 1979]. `http://hdl.handle.net/1842/6649`.

[Aubin, 1979] Raymond Aubin. Mechanizing Structural Induction — Part I: Formal System. Part II: Strategies. *Theoretical Computer Sci.*, 9:329–345+347–362, 1979. Received March (Part I) and November (Part II) 1977, rev. March 1978. Long version is [Aubin, 1976].

[Autexier *et al.*, 1999] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. System description: Inka 5.0 – a logical voyager. 1999. In [Ganzinger, 1999, pp. 207–211].

[Autexier, 2005] Serge Autexier. On the dynamic increase of multiplicities in matrix proof methods for classical higher-order logic. 2005. In [Beckert, 2005, pp. 48–62].

[Avenhaus *et al.*, 2003] Jürgen Avenhaus, Ulrich Kühler, Tobias Schmidt-Samoa, and Claus-Peter Wirth. How to prove inductive theorems? QuodLibet! 2003. In [Baader, 2003, pp. 328–333], `http://wirth.bplaced.net/p/quodlibet`.

[Baader, 2003] Franz Baader, editor. *19th Int. Conf. on Automated Deduction (CADE), Miami Beach (FL), 2003*, number 2741 in Lecture Notes in Artificial Intelligence. Springer, 2003.

[Baaz and Leitsch, 1995] Matthias Baaz and Alexander Leitsch. Methods of functional extension. *Collegium Logicum — Annals of the Kurt Gödel Society*, 1:87–122, 1995.

[Bachmair *et al.*, 1992] Leo Bachmair, Harald Ganzinger, and Wolfgang J. Paul, editors. *Informatik – Festschrift zum 60. Geburtstag von Günter Hotz.* B. G. Teubner Verlagsgesellschaft, 1992.

[Bachmair, 1988] Leo Bachmair. Proof by consistency in equational theories. 1988. In [LICS, 1988, pp. 228–233].

[Bajscy, 1993] Ruzena Bajscy, editor. *Proc. 13th Int. Joint Conf. on Artificial Intelligence (IJCAI), Chambery (France).* Morgan Kaufmann, Los Altos (CA) (Elsevier), 1993. `http://ijcai.org/Past%20Proceedings`.

[Barendregt, 1981] Hen(dri)k P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland (Elsevier), 1981. 1st edn. (final rev. edn. is [Barendregt, 2012]).

[Barendregt, 2012] Hen(dri)k P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* Number 40 in Studies in Logic. College Publications, London, 2012. 6th rev. edn. (1st edn. is [Barendregt, 1981]).

[Barner, 2001a] Klaus Barner. Pierre Fermat (1601?–1665) — His life beside mathematics. *European Mathematical Society Newsletter*, 43 (Dec. 2001):12–16, 2001. Long version in German is [Barner, 2001b]. `www.ems-ph.org/journals/newsletter/pdf/2001-12-42.pdf`.

[Barner, 2001b] Klaus Barner. Das Leben Fermats. *DMV-Mitteilungen*, 3/2001:12–26, 2001. Extensions in [Barner, 2007]. Short versions in English are [Barner, 2001c; 2001a].

[Barner, 2001c] Klaus Barner. How old did Fermat become? *NTM Internationale Zeitschrift für Geschichte und Ethik der Naturwissenschaften, Technik und Medizin, Neue Serie, ISSN 00366978*, 9:209–228, 2001. Long version in German is [Barner, 2001b]. New results on the subject in [Barner, 2007].

[Barner, 2007] Klaus Barner. Neues zu Fermats Geburtsdatum. *DMV-Mitteilungen*, 15:12–14, 2007. *(Further support for the results of [Barner, 2001c], narrowing down Fermat's birth date from 1607/8 to Nov. 1607).*

[Basin and Walsh, 1996] David Basin and Toby Walsh. A calculus for and termination of rippling. *J. Automated Reasoning*, 16:147–180, 1996.

[Becker, 1965] Oscar Becker, editor. *Zur Geschichte der griechischen Mathematik.* Wissenschaftliche Buchgesellschaft, Darmstadt, 1965.

[Beckert, 2005] Bernhard Beckert, editor. *14th Int. Conf. on Tableaux and Related Methods, Koblenz (Germany), 2005*, number 3702 in Lecture Notes in Artificial Intelligence. Springer, 2005.

[Bell and Thayer, 1976] Thomas E. Bell and T. A. Thayer. Software requirements: Are they really a problem? 1976. In [Yeh and Ramamoorthy, 1976, pp. 61–68], `http://pdf.aminer.org/000/361/405/software_requirements_are_they_really_a_problem.pdf`.

[Benzmüller *et al.*, 2008] Christoph Benzmüller, Frank Theiss, Lawrence C. Paulson, and Arnaud Fietzke. LEO-II — a cooperative automatic theorem prover for higher-order logic. 2008. In [Armando *et al.*, 2008, pp. 162–170].

[Berka and Kreiser, 1973] Karel Berka and Lothar Kreiser, editors. *Logik-Texte – Kommentierte Auswahl zur Geschichte der modernen Logik.* Akademie Verlag GmbH, Berlin, 1973. 2nd rev. edn. (1st edn. 1971; 4th rev. rev. edn. 1986).

[Bevers and Lewi, 1990] Eddy Bevers and Johan Lewi. Proof by consistency in conditional equational theories. Tech. Report CW 102, Dept. Comp. Sci., K. U. Leuven, 1990. Rev. July 1990. Short version in [Kaplan and Okada, 1991, pp. 194–205].

[Bevier *et al.*, 1989] William R. Bevier, Warren A. Hunt, J Strother Moore, and William D. Young. An approach to systems verification. *J. Automated Reasoning*, 5:411–428, 1989.

[Bevier, 1989] William R. Bevier. Kit and the short stack. *J. Automated Reasoning*, 5:519–530, 1989.

[Bibel and Kowalski, 1980] Wolfgang Bibel and Robert A. Kowalski, editors. *5th Int. Conf. on Automated Deduction (CADE), Les Arcs (France), 1980*, number 87 in Lecture Notes in Computer Science. Springer, 1980.

[Biundo *et al.*, 1986] Susanne Biundo, Birgit Hummel, Dieter Hutter, and Christoph Walther. The Karlsruhe inductive theorem proving system. 1986. In [Siekmann, 1986, pp. 673–675].

[Bledsoe and Loveland, 1984] W. W. Bledsoe and Donald W. Loveland, editors. *Automated Theorem Proving: After 25 Years.* Number 29 in Contemporary Mathematics. American Math. Soc., Providence (RI), 1984. Proc. of the Special Session on Automatic Theorem Proving, 89th Annual Meeting of the American Math. Soc., Denver (CO), Jan. 1983.

[Bledsoe *et al.*, 1971] W. W. Bledsoe, Robert S. Boyer, and William H. Henneman. Computer proofs of limit theorems. 1971. In [Cooper, 1971, pp. 586–600]. Long version is [Bledsoe *et al.*, 1972].

[Bledsoe *et al.*, 1972] W. W. Bledsoe, Robert S. Boyer, and William H. Henneman. Computer proofs of limit theorems. *Artificial Intelligence*, 3:27–60, 1972. Short version is [Bledsoe *et al.*, 1971].

[Bledsoe, 1971] W. W. Bledsoe. Splitting and reduction heuristics in automatic theorem proving. *Artificial Intelligence*, 2:55–77, 1971.

[Bouajjani and Maler, 2009] Ahmed Bouajjani and Oded Maler, editors. *Proc. 21st Int. Conf. on Computer Aided Verification (CAV), Grenoble (France), 2009*, volume 5643 of *Lecture Notes in Computer Science.* Springer, 2009.

[Bouhoula and Rusinowitch, 1995] Adel Bouhoula and Michaël Rusinowitch. Implicit induction in conditional theories. *J. Automated Reasoning*, 14:189–235, 1995.

[Bourbaki, 1939] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Fascicule De Résultats.* Number 846 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1939. 1st edn., VIII + 50 pp.. Review is [Church, 1946]. 2nd rev. extd. edn. is [Bourbaki, 1951].

[Bourbaki, 1951] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Fascicule De Résultats.* Number 846-1141 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1951. 2nd rev. extd. edn. of [Bourbaki, 1939]. 3rd rev. extd. edn. is [Bourbaki, 1958b].

[Bourbaki, 1954] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Chapitre I & II.* Number 1212 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1954. 1st edn.. 2nd rev. edn. is [Bourbaki, 1960].

[Bourbaki, 1956] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Chapitre III.* Number 1243 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1956. 1st edn., II + 119 + 4 (mode d'emploi) + 23 (errata no. 6) pp.. 2nd rev. extd. edn. is [Bourbaki, 1967].

[Bourbaki, 1958a] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Chapitre IV.* Number 1258 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1958. 1st edn.. 2nd rev. extd. edn. is [Bourbaki, 1966a].

[Bourbaki, 1958b] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Fascicule De Résultats.* Number 1141 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1958. 3rd rev. extd. edn. of [Bourbaki, 1951]. 4th rev. extd. edn. is [Bourbaki, 1964].

[Bourbaki, 1960] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Chapitre I & II.* Number 1212 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1960. 2nd rev. extd. edn. of [Bourbaki, 1954]. 3rd rev. edn. is [Bourbaki, 1966b].

[Bourbaki, 1964] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Fascicule De Résultats.* Number 1141 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1964. 4th rev. extd. edn. of [Bourbaki, 1958b]. 5th rev. extd. edn. is [Bourbaki, 1968b].

[Bourbaki, 1966a] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Chapitre IV.* Number 1258 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1966. 2nd rev. extd. edn. of [Bourbaki, 1958a]. English translation in [Bourbaki, 1968a].

[Bourbaki, 1966b] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Chapitres I & II.* Number 1212 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1966. 3rd rev. edn. of [Bourbaki, 1960], VI + 143 + 7 (errata no. 13) pp.. English translation in [Bourbaki, 1968a].

[Bourbaki, 1967] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Chapitre III.* Number 1243 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1967. 2nd rev. extd. edn. of [Bourbaki, 1956], 151 + 7 (errata no. 13) pp.. 3rd rev. edn. results from executing these errata. English translation in [Bourbaki, 1968a].

[Bourbaki, 1968a] Nicolas Bourbaki. *Elements of Mathematics — Theory of Sets.* Actualités Scientifiques et Industrielles. Hermann, Paris, jointly published with AdiWes International Series in Mathematics, Addison–Wesley, Reading (MA), 1968. English translation of [Bourbaki, 1966b; 1967; 1966a; 1968b].

[Bourbaki, 1968b] Nicolas Bourbaki. *Éléments des Mathématique — Livre 1: Théorie des Ensembles. Fascicule De Résultats.* Number 1141 in Actualités Scientifiques et Industrielles. Hermann, Paris, 1968. 5th rev. extd. edn. of [Bourbaki, 1964]. English translation in [Bourbaki, 1968a].

[Boyer and Moore, 1971] Robert S. Boyer and J Strother Moore. The sharing of structure in resolution programs. Memo 47, Univ. Edinburgh, Dept. of Computational Logic, 1971. II + 24 pp.. Revised version is [Boyer and Moore, 1972].

[Boyer and Moore, 1972] Robert S. Boyer and J Strother Moore. The sharing of structure in theorem-proving programs. 1972. In [Meltzer and Michie, 1972, pp. 101–116].

[Boyer and Moore, 1973] Robert S. Boyer and J Strother Moore. Proving theorems about LISP functions. 1973. In [Nilsson, 1973, pp. 486–493]. `http://ijcai.org/Past%20Proceedings/IJCAI-73/PDF/053.pdf`. Rev. version, extd. with a section "Failures", is [Boyer and Moore, 1975].

[Boyer and Moore, 1975] Robert S. Boyer and J Strother Moore. Proving theorems about LISP functions. *J. of the ACM*, 22:129–144, 1975. Rev. extd. edn. of [Boyer and Moore, 1973]. Received Sept. 1973, Rev. April 1974.

[Boyer and Moore, 1977a] Robert S. Boyer and J Strother Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977. `http://doi.acm.org/10.1145/359842.359859`.

[Boyer and Moore, 1977b] Robert S. Boyer and J Strother Moore. A lemma driven automatic theorem prover for recursive function theory. 1977. In [Reddy, 1977, Vol. I, pp. 511–519]. http://ijcai.org/Past%20Proceedings/IJCAI-77-VOL1/PDF/089.pdf.

[Boyer and Moore, 1979] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press (Elsevier), 1979. http://www.cs.utexas.edu/users/boyer/acl.text.

[Boyer and Moore, 1981a] Robert S. Boyer and J Strother Moore, editors. *The Correctness Problem in Computer Science*. Academic Press (Elsevier), 1981.

[Boyer and Moore, 1981b] Robert S. Boyer and J Strother Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. 1981. In [Boyer and Moore, 1981a, pp. 103–184].

[Boyer and Moore, 1984a] Robert S. Boyer and J Strother Moore. A mechanical proof of the Turing completeness of pure LISP. 1984. In [Bledsoe and Loveland, 1984, pp. 133–167].

[Boyer and Moore, 1984b] Robert S. Boyer and J Strother Moore. A mechanical proof of the unsolvability of the halting problem. *J. of the ACM*, 31:441–458, 1984.

[Boyer and Moore, 1984c] Robert S. Boyer and J Strother Moore. Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, 91:181–189, 1984.

[Boyer and Moore, 1985] Robert S. Boyer and J Strother Moore. Program verification. *J. Automated Reasoning*, 1:17–23, 1985.

[Boyer and Moore, 1987] Robert S. Boyer and J Strother Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. Technical Report ICSCA-CMP-52, Inst. for Computing Science and Computing Applications, The University of Texas at Austin, 1987. Printed Jan. 1987. Also published as [Boyer and Moore, 1988a; 1989].

[Boyer and Moore, 1988a] Robert S. Boyer and J Strother Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *J. Automated Reasoning*, 4:117–172, 1988. Received Feb. 11, 1987. Also pubished as [Boyer and Moore, 1987; 1989].

[Boyer and Moore, 1988b] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press (Elsevier), 1988. 2nd rev. extd. edn. is [Boyer and Moore, 1998].

[Boyer and Moore, 1988c] Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of Linear Arithmetic, note=In [Hayes *et al.*, 1988, pp. 83–124],. 1988.

[Boyer and Moore, 1989] Robert S. Boyer and J Strother Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. 1989. In [Broy, 1989, pp. 95–145] (received Jan. 1988). Also published as [Boyer and Moore, 1987; 1988a].

[Boyer and Moore, 1990] Robert S. Boyer and J Strother Moore. A theorem prover for a computational logic. 1990. In [Stickel, 1990, pp. 1–15].

[Boyer and Moore, 1998] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. International Series in Formal Methods. Academic Press (Elsevier), 1998. 2nd rev. extd. edn. of [Boyer and Moore, 1988b], rev. to work with NQTHM–1992, a new version of NQTHM.

[Boyer and Yu, 1992] Robert S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. 1992. In [Kapur, 1992, 416–430].

[Boyer and Yu, 1996] Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *J. of the ACM*, 43:166–192, 1996.

[Boyer *et al.*, 1973] Robert S. Boyer, D. Julian M. Davies, and J Strother Moore. The 77-editor. Memo 62, Univ. Edinburgh, Dept. of Computational Logic, 1973.

[Boyer *et al.*, 1976] Robert S. Boyer, J Strother Moore, and Robert E. Shostak. Primitive recursive program transformations. 1976. In [Graham *et al.*, 1976, pp. 171–174]. http://doi.acm.org/10.1145/800168.811550.

[Boyer, 1971] Robert S. Boyer. *Locking: a restriction of resolution*. PhD thesis, The University of Texas at Austin, 1971.

[Boyer, 2012] Robert S. Boyer. E-mail to Claus-Peter Wirth, Nov. 19, 2012.

[Brock and Hunt, 1999] Bishop Brock and Warren A. Hunt. Formal analysis of the Motorola CAP DSP. 1999. In [Hinchey and Bowen, 1999, pp. 81–116].

[Brotherston and Simpson, 2007] James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. 2007. In [LICS, 2007, pp. 51–62?]. Thoroughly rev. version in [Brotherston and Simpson, 2011].

[Brotherston and Simpson, 2011] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *J. Logic and Computation*, 21:1177–1216, 2011. Thoroughly rev. version of [Brotherston, 2005] and [Brotherston and Simpson, 2007]. Received April 3, 2009. Published online Sept. 30, 2010, `http://dx.doi.org/10.1093/logcom/exq052`.

[Brotherston, 2005] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. 2005. In [Beckert, 2005, pp. 78–92]. Thoroughly rev. version in [Brotherston and Simpson, 2011].

[Brown, 2012] Chad E. Brown. SATALLAX: An automatic higher-order prover. 2012. In [Gramlich *et al.*, 2012, pp. 111–117].

[Broy, 1989] Manfred Broy, editor. *Constructive Methods in Computing Science*, number F 55 in NATO ASI Series. Springer, 1989.

[Buch and Hillenbrand, 1996] Armin Buch and Thomas Hillenbrand. WALDMEISTER*: Development of a High Performance Completion-Based Theorem Prover*. SEKI-Report SR–96–01 (ISSN 1860–5931). SEKI Publications, FB Informatik, Univ. Kaiserslautern, 1996. `agent.informatik.uni-kl.de/seki/1996/Buch.SR-96-01.ps.gz`.

[Bundy *et al.*, 1989] Alan Bundy, Frank van Harmelen, Jane Hesketh, Alan Smaill, and Andrew Stevens. A rational reconstruction and extension of recursion analysis. 1989. In [Sridharan, 1989, pp. 359–365].

[Bundy *et al.*, 1990] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The OYSTER/CLAM system. 1990. In [Stickel, 1990, pp. 647–648].

[Bundy *et al.*, 1991] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. *Rippling: A Heuristic for Guiding Inductive Proofs*. 1991. DAI Research Paper No. 567, Dept. Artificial Intelligence, Univ. Edinburgh. Also in Artificial Intelligence 62:185–253, 1993.

[Bundy *et al.*, 2005] Alan Bundy, Dieter Hutter, David Basin, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge Univ. Press, 2005.

[Bundy, 1988] Alan Bundy. *The use of Explicit Plans to Guide Inductive Proofs*. 1988. DAI Research Paper No. 349, Dept. Artificial Intelligence, Univ. Edinburgh. Short version in [Lusk and Overbeek, 1988, pp. 111–120].

[Bundy, 1989] Alan Bundy. *A Science of Reasoning*. 1989. DAI Research Paper No. 445, Dept. Artificial Intelligence, Univ. Edinburgh. Also in [Lassez and Plotkin, 1991, pp. 178–198].

[Bundy, 1994] Alan Bundy, editor. *12th Int. Conf. on Automated Deduction (CADE), Nancy, 1994*, number 814 in Lecture Notes in Artificial Intelligence. Springer, 1994.

[Bundy, 1999] Alan Bundy. *The Automation of Proof by Mathematical Induction*. Informatics Research Report No. 2, Division of Informatics, Univ. Edinburgh, 1999. Also in [Robinson and Voronkow, 2001, Vol. 1, pp. 845–911].

[Burstall *et al.*, 1971] Rod M. Burstall, John S. Collins, and Robin J. Popplestone. *Programming in* POP–2. Univ. Edinburgh Press, 1971.

[Burstall, 1969] Rod M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12:48–51, 1969. Received April 1968, rev. Aug. 1968.

[Bussey, 1917] W. H. Bussey. The origin of mathematical induction. *American Mathematical Monthly*, XXIV:199–207, 1917.

[Bussotti, 2006] Paolo Bussotti. *From Fermat to Gauß: indefinite descent and methods of reduction in number theory*. Number 55 in Algorismus. Dr. Erwin Rauner Verlag, Augsburg, 2006.

[Cajori, 1918] Florian Cajori. Origin of the name "mathematical induction". *American Mathematical Monthly*, 25:197–201, 1918.

[Church, 1946] Alonzo Church. Review of [Bourbaki, 1939]. *J. Symbolic Logic*, 11:91, 1946.

[Clocksin and Mellish, 2003] William F. Clocksin and Christopher S. Mellish. *Programming in* PROLOG. Springer, 2003. 5th edn. (1st edn. 1981).

[Cohn, 1965] Paul Moritz Cohn. *Universal Algebra*. Harper & Row, New York, 1965. 1st edn.. 2nd rev. edn. is [Cohn, 1981].

[Cohn, 1981] Paul Moritz Cohn. *Universal Algebra*. Number 6 in Mathematics and Its Applications. D. Reidel Publ., Dordrecht, now part of Springer Science+Business Media, 1981. 2nd rev. edn. (1st edn. is [Cohn, 1965]).

[Comon, 1997] Hubert Comon, editor. *8th Int. Conf. on Rewriting Techniques and Applications (RTA), Sitges (Spain), 1997*, number 1232 in Lecture Notes in Computer Science. Springer, 1997.

[Comon, 2001] Hubert Comon. Inductionless induction. 2001. In [Robinson and Voronkow, 2001, Vol. I, pp. 913–970].

[Constable *et al.*, 1985] Robert L. Constable, Stuart F. Allen, H. M. Bromly, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the* NUPRL *Proof Development System*. Prentice–Hall, Inc., 1985. `http://www.nuprl.org/book`.

[Cooper, 1971] D. C. Cooper, editor. *Proc. 2nd Int. Joint Conf. on Artificial Intelligence (IJCAI), Sept. 1971, Imperial College, London*. Morgan Kaufmann, Los Altos (CA), Los Altos (CA), 1971. `http://ijcai.org/Past%20Proceedings/IJCAI-1971/CONTENT/content.htm`.

[DAC, 2001] *Proc. 38th Design Automation Conference (DAC), Las Vegas (NV), 2001*. ACM Press, 2001.

[Darlington, 1968] Jared L. Darlington. Automated theorem proving with equality substitutions and mathematical induction. 1968. In [Michie, 1968, pp. 113–127].

[Davis, 2009] Jared Davis. *A Self-Verifying Theorem Prover*. PhD thesis, The University of Texas at Austin, 2009.

[Dedekind, 1888] Richard Dedekind. *Was sind und was sollen die Zahlen*. Vieweg, Braunschweig, 1888. Also in [Dedekind, 1930–32, Vol. 3, pp. 335–391]. Also in [Dedekind, 1969].

[Dedekind, 1930–32] Richard Dedekind. *Gesammelte mathematische Werke*. Vieweg, Braunschweig, 1930–32. Ed. by Robert Fricke, Emmy Noether, and Öystein Ore.

[Dedekind, 1969] Richard Dedekind. *Was sind und was sollen die Zahlen? Stetigkeit und irrationale Zahlen*. Friedrich Vieweg und Sohn, Braunschweig, 1969.

[Dennis *et al.*, 2005] Louise A. Dennis, Mateja Jamnik, and Martin Pollet. On the comparison of proof planning systems λCLAM, ΩMEGA and ISAPLANNER. *Electronic Notes in Theoretical Computer Sci.*, 151:93–110, 2005.

[Dershowitz and Jouannaud, 1990] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. 1990. In [Leeuwen, 1990, Vol. B, pp. 243–320].

[Dershowitz and Lindenstrauss, 1995] Nachum Dershowitz and Naomi Lindenstrauss, editors. *4th Int. Workshop on Conditional Term Rewriting Systems (CTRS), Jerusalem, 1994*, number 968 in Lecture Notes in Computer Science, 1995.

[Dershowitz, 1989] Nachum Dershowitz, editor. *3rd Int. Conf. on Rewriting Techniques and Applications (RTA), Chapel Hill (NC), 1989*, number 355 in Lecture Notes in Computer Science. Springer, 1989.

[Dietrich, 2011] Dominik Dietrich. *Assertion Level Proof Planning with Compiled Strategies*. Optimus Verlag, Alexander Mostafa, Göttingen, 2011. PhD thesis, Dept. Informatics, FR Informatik, Saarland Univ..

[Euclid, ca. 300 B.C.] Euclid, of Alexandria. *Elements*. ca. 300 B.C.. Web version without the figures: `http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.01.0085`. English translation: Thomas L. Heath (ed.). *The Thirteen Books of Euclid's Elements*. Cambridge Univ. Press, 1908; web version without the figures: `http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.01.0086`. English web version (incl. figures): D. E. Joyce (ed.). *Euclid's Elements*. `http://aleph0.clarku.edu/~djoyce/java/elements/elements.html`, Dept. Math. & Comp. Sci., Clark Univ., Worcester (MA).

[Fermat, 1891ff.] Pierre Fermat. *Œuvres de Fermat*. Gauthier-Villars, Paris, 1891ff.. Ed. by Paul Tannery, Charles Henry.

[Fitting, 1990] Melvin Fitting. *First-order logic and automated theorem proving*. Springer, 1990. 1st edn. (2nd rev. edn. is [Fitting, 1996]).

[Fitting, 1996] Melvin Fitting. *First-order logic and automated theorem proving*. Springer, 1996. 2nd rev. edn. (1st edn. is [Fitting, 1990]).

[FOCS, 1980] *Proc. 21st Annual Symposium on Foundations of Computer Sci., Syracuse, 1980*. IEEE Press, 1980. `http://ieee-focs.org/`.

[Fowler, 1994] David Fowler. Could the Greeks have used mathematical induction? Did they use it? *Physis*, XXXI(1):253–265, 1994.

[Freudenthal, 1953] Hans Freudenthal. Zur Geschichte der vollständigen Induktion. *Archives Internationales d'Histoire des Sciences*, 6:17–37, 1953.

[Fribourg, 1986] Laurent Fribourg. A strong restriction of the inductive completion procedure. 1986. In [Kott, 1986, pp. 105–116]. Also in J. Symbolic Computation 8:253–276, 1989, Academic Press (Elsevier).

[Fries, 1822] Jakob Friedrich Fries. *Die mathematische Naturphilosophie nach philosophischer Methode bearbeitet – Ein Versuch.* Christian Friedrich Winter, Heidelberg, 1822.

[Fritz, 1945] Kurt von Fritz. The discovery of incommensurability by Hippasus of Metapontum. *Annals of Mathematics*, 46:242–264, 1945. German translation: *Die Entdeckung der Inkommensurabilität durch Hippasos von Metapont* in [Becker, 1965, pp. 271–308].

[Fuchi and Kott, 1988] Kazuhiro Fuchi and Laurent Kott, editors. *Programming of Future Generation Computers II: Proc. of the 2nd Franco-Japanese Symposium.* North-Holland (Elsevier), 1988.

[Gabbay and Woods, 2004ff.] Dov Gabbay and John Woods, editors. *Handbook of the History of Logic.* North-Holland (Elsevier), 2004ff..

[Gabbay *et al.*, 1994] Dov Gabbay, Christopher John Hogger, and J. Alan Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 2: Deduction Methodologies.* Oxford Univ. Press, 1994.

[Ganzinger and Stuber, 1992] Harald Ganzinger and Jürgen Stuber. Inductive Theorem Proving by Consistency for First-Order Clauses. 1992. In [Bachmair *et al.*, 1992, pp. 441–462]. Also in [Rusinowitch and Remy, 1993, pp. 226–241].

[Ganzinger, 1996] Harald Ganzinger, editor. *7th Int. Conf. on Rewriting Techniques and Applications (RTA), New Brunswick (NJ), 1996*, number 1103 in Lecture Notes in Computer Science. Springer, 1996.

[Ganzinger, 1999] Harald Ganzinger, editor. *16th Int. Conf. on Automated Deduction (CADE), Trento (Italy), 1999*, number 1632 in Lecture Notes in Artificial Intelligence. Springer, 1999.

[Gentzen, 1935] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210,405–431, 1935. Also in [Berka and Kreiser, 1973, pp. 192–253]. English translation in [Gentzen, 1969].

[Gentzen, 1969] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen.* North-Holland (Elsevier), 1969. Ed. by Manfred E. Szabo.

[Geser, 1995] Alfons Geser. A principle of non-wellfounded induction. 1995. In [Margaria, 1995, pp. 117–124].

[Geser, 1996] Alfons Geser. An improved general path order. *J. Applicable Algebra in Engineering, Communication and Computing (AAECC)*, 7:469–511, 1996.

[Gillman, 1987] Leonard Gillman. *Writing Mathematics Well.* The Mathematical Association of America, 1987.

[Göbel, 1985] Richard Göbel. Completion of globally finite term rewriting systems for inductive proofs. 1985. In [Stoyan, 1985, pp. 101–110].

[Gödel, 1931] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. With English translation also in [Gödel, 1986ff., Vol. I, pp. 145–195]. English translation also in [Heijenoort, 1971, pp. 596–616] and in [Gödel, 1962].

[Gödel, 1962] Kurt Gödel. *On formally undecidable propositions of Principia Mathematica and related systems.* Basic Books, New York, 1962. English translation of [Gödel, 1931] by Bernard Meltzer. With an introduction by R. B. Braithwaite. 2nd edn. by Dover Publications, 1992.

[Gödel, 1986ff.] Kurt Gödel. *Collected Works.* Oxford Univ. Press, 1986ff. Ed. by Sol Feferman, John W. Dawson Jr., Warren Goldfarb, Jean van Heijenoort, Stephen C. Kleene, Charles Parsons, Wilfried Sieg, *et al.*.

[Goguen, 1980] Joseph Goguen. How to prove algebraic inductive hypotheses without induction. 1980. In [Bibel and Kowalski, 1980, pp. 356–373].

[Goldstein, 2008] Catherine Goldstein. Pierre Fermat. 2008. In [Gowers *et al.*, 2008, § VI.12, pp. 740–741].

[Gordon, 2000] Mike J. C. Gordon. From LCF to HOL: a short history. 2000. In [Plotkin *et al.*, 2000, pp. 169–186]. `http://www.cl.cam.ac.uk/~mjcg/papers/HolHistory.pdf`.

[Gore *et al.*, 2001] Rajeev Gore, Alexander Leitsch, and Tobias Nipkow, editors. *1st Int. Joint Conf. on Automated Reasoning (IJCAR), Siena (Italy), 2001*, number 2083 in Lecture Notes in Artificial Intelligence. Springer, 2001.

[Gowers *et al.*, 2008] Timothy Gowers, June Barrow-Green, and Imre Leader, editors. *The Princeton Companion to Mathematics.* Princeton Univ. Press, 2008.

[Graham *et al.*, 1976] Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman, editors. *Conference Record of the 3rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Atlanta (GA), Jan. 1976*. ACM Press, 1976. `http://dl.acm.org/citation.cfm?id=800168`.

[Gramlich and Lindner, 1991] Bernhard Gramlich and Wolfgang Lindner. *A Guide to* UNICOM*, an Inductive Theorem Prover Based on Rewriting and Completion Techniques*. SEKI-Report SR–91–17 (ISSN 1860–5931). SEKI Publications, FB Informatik, Univ. Kaiserslautern, 1991. `http://agent.informatik.uni-kl.de/seki/1991/Lindner.SR-91-17.ps.gz`.

[Gramlich and Wirth, 1996] Bernhard Gramlich and Claus-Peter Wirth. Confluence of terminating conditional term rewriting systems revisited. 1996. In [Ganzinger, 1996, pp. 245–259].

[Gramlich *et al.*, 2012] Bernhard Gramlich, Dale A. Miller, and Uli Sattler, editors. *6th Int. Joint Conf. on Automated Reasoning (IJCAR), Manchester, 2012*, number 7364 in Lecture Notes in Artificial Intelligence. Springer, 2012.

[Hayes *et al.*, 1988] Jean E. Hayes, Donald Michie, and Judith Richards, editors. *Proceedings of the 11th Annual Machine Intelligence Workshop (Machine Intelligence 11), Univ. Strathclyde, Glasgow, 1985*. Clarendon Press, Oxford (Oxford Univ. Press), 1988. `aitopics.org/sites/default/files/classic/Machine_Intelligence_11/Machine_Intelligence_v.11.pdf`.

[Heijenoort, 1971] Jean van Heijenoort. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard Univ. Press, 1971. 2nd rev. edn. (1st edn. 1967).

[Herbelin, 2009] Hugo Herbelin, editor. *The 1st* COQ *Workshop*. Inst. für Informatik, Tech. Univ. München, 2009. TUM-I0919, `http://www.lix.polytechnique.fr/coq/files/coq-workshop-TUM-I0919.pdf`.

[Hilbert and Bernays, 1934] David Hilbert and Paul Bernays. *Die Grundlagen der Mathematik — Erster Band*. Number XL in Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen. Springer, 1934. 1st edn. (2nd edn. is [Hilbert and Bernays, 1968]).

[Hilbert and Bernays, 1939] David Hilbert and Paul Bernays. *Die Grundlagen der Mathematik — Zweiter Band*. Number L in Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen. Springer, 1939. 1st edn. (2nd edn. is [Hilbert and Bernays, 1970]).

[Hilbert and Bernays, 1968] David Hilbert and Paul Bernays. *Die Grundlagen der Mathematik I*. Number 40 in Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen. Springer, 1968. 2nd rev. edn. of [Hilbert and Bernays, 1934].

[Hilbert and Bernays, 1970] David Hilbert and Paul Bernays. *Die Grundlagen der Mathematik II*. Number 50 in Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen. Springer, 1970. 2nd rev. edn. of [Hilbert and Bernays, 1939].

[Hilbert and Bernays, 2013] David Hilbert and Paul Bernays. *Grundlagen der Mathematik I — Foundations of Mathematics I, Part A: Title Pages, Prefaces, and §§ 1–2*. `http://wirth.bplaced.net/p/hilbertbernays`, 2013. Thoroughly rev. 2nd edn. (1st edn. College Publications, London, 2011). First English translation and bilingual facsimile edn. of the 2nd German edn. [Hilbert and Bernays, 1968], incl. the annotation and translation of all differences of the 1st German edn. [Hilbert and Bernays, 1934]. Translated and commented by Claus-Peter Wirth. Ed. by Claus-Peter Wirth, Jörg Siekmann, Michael Gabbay, Dov Gabbay. Advisory Board: Wilfried Sieg (chair), Irving H. Anellis, Steve Awodey, Matthias Baaz, Wilfried Buchholz, Bernd Buldt, Reinhard Kahle, Paolo Mancosu, Charles Parsons, Volker Peckhaus, William W. Tait, Christian Tapp, Richard Zach.

[Hillenbrand and Löchner, 2002] Thomas Hillenbrand and Bernd Löchner. The next WALDMEISTER loop. 2002. In [Voronkov, 2002, pp. 486–500]. `http://www.waldmeister.org`.

[Hinchey and Bowen, 1999] Michael G. Hinchey and Jonathan P. Bowen, editors. *Industrial-Strength Formal Methods in Practice*. Formal Approaches to Computing and Information Technology (FACIT). Springer, 1999.

[Hobson and Love, 1913] E. W. Hobson and A. E. H. Love, editors. *Proc. 5th Int. Congress of Mathematicians, Cambridge, Aug 22–28, 1912*. Cambridge Univ. Press, 1913. `http://gallica.bnf.fr/ark:/12148/bpt6k99444q`.

[Howard and Rubin, 1998] Paul Howard and Jean E. Rubin. *Consequences of the Axiom of Choice*. American Math. Society, 1998.

[Hudlak *et al.*, 1999] Paul Hudlak, John Peterson, and Joseph H. Fasel. A gentle introduction to HASKELL. Web only: `http://www.haskell.org/tutorial`, 1999.

[Huet and Hullot, 1980] Gérard Huet and Jean-Marie Hullot. Proofs by induction in equational theories with constructors. 1980. In [FOCS, 1980, pp. 96–107]. Also in J. Computer and System Sci. 25:239–266, 1982, Academic Press (Elsevier).

[Huet, 1980] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. of the ACM*, 27:797–821, 1980.

[Hunt and Swords, 2009] Warren A. Hunt and Sol Swords. Centaur technology media unit verification. 2009. In [Bouajjani and Maler, 2009, pp. 353–367].

[Hunt, 1985] Warren A. Hunt. *FM8501: A Verified Microprocessor*. PhD thesis, The University of Texas at Austin, 1985. Also published as [Hunt, 1994].

[Hunt, 1989] Warren A. Hunt. Microprocessor design verification. *J. Automated Reasoning*, 5:429–460, 1989.

[Hunt, 1994] Warren A. Hunt. *FM8501: A Verified Microprocessor*. Number 795 in Lecture Notes in Artificial Intelligence. Springer, 1994. Originally published as [Hunt, 1985].

[Hutter and Bundy, 1999] Dieter Hutter and Alan Bundy. The design of the CADE-16 Inductive Theorem Prover Contest. 1999. In [Ganzinger, 1999, pp. 374–377].

[Hutter and Sengler, 1996] Dieter Hutter and Claus Sengler. INKA: the next generation. 1996. In [McRobbie and Slaney, 1996, pp. 288–292].

[Hutter and Stephan, 2005] Dieter Hutter and Werner Stephan, editors. *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg Siekmann on the Occasion of His 60th Birthday*. Number 2605 in Lecture Notes in Artificial Intelligence. Springer, 2005.

[Hutter, 1990] Dieter Hutter. Guiding inductive proofs. 1990. In [Stickel, 1990, pp. 147–161].

[Ireland and Bundy, 1994] Andrew Ireland and Alan Bundy. *Productive Use of Failure in Inductive Proof*. 1994. DAI Research Paper No. 716, Dept. Artificial Intelligence, Univ. Edinburgh. Also in: J. Automated Reasoning 16:79–111, 1996, Kluwer (Springer Science+Business Media).

[Jamnik *et al.*, 2003] Mateja Jamnik, Manfred Kerber, Martin Pollet, and Christoph Benzmüller. Automatic learning of proof methods in proof planning. *Logic J. of the IGPL*, 11:647–673, 2003.

[Jouannaud and Kounalis, 1986] Jean-Pierre Jouannaud and Emmanuël Kounalis. Automatic proofs by induction in equational theories without constructors. 1986. In [LICS, 1986, pp. 358–366]. Also in Information and Computation 82:1–33, 1989, Academic Press (Elsevier), 1989.

[Kaplan and Jouannaud, 1988] Stéphane Kaplan and Jean-Pierre Jouannaud, editors. *1st Int. Workshop on Conditional Term Rewriting Systems (CTRS), Orsay (France), 1987*, number 308 in Lecture Notes in Computer Science, 1988.

[Kaplan and Okada, 1991] Stéphane Kaplan and Mitsuhiro Okada, editors. *2nd Int. Workshop on Conditional Term Rewriting Systems (CTRS), Montreal, 1990*, number 516 in Lecture Notes in Computer Science, 1991.

[Kapur and Musser, 1986] Deepak Kapur and David R. Musser. Inductive reasoning with incomplete specifications. 1986. In [LICS, 1986, pp. 367–377].

[Kapur and Musser, 1987] Deepak Kapur and David R. Musser. Proof by consistency. *Artificial Intelligence*, 31:125–157, 1987.

[Kapur and Subramaniam, 1996] Deepak Kapur and Mahadevan Subramaniam. Automating induction over mutually recursive functions. 1996. In [Wirsing and Nivat, 1996, pp. 117–131].

[Kapur and Zhang, 1989] Deepak Kapur and Hantao Zhang. An overview of Rewrite Rule Laboratory (RRL). 1989. In [Dershowitz, 1989, pp. 559–563]. Journal version is [Kapur and Zhang, 1995].

[Kapur and Zhang, 1995] Deepak Kapur and Hantao Zhang. An overview of Rewrite Rule Laboratory (RRL). *Computers and Mathematics with Applications*, 29(2):91–114, 1995.

[Kapur, 1992] Deepak Kapur, editor. *11th Int. Conf. on Automated Deduction (CADE), Saratoga Springs (NY), 1992*, number 607 in Lecture Notes in Artificial Intelligence. Springer, 1992.

[Katz, 1998] Victor J. Katz. *A History of Mathematics: An Introduction*. Addison–Wesley, Reading (MA), 1998. 2nd edn..

[Kaufmann *et al.*, 2000a] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning:* ACL2 *Case Studies*. Number 4 in Advances in Formal Methods. Kluwer (Springer Science+Business Media), 2000. With a foreword from the series editor Mike Hinchey.

[Kaufmann *et al.*, 2000b] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Number 3 in Advances in Formal Methods. Kluwer (Springer Science+Business Media), 2000. With a foreword from the series editor Mike Hinchey.

[Knuth and Bendix, 1970] Donald E Knuth and Peter B. Bendix. Simple word problems in universal algebra. 1970. In [Leech, 1970, pp. 263–297].

[Kodratoff, 1988] Yves Kodratoff, editor. *Proc. 8th European Conf. on Artificial Intelligence (ECAI)*. Pitman Publ., London, 1988.

[Kott, 1986] Laurent Kott, editor. *13th Int. Colloquium on Automata, Languages and Programming (ICALP), Rennes (France)*, number 226 in Lecture Notes in Computer Science. Springer, 1986.

[Kowalski, 1974] Robert A. Kowalski. Predicate logic as a programming language. 1974. In [Rosenfeld, 1974, pp. 569–574].

[Kowalski, 1988] Robert A. Kowalski. The early years of logic programming. *Comm. ACM*, 31:38–43, 1988.

[Kreisel, 1965] Georg Kreisel. Mathematical logic. 1965. In [Saaty, 1965, Vol. III, pp. 95–195].

[Küchlin, 1989] Wolfgang Küchlin. Inductive completion by ground proof transformation. 1989. In [Aït-Kaci and Nivat, 1989, Vol. 2, pp. 211–244].

[Kühler and Wirth, 1996] Ulrich Kühler and Claus-Peter Wirth. *Conditional Equational Specifications of Data Types with Partial Operations for Inductive Theorem Proving*. SEKI-Report SR–1996–11 (ISSN 1437–4447). SEKI Publications, FB Informatik, Univ. Kaiserslautern, 1996. 24 pp., `http://wirth.bplaced.net/p/rta97`. Short version is [Kühler and Wirth, 1997].

[Kühler and Wirth, 1997] Ulrich Kühler and Claus-Peter Wirth. Conditional equational specifications of data types with partial operations for inductive theorem proving. 1997. In [Comon, 1997, pp. 38–52]. Extended version is [Kühler and Wirth, 1996].

[Kühler, 1991] Ulrich Kühler. Ein funktionaler und struktureller Vergleich verschiedener Induktionsbeweiser. (English translation of title: "A functional and structural comparsion of several inductive theorem-proving systems" (INKA, LP (Larch Prover), NQTHM, RRL, UNICOM)). vi+143 pp., Diplomarbeit (Master's thesis), FB Informatik, Univ. Kaiserslautern, 1991.

[Kühler, 2000] Ulrich Kühler. *A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations*. Infix, Akademische Verlagsgesellschaft Aka GmbH, Sankt Augustin, Berlin, 2000. PhD thesis, Univ. Kaiserslautern, ISBN 1586031287, `http://wirth.bplaced.net/p/kuehlerdiss`.

[Lambert, 1764] Johann Heinrich Lambert. *Neues Organon oder Gedanken über die Erforschung und Bezeichnung des Wahren und dessen Unterscheidung von Irrthum und Schein.* Johann Wendler, Leipzig, 1764. Vol. I (Dianoiologie oder die Lehre von den Gesetzen des Denkens, Alethiologie oder Lehre von der Wahrheit) (`http://books.google.de/books/about/Neues_Organon_oder_Gedanken_Uber_die_Erf.html?id=ViS3XCuJEw8C`) & Vol. II (Semiotik oder Lehre von der Bezeichnung der Gedanken und Dinge, Phänomenologie oder Lehre von dem Schein) (`http://books.google.de/books/about/Neues_Organon_oder_Gedanken_%C3%BCber_die_Er.html?id=X8UAAAAAcAAj`). Facsimile reprint by Georg Olms Verlag, Hildesheim (Germany), 1965, with a German introduction by Hans Werner Arndt.

[Lankford, 1980] Dallas S. Lankford. Some remarks on inductionless induction. Memo MTP-11, Math. Dept., Louisiana Tech. Univ., Ruston (LA), 1980.

[Lankford, 1981] Dallas S. Lankford. A simple explanation of inductionless induction. Memo MTP-14, Math. Dept., Louisiana Tech. Univ., Ruston (LA), 1981.

[Lassez and Plotkin, 1991] Jean-Louis Lassez and Gordon D. Plotkin, editors. *Computational Logic — Essays in Honor of J. Alan Robinson*. MIT Press, 1991.

[Leech, 1970] John Leech, editor. *Computational Word Problems in Abstract Algebra — Proc. of a Conf. held at Oxford, under the auspices of the Science Research Council, Atlas Computer Laboratory, 29th Aug. to 2nd Sept. 1967*. Pergamon Press, Oxford, 1970. With a foreword by J. Howlett.

[Leeuwen, 1990] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Sci.*. MIT Press, 1990.

[LICS, 1986] *Proc. 1st Annual IEEE Symposium on Logic In Computer Sci. (LICS), Cambridge (MA), 1986*. IEEE Press, 1986. `http://lii.rwth-aachen.de/lics/archive/1986`.

[LICS, 1988] *Proc. 3rd Annual IEEE Symposium on Logic In Computer Sci. (LICS), Edinburgh, 1988*. IEEE Press, 1988. `http://lii.rwth-aachen.de/lics/archive/1988`.

[LICS, 2007] *Proc. 22nd Annual IEEE Symposium on Logic In Computer Sci. (LICS), Wrocław (i.e. Breslau, Silesia), 2007*. IEEE Press, 2007. `http://lii.rwth-aachen.de/lics/archive/2007`.

[Löchner, 2006] Bernd Löchner. Things to know when implementing LPO. *Int. J. Artificial Intelligence Tools*, 15:53–79, 2006.

[Lusk and Overbeek, 1988] Ewing Lusk and Ross Overbeek, editors. *9th Int. Conf. on Automated Deduction (CADE), Argonne National Laboratory (IL), 1988*, number 310 in Lecture Notes in Artificial Intelligence. Springer, 1988.

[Mahoney, 1994] Michael Sean Mahoney. *The Mathematical Career of Pierre* de *Fermat 1601–1665*. Princeton Univ. Press, 1994. 2nd rev. edn. (1st edn. 1973).

[Marchisotto and Smith, 2007] Elena Anne Marchisotto and James T. Smith. *The Legacy of Mario Pieri in Geometry and Arithmetic*. Birkhäuser (Springer), 2007.

[Margaria, 1995] Tiziana Margaria, editor. *Kolloquium Programmiersprachen und Grundlagen der Programmierung*, 1995. Tech. Report MIP–9519, Univ. Passau.

[McCarthy *et al.*, 1965] John McCarthy, Paul W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, 1965.

[McRobbie and Slaney, 1996] Michael A. McRobbie and John K. Slaney, editors. *13th Int. Conf. on Automated Deduction (CADE), New Brunswick (NJ), 1996*, number 1104 in Lecture Notes in Artificial Intelligence. Springer, 1996.

[Melis *et al.*, 2008] Erica Melis, Andreas Meier, and Jörg Siekmann. Proof planning with multiple strategies. *Artificial Intelligence*, 172:656–684, 2008. Received May 2, 2006. Published online Nov. 22, 2007. `http://dx.doi.org/10.1016/j.artint.2007.11.004`.

[Meltzer and Michie, 1972] Bernard Meltzer and Donald Michie, editors. *Proceedings of the 7th Annual Machine Intelligence Workshop (Machine Intelligence 7), Edinburgh, 1971*. Univ. Edinburgh Press, 1972. `http://aitopics.org/sites/default/files/classic/Machine%20Intelligence%203/Machine%20Intelligence%20v3.pdf`.

[Meltzer, 1975] Bernard Meltzer. Department of A.I. – Univ. of Edinburgh. *ACM SIGART Bulletin*, 50:5, 1975.

[Michie, 1968] Donald Michie, editor. *Proceedings of the 3rd Annual Machine Intelligence Workshop (Machine Intelligence 3), Edinburgh, 1967*. Univ. Edinburgh Press, 1968. `http://aitopics.org/sites/default/files/classic/Machine%20Intelligence%203/Machine%20Intelligence%20v3.pdf`.

[Milner, 1972] Robin Milner. Logic for computable functions — description of a machine interpretation. Technical Report Memo AIM–169, STAN–CS–72–288, Dept. Computer Sci., Stanford University, 1972. `ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/72/288/CS-TR-72-288.pdf`.

[Moore *et al.*, 1998] J Strother Moore, Thomas Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47:913–926, 1998.

[Moore, 1973] J Strother Moore. *Computational Logic: Structure Sharing and Proof of Program Properties*. PhD thesis, Dept. Artificial Intelligence, Univ. Edinburgh, 1973. `http://hdl.handle.net/1842/2245`.

[Moore, 1975a] J Strother Moore. Introducing iteration into the Pure LISP Theorem Prover. Technical Report CSL 74–3, Xerox, Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto (CA), 1975. ii+37 pp., Received Dec. 1974, rev. March 1975. Short version is [Moore, 1975b].

[Moore, 1975b] J Strother Moore. Introducing iteration into the Pure LISP Theorem Prover. *IEEE Transactions on Software Engineering*, 1:328–338, 1975. `http://doi.ieeecomputersociety.org/10.1109/TSE.1975.6312857`. Long version is [Moore, 1975a].

[Moore, 1979] J Strother Moore. A mechanical proof of the termination of Takeuti's function. *Information Processing Letters*, 9:176–181, 1979. Received July 13, 1979. Rev. Sept. 5, 1979. `http://dx.doi.org/10.1016/0020-0190(79)90063-2`.

[Moore, 1981] J Strother Moore. Text editing primitives — the TXDT package. Technical Report CSL 81–2, Xerox, Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto (CA), 1981.

[Moore, 1989a] J Strother Moore. A mechanically verified language implementation. *J. Automated Reasoning*, 5:461–492, 1989.

[Moore, 1989b] J Strother Moore. System verification. *J. Automated Reasoning*, 5:409–410, 1989.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. 2001. In [DAC, 2001, pp. 530–535].

[Musser, 1980] David R. Musser. On proving inductive properties of abstract data types. 1980. In [Abrahams *et al.*, 1980, pp. 154–162]. `http://dl.acm.org/citation.cfm?id=567461`.

[Nilsson, 1973] Nils J. Nilsson, editor. *Proc. 3rd Int. Joint Conf. on Artificial Intelligence (IJCAI), Stanford (CA)*. Stanford Research Institute, Publications Dept., Stanford (CA), 1973. `http://ijcai.org/Past%20Proceedings/IJCAI-73/CONTENT/content.htm`.

[Padawitz, 1996] Peter Padawitz. Inductive theorem proving for design specifications. *J. Symbolic Computation*, 21:41–99, 1996.

[Padoa, 1913] Alessandro Padoa. La valeur et les rôles du principe d'induction mathématique. 1913. In [Hobson and Love, 1913, pp. 471–479].

[Pascal, 1954] Blaise Pascal. *Œuvres Complètes*. Gallimard, Paris, 1954. Ed. by Jacques Chevalier.

[Paulson, 1996] Lawrence C. Paulson. ML *for the Working Programmer*. Cambridge Univ. Press, 1996. 2nd edn. (1st edn. 1991).

[Peano, 1889] Guiseppe Peano. *Arithmetices principia, novo methodo exposita*. Fratelli Bocca, Torino (i.e. Turin, Italy), 1889.

[Péter, 1951] Rósza Péter. *Rekursive Funktionen*. Akad. Kiadó, Budapest, 1951.

[Pieri, 1908] Mario Pieri. Sopra gli assiomi aritmetici. *Il Bollettino delle seduta della Accademia Gioenia di Scienze Naturali in Catania*, Series 2, 1–2:26–30, 1908. Written Dec. 1907. Received Jan. 8, 1908. English translation *On the Axioms of Arithmetic* in [Marchisotto and Smith, 2007, § 4.2, pp. 308–313].

[Plotkin *et al.*, 2000] Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors. *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. MIT Press, 2000.

[Presburger, 1930] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków krajów słowianskich, Warszawa 1929 (Comptes-rendus du 1re Congrès des Mathématiciens des Pays Slaves, Varsovie 1929)*, pages 92–101+395, 1930. Remarks and English translation in [Stansifer, 1984].

[Protzen, 1994] Martin Protzen. Lazy generation of induction hypotheses. 1994. In [Bundy, 1994, pp. 42–56].

[Protzen, 1995] Martin Protzen. *Lazy Generation of Induction Hypotheses and Patching Faulty Conjectures*. Infix, Akademische Verlagsgesellschaft Aka GmbH, Sankt Augustin, Berlin, 1995. PhD thesis.

[Protzen, 1996] Martin Protzen. Patching faulty conjectures. 1996. In [McRobbie and Slaney, 1996, pp. 77–91].

[Rabinovitch, 1970] Nachum L. Rabinovitch. Rabbi Levi ben Gerson and the origins of mathematical induction. *Archive for History of Exact Sciences*, 6:237–248, 1970. Received Jan. 12, 1970.

[Reddy, 1977] Ray Reddy, editor. *Proc. 5th Int. Joint Conf. on Artificial Intelligence (IJCAI), Cambridge (MA)*. Dept. of Computer Sci., Carnegie Mellon Univ., Cambridge (MA), 1977. `http://ijcai.org/Past%20Proceedings`.

[Reddy, 1990] Uday S. Reddy. Term rewriting induction. 1990. [Stickel, 1990, pp. 162–177].

[Riazanov and Voronkov, 2001] Alexander Riazanov and Andrei Voronkov. Vampire 1.1 (system description). 2001. In [Gore *et al.*, 2001, pp. 376–380].

[Robinson and Voronkow, 2001] J. Alan Robinson and Andrei Voronkow, editors. *Handbook of Automated Reasoning*. Elsevier, 2001.

[Rosenfeld, 1974] Jack L. Rosenfeld, editor. *Proc. of the Congress of the Int. Federation for Information Processing (IFIP), Stockholm (Sweden), Aug. 5–10, 1974*. North-Holland (Elsevier), 1974.

[Rubin and Rubin, 1985] Herman Rubin and Jean E. Rubin. *Equivalents of the Axiom of Choice*. North-Holland (Elsevier), 1985. 2nd rev. edn. (1st edn. 1963).

[Rusinowitch and Remy, 1993] Michaël Rusinowitch and Jean-Luc Remy, editors. *3rd Int. Workshop on Conditional Term Rewriting Systems (CTRS), Pont-à-Mousson (France), 1992*, number 656 in Lecture Notes in Computer Science, 1993.

[Russinoff, 1998] David M. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, 1998.

[Saaty, 1965] T. L. Saaty, editor. *Lectures on Modern Mathematics*. John Wiley & Sons, New York, 1965.

[Schmidt-Samoa, 2006a] Tobias Schmidt-Samoa. An even closer integration of linear arithmetic into inductive theorem proving. *Electronic Notes in Theoretical Computer Sci.*, 151:3–20, 2006. `http://wirth.bplaced.net/p/evencloser`, `http://dx.doi.org/10.1016/j.entcs.2005.11.020`.

[Schmidt-Samoa, 2006b] Tobias Schmidt-Samoa. *Flexible Heuristic Control for Combining Automation and User-Interaction in Inductive Theorem Proving.* PhD thesis, Univ. Kaiserslautern, 2006. `http://wirth.bplaced.net/p/samoadiss`.

[Schmidt-Samoa, 2006c] Tobias Schmidt-Samoa. Flexible heuristics for simplification with conditional lemmas by marking formulas as forbidden, mandatory, obligatory, and generous. *J. Applied Non-Classical Logics*, 16:209–239, 2006. `http://dx.doi.org/10.3166/jancl.16.208-239`.

[Schoenfield, 1967] Joseph R. Schoenfield. *Mathematical Logic.* Addison–Wesley, Reading (MA), 1967.

[Scott, 1993] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Sci.*, 121:411–440, 1993. Annotated version of a manuscript from the year 1969. `www.cs.cmu.edu/~kw/scans/scott93tcs.pdf`.

[Shankar, 1986] Natarajan Shankar. *Proof-checking Metamathematics.* PhD thesis, The University of Texas at Austin, 1986. Thoroughly revised version is [Shankar, 1994].

[Shankar, 1988] Natarajan Shankar. A mechanical proof of the Church–Rosser theorem. *J. of the ACM*, 35:475–522, 1988. Received May 1985, rev. Aug. 1987. See also Chapter 6 in [Shankar, 1994].

[Shankar, 1994] Natarajan Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge Univ. Press, 1994. Originally published as [Shankar, 1986]. Paperback reprint 1997.

[Siekmann, 1986] Jörg Siekmann, editor. *8th Int. Conf. on Automated Deduction (CADE), Oxford, 1986*, number 230 in Lecture Notes in Artificial Intelligence. Springer, 1986.

[Sridharan, 1989] N. S. Sridharan, editor. *Proc. 11th Int. Joint Conf. on Artificial Intelligence (IJCAI), Detroit (MI).* Morgan Kaufmann, Los Altos (CA) (Elsevier), 1989. `http://ijcai.org/Past%20Proceedings`.

[Stansifer, 1984] Ryan Stansifer. Presburger's Article on Integer Arithmetic: Remarks and Translation. Technical Report TR 84–639, Dept. of Computer Sci., Cornell Univ., Ithaca (NY), 1984. `http://hdl.handle.net/1813/6478`.

[Steele, 1990] Guy L. Steele, Jr.. Common Lisp — *The Language.* Digital Press (Elsevier), 1990. 2nd edn. (1st edn. 1984).

[Steinbach, 1988] Joachim Steinbach. *Term Orderings With Status.* SEKI-Report SR–88–12 (ISSN 1437–4447). SEKI Publications, FB Informatik, Univ. Kaiserslautern, 1988. 57 pp., `http://wirth.bplaced.net/SEKI/welcome.html#SR-88-12`.

[Steinbach, 1995] Joachim Steinbach. Simplification orderings — history of results. *Fundamenta Informaticae*, 24:47–87, 1995.

[Stevens, 1988] Andrew Stevens. *A Rational Reconstruction of Boyer and Moore's Technique for Constructing Induction Formulas.* 1988. DAI Research Paper No. 360, Dept. Artificial Intelligence, Univ. Edinburgh. Also in [Kodratoff, 1988, pp. 565–570].

[Stickel, 1990] Mark E. Stickel, editor. *10th Int. Conf. on Automated Deduction (CADE), Kaiserslautern (Germany), 1990*, number 449 in Lecture Notes in Artificial Intelligence. Springer, 1990.

[Stoyan, 1985] Herbert Stoyan, editor. *9th German Workshop on Artificial Intelligence (GWAI), Dassel (Germany), 1985*, number 118 in Informatik-Fachberichte. Springer, 1985.

[Toyama, 1988] Yoshihito Toyama. Commutativity of term rewriting systems. 1988. In [Fuchi and Kott, 1988, pp. 393–407]. Also in [Toyama, 1990].

[Toyama, 1990] Yoshihito Toyama. *Term Rewriting Systems and the Church–Rosser Property.* PhD thesis, Tohoku Univ. / Nippon Telegraph and Telephone Corporation, 1990.

[Unguru, 1991] Sabetai Unguru. Greek mathematics and mathematical induction. *Physis*, XXVIII(2):273–289, 1991.

[Verma, 2005?] Shamit Verma. Interview with Charles Simonyi. WWW only: `http://www.shamit.org/charles_simonyi.htm`, 2005?

[Voicu and Li, 2009] Răzvan Voicu and Mengran Li. *Descente Infinie* proofs in Coq. 2009. In [Herbelin, 2009, pp. 73–84].

[Voronkov, 1992] Andrei Voronkov, editor. *Proc. 3rd Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, number 624 in Lecture Notes in Artificial Intelligence. Springer, 1992.

[Voronkov, 2002] Andrei Voronkov, editor. *18th Int. Conf. on Automated Deduction (CADE), København (Denmark), 2002*, number 2392 in Lecture Notes in Artificial Intelligence. Springer, 2002.

[Walther, 1988] Christoph Walther. Argument-bounded algorithms as a basis for automated termination proofs. 1988. In [Lusk and Overbeek, 1988, pp. 601–622].

[Walther, 1992] Christoph Walther. Computing induction axioms. 1992. In [Voronkov, 1992, pp. 381–392].

[Walther, 1993] Christoph Walther. Combining induction axioms by machine. 1993. In [Bajscy, 1993, pp. 95–101].

[Walther, 1994a] Christoph Walther. Mathematical induction. 1994. In [Gabbay *et al.*, 1994, pp. 127–228].

[Walther, 1994b] Christoph Walther. On proving termination of algorithms by machine. *Artificial Intelligence*, 71:101–157, 1994.

[Wirsing and Nivat, 1996] Martin Wirsing and Maurice Nivat, editors. *Proc. 5th Int. Conf. on Algebraic Methodology and Software Technology (AMAST), München (Germany), 1996*, number 1101 in Lecture Notes in Computer Science. Springer, 1996.

[Wirth and Becker, 1995] Claus-Peter Wirth and Klaus Becker. Abstract notions and inference systems for proofs by mathematical induction. 1995. In [Dershowitz and Lindenstrauss, 1995, pp. 353–373].

[Wirth and Gramlich, 1994a] Claus-Peter Wirth and Bernhard Gramlich. A constructor-based approach to positive/negative-conditional equational specifications. *J. Symbolic Computation*, 17:51–90, 1994. `http://dx.doi.org/10.1006/jsco.1994.1004`, `http://wirth.bplaced.net/p/jsc94`.

[Wirth and Gramlich, 1994b] Claus-Peter Wirth and Bernhard Gramlich. On notions of inductive validity for first-order equational clauses. 1994. In [Bundy, 1994, pp. 162–176], `http://wirth.bplaced.net/p/cade94`.

[Wirth and Kühler, 1995] Claus-Peter Wirth and Ulrich Kühler. *Inductive Theorem Proving in Theories Specified by Positive/Negative-Conditional Equations.* SEKI-Report SR–95–15 (ISSN 1437–4447). SEKI Publications, Univ. Kaiserslautern, 1995. iv+126 pp..

[Wirth *et al.*, 1993] Claus-Peter Wirth, Bernhard Gramlich, Ulrich Kühler, and Horst Prote. *Constructor-Based Inductive Validity in Positive/Negative-Conditional Equational Specifications.* SEKI-Report SR–93–05 (SFB) (ISSN 1437–4447). SEKI Publications, FB Informatik, Univ. Kaiserslautern, 1993. iv+58 pp., `http://wirth.bplaced.net/SEKI/welcome.html#SR-93-05`. Rev. extd. edn. of 1st part is [Wirth and Gramlich, 1994a], rev. edn. of 2nd part is [Wirth and Gramlich, 1994b].

[Wirth *et al.*, 2009] Claus-Peter Wirth, Jörg Siekmann, Christoph Benzmüller, and Serge Autexier. Jacques Herbrand: Life, logic, and automated deduction. 2009. In [Gabbay and Woods, 2004ff., Vol. 5: Logic from Russell to Church, pp. 195–254].

[Wirth, 1971] Niklaus Wirth. The programming language PASCAL. *Acta Informatica*, 1:35–63, 1971.

[Wirth, 1991] Claus-Peter Wirth. Inductive theorem proving in theories specified by positive/negative-conditional equations. Diplomarbeit (Master's thesis), FB Informatik, Univ. Kaiserslautern, 1991.

[Wirth, 1997] Claus-Peter Wirth. *Positive/Negative-Conditional Equations: A Constructor-Based Framework for Specification and Inductive Theorem Proving*, volume 31 of *Schriftenreihe Forschungsergebnisse zur Informatik.* Verlag Dr. Kovač, Hamburg, 1997. PhD thesis, Univ. Kaiserslautern, ISBN 386064551X, `http://wirth.bplaced.net/p/diss`.

[Wirth, 2004] Claus-Peter Wirth. Descente Infinie + Deduction. *Logic J. of the IGPL*, 12:1–96, 2004. `http://wirth.bplaced.net/p/d`.

[Wirth, 2005a] Claus-Peter Wirth. History and future of implicit and inductionless induction: Beware the old jade and the zombie! 2005. In [Hutter and Stephan, 2005, pp. 192–203], `http://wirth.bplaced.net/p/zombie`.

[Wirth, 2005b] Claus-Peter Wirth. *Syntactic Confluence Criteria for Positive/Negative-Conditional Term Rewriting Systems.* SEKI-Report SR–95–09 (ISSN 1437–4447). SEKI Publications, Univ. Kaiserslautern, 2005. Rev. edn. Oct. 2005 (1st edn. 1995), ii+188 pp., `http://arxiv.org/abs/0902.3614`.

[Wirth, 2006] Claus-Peter Wirth. lim+, $\delta^+$, *and Non-Permutability of* $\beta$*-Steps*. SEKI-Report SR–2005–01 (ISSN 1437–4447). SEKI Publications, Saarland Univ., 2006. Rev. edn. July 2006 (1st edn. 2005), ii+36 pp., `http://arxiv.org/abs/0902.3635`. Thoroughly improved version is [Wirth, 2012b].

[Wirth, 2009] Claus-Peter Wirth. Shallow confluence of conditional term rewriting systems. *J. Symbolic Computation*, 44:69–98, 2009. `http://dx.doi.org/10.1016/j.jsc.2008.05.005`.

[Wirth, 2010a] Claus-Peter Wirth. *Progress in Computer-Assisted Inductive Theorem Proving by Human-Orientedness and Descente Infinie?* SEKI-Working-Paper SWP–2006–01 (ISSN 1860–5931). SEKI Publications, Saarland Univ., 2010. Rev. edn. Dec 2010 (1st edn. 2006), ii+36 pp., `http://arxiv.org/abs/0902.3294`.

[Wirth, 2010b] Claus-Peter Wirth. *A Self-Contained and Easily Accessible Discussion of the Method of Descente Infinie and Fermat's Only Explicitly Known Proof by Descente Infinie*. SEKI-Working-Paper SWP–2006–02 (ISSN 1860–5931). SEKI Publications, DFKI Bremen GmbH, Safe and Secure Cognitive Systems, Cartesium, Enrique Schmidt Str. 5, D–28359 Bremen, Germany, 2010. Rev. ed. Dec. 2010, ii+36 pp., `http://arxiv.org/abs/0902.3623`.

[Wirth, 2012a] Claus-Peter Wirth. Herbrand's Fundamental Theorem in the eyes of Jean van Heijenoort. *Logica Universalis*, 6:485–520, 2012. Received Jan. 12, 2012. Published online June 22, 2012, `http://dx.doi.org/10.1007/s11787-012-0056-7`.

[Wirth, 2012b] Claus-Peter Wirth. lim +, $\delta^+$, and Non-Permutability of $\beta$-Steps. *J. Symbolic Computation*, 47:1109–1135, 2012. Received Jan. 18, 2011. Published online July 15, 2011, `http://dx.doi.org/10.1016/j.jsc.2011.12.035`. More funny version is [Wirth, 2006].

[Wirth, 2012c] Claus-Peter Wirth. Human-oriented inductive theorem proving by descente infinie — a manifesto. *Logic J. of the IGPL*, 20:1046–1063, 2012. Received July 11, 2011. Published online March 12, 2012, `http://dx.doi.org/10.1093/jigpal/jzr048`.

[Wirth, 2012d] Claus-Peter Wirth. Unpublished Interview of Robert S. Boyer and J Strother Moore at Boyer's place in Austin (TX) on Thursday, Oct. 7. 2012.

[Wirth, 2013] Claus-Peter Wirth. *A Simplified and Improved Free-Variable Framework for Hilbert's epsilon as an Operator of Indefinite Committed Choice*. SEKI Report SR–2011–01 (ISSN 1437–4447). SEKI Publications, DFKI Bremen GmbH, Safe and Secure Cognitive Systems, Cartesium, Enrique Schmidt Str. 5, D–28359 Bremen, Germany, 2013. Rev. edn. Jan. 2013 (1st edn. 2011), ii+65 pp., `http://arxiv.org/abs/1104.2444`.

[Wolff, 1728] Christian Wolff. *Philosophia rationalis sive Logica, methodo scientifica pertractata et ad usum scientiarium atque vitae aptata*. Rengerische Buchhandlung, Frankfurt am Main & Leipzig, 1728. 1st edn..

[Wolff, 1740] Christian Wolff. *Philosophia rationalis sive Logica, methodo scientifica pertractata et ad usum scientiarium atque vitae aptata*. Rengerische Buchhandlung, Frankfurt am Main & Leipzig, 1740. 3rd extd. edn. of [Wolff, 1728]. Facsimile reprint by Georg Olms Verlag, Hildesheim (Germany), 1983, with a French introduction by Jean École.

[Yeh and Ramamoorthy, 1976] Raymond T. Yeh and C. V. Ramamoorthy, editors. *Proc. 2nd Int. Conf. on Software Engineering, San Francisco (CA), Oct. 13–15, 1976*. IEEE Computer Sci. Press, Los Alamitos (CA), 1976. `http://dl.acm.org/citation.cfm?id=800253`.

[Young, 1989] William D. Young. A mechanically verified code generator. *J. Automated Reasoning*, 5:493–518, 1989.

[Zhang *et al.*, 1988] Hantao Zhang, Deepak Kapur, and Mukkai S. Krishnamoorthy. A mechanizable induction principle for equational specifications. 1988. In [Lusk and Overbeek, 1988, pp. 162–181].

[Zygmunt, 1991] Jan Zygmunt. Mojżesz Presburger: Life and work. *History and Philosophy of Logic*, 12:211–223, 1991.

# INDEX