THE AUTOMATION OF MATHEMATICAL INDUCTION[*]

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, neither for the *purely equational fragment* nor for *propositional logic*.[2] Moreover, automation of theorem proving for *higher-order logic* is making progress towards general usefulness just during the last ten years.[3]

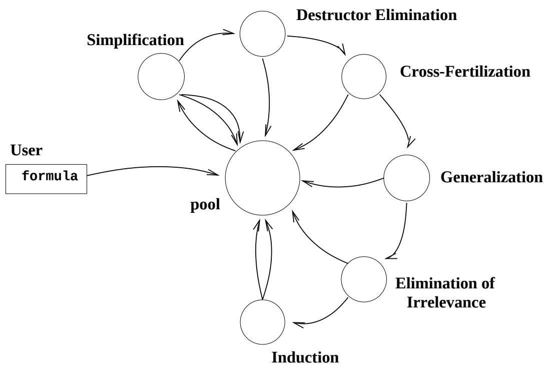

Figure 1. The Boyer–Moore Waterfall

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

[*]Second Readers: Somebody from the history of math & logic. I wanna ask Wilfried Sieg. OK?

[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 still NP-complete), a breakthrough toward industrial strength was the SAT solver CHAFF, cf. e.g. [Moskewicz *et al.*, 2001].

[3]One of the driving forces in the automaton of higher-order theorem proving is the system LEO-II, cf. e.g. [Benzmüller *et al.*, 2008].

In this context, it is most astonishing that for the field of quantifier-free first-order *inductive* theorem proving based on recursive functions, the whole development — from the scratch to general usefulness — took place within the 1970s.

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

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, and most of the crucial steps and their synergetic combination in the now famous "waterfall" (cf. Figure 1) are already implemented in their "PURE LISP THEOREM PROVER", presented at IJCAI in Stanford (CA) in August 1973,[4] and documented in Moore's PhD thesis [1973], defended in November 1973.

At that time, Boyer and Moore were both members of the Metamathematics Unit of the University of Edinburgh.[5]

Boyer and Moore had met for the first time in August 1971 in Edinburgh, and had worked together on structure sharing in resolution theorem proving, for which the inventor of resolution, J. Alan Robinson, invented and granted to them the "1971 Programming Prize" on December 17, 1971, half jokingly, half seriously.[6] In spite of this experience, the original plan to implement structure sharing in the PURE LISP THEOREM PROVER was dropped at some point.

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

[5]The Metamathematics Unit of the University of Edinburgh was renamed into "Dept. of Computational Logic" late in 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, ?)	(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 & CL)	(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	(1971–1974, 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: [Kowalski, 1988], [Meltzer, 1975], etc.)

Boyer and Moore's achievements become even more surprising by the fact that they had to work with inferior computing machinery and that they did all the programming themselves.

Working with computers was pretty troublesome and difficult at that time: The storage medium was paper tapes, and the editor programs of the time still simulated paper tape editing.

Moreover, the Metamathematics Unit did not even have a PDP–10, which was the top model of the time, but only an ICL–4130, which had only 64 kByte core memory (RAM). A funny feature was the loudspeaker connect to this core memory, which, however, did not produce a regular sound when the PURE LISP THEOREM PROVER was running (unless during garbage collection). An irregular sound was meant to indicate that it was not in a constant loop, but making "progress".

What made the situation worse is that Boyer and Moore were granted sufficient computation time only at nights, for which Boyer had to have a huge teletype terminal in his tiny house in Edinburgh.

It is easy to imagine what this meant for the wives of Boyer and Moore who lived with them in Edinburgh.

> "The PDP–10, the PDP–10,
> I've sung it before and I'll sing it again.
> I hav'nt seen my man since I don't know when —
> Oh, I lost my husband to a PDP–10.
>
> If he'd a woman, the fight would be fair.
> If he were out drinking, his liquor I'd share.
> But the odds are against me, you know what I mean —
> Even John Henry couldn't beat a machine.
>
> Guess I'll give up, I know when I am beat.
> Sure hate to say it, but this is defeat.
> There's just one more problem that I've got to face —
> Can you name a machine in an adultery case?"

The Boyers and Moores used to sing this text to the melody of the popular American folksong "John Henry", a legendary African-American railroad worker who died in a competition with a steam-powered hammer. The new text — which paradigmatically describes a situation well-known to probably all spouses of scientists until today — was written by Anne Boyer at a time when actually no PDP–10 was available to Boyer and Moore; but their situation had improved in this aspect when she published it [1980]. Needless to say that Anne and Bob Boyer are still living together today.

[6]The document, handwritten by J. Alan Robinson (*1930?) actually says:

> "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'."

2 METHOD OF PROCEDURE AND PRESENTATION

Contrary to the excellent handbook articles [Walther, 1994] and [Bundy, 1999] on the *automation of mathematical induction*, our focus in this article is neither on current standards, nor on the engineering and research problems of the field, but on its history. Consequently, this text should be accessible to an audience that goes beyond the technical experts and programmers of the day, should use common mathematical language and representation, should focus on the global and eternal ideas and their developments, and should paradigmatically display the *historically most significant achievements.*

Because these achievements manifest themselves in the line of the Boyer–Moore theorem provers, we nevertheless have to confront 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,[7] 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. Here we do not necessarily refer to the implementation language of these software systems, but actually to the logic language used both for proof representation and for communication with the user.

For the first system in this line of development, Boyer and Moore had the free choice, but — in 1972 — there was actually no alternative to a logic based on LISP, because 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.

Beside LISP, a candidate to be considered today would be the functional programming language HASKELL.[8] HASKELL, however, was not available in 1972. And still today, LISP is to be preferred to HASKELL as the logic of an inductive theorem prover because of LISP's 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 their *constructor variables*[9] and their *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

[7] Cf. [McCarthy *et al.*, 1965].

[8] Cf. e.g. [Hudlak *et al.*, 1999].

[9] See § 4.4 of this article.

which its development is rooted,[10] were available in 1972. And still today, the applicative subset of COMMON LISP as part of ACL2 (= $(\mathrm{ACL})^2$ = $\underline{\text{A}}$ $\underline{\text{C}}$omputational $\underline{\text{L}}$ogic for $\underline{\text{A}}$pplicative $\underline{\text{C}}$ommon $\underline{\text{L}}$ISP) is again to preferred to these positive/negative-conditional rewrite systems under the aspect 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 successfully — the theorem prover can defer the interpretation of ground terms to the analogous interpretation in its own programming language.

For many of our illustrative examples in this article, however, we will use the higher flexibility and conceptual adequacy of positive/negative-conditional rewrite systems, especially because they are so close to logic that we can dispense their semantics to the readers intuition, and because their typed (many-sorted) approach admits to present the formulas in a form that is much easier to grasp for human readers than the corresponding sugar-free LISP notation with its additional explicit type restrictions. Nevertheless, small LISP expression cannot be avoided if we want to get close to an implementation or if we want to show the advantages of LISP's untypedness.[11] The readers interested in these aspects, 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 and $t_1, \ldots, t_n$ are LISP terms.

Another good reason for us to try to avoid LISP notation is the following: We want to make it most obvious that the great achievements of the Boyer–Moore theorem provers are not limited to their LISP logic and would well survive in a better world, where — based on the computing technology of the future — positive/negative-conditional rewriting is not too inefficient for the relevant applications anymore.

For the same reason, we also prefer examples from arithmetic to examples from list theory, which could be considered to profit especially from the logic of LISP. The reader can find the famous examples from list theory in almost any other publication on the subject.[12]

[10]The general theory on which the rewrite systems of [Wirth and Gramlich, 1994a] and [Wirth, 1991; 2009] is rooted is documented in [Dershowitz and Jouannaud, 1990]. 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 at the breath-taking 1st Int. Workshop on Conditional Term Rewriting Systems (CTRS), Orsay (France), 1987; cf. [Kaplan and Jouannaud, 1988].

[11]See e.g. of the advantages of the untyped and type-restriction-free declaration of the shell `CONS` in § 5.3.

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

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.

We have to add here something like an abstract and an organization. Note that the handbook does not admit a table of contents.

3 MATHEMATICAL INDUCTION

In this section, we introduce mathematical induction with its rich history since the 6th century B.C., and clarify the difference between *structural induction* and *Noetherian induction* with its traditional variant called *descente infinie*.

According to Aristotle, *induction* means to go from the special to the general, in particular to obtain *general laws* from special cases, which plays a major rôle in the generation of conjectures in mathematics and the natural sciences. Modern scientists design experiments to falsify such a 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 of Alexandria, mathematicians accept a conjectured mathematical law 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 all information we can obtain by it, is implicitly contained in the initial judgments, though we can hardly be aware of all deducible consequences in advance.

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[13] — the term "induction" was introduced and became standard in English and German mathematics. 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 19th century, mathematical induction has been a standard method of every working mathematician at all times. Hippasus of Metapontum (Italy) (ca. 550 B.C.) is reported[14] to have proved the irrationality of the golden number by a form of mathematical induction, which later was 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.).[15] In the famous "Elements" of Euclid [ca. 300 B.C.], we find several applications of *descente infinie* and structural induction.[16] 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).[17] Furthermore, structural induction was used by Francesco Maurolico (Messina) (1494–1575),[18] and by Blaise Pascal (1623–1662).[19] After an absence of more than one millennium, *descente infinie* was reinvented by Pierre Fermat (1607?–1665).[20]

[13]Cf. [Cajori, 1918].

[14]Cf. [Fritz, 1945].

[15]Cf. [Acerbi, 2000].

[16]An example for *descente infinie* is Proposition 31 of Vol. VII of the Elements, and an example for structural induction is Proposition 8 of Vol. IX, cf. [Wirth, 2010b, § 2.4].

[17]Cf. [Katz, 1998].

[18]Cf. [Bussey, 1917].

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

[20]See [Barner, 2001] for the correction on Fermat's year of birth as compared to the wrong date in the title of [Mahoney, 1994]. The best-documented example of a proof by *descente infinie* of one of Fermat's outstanding results in number theory is the proof of the following theorem: *The area of a Pythagorean triangle with positive integer side lengths is not the square of an integer*; cf. [Wirth, 2010b].

3.1 Well-foundedness and Termination

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

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

Moreover, $<$ is an (irreflexive) *ordering* if it is irreflexive and transitive. There is not much difference between a well-founded relation and a well-founded ordering:[21]

LEMMA 1. *A relation is well-founded if and only if its transitive closure 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;[22] indeed, it admits 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 this difference is relevant in our context here, where we cannot take the Axiom of Choice for granted because it can be seen as a very strong induction axiom.[23]

3.2 The Theorem of Noetherian Induction

In its modern standard meaning, the method of mathematical induction can easily be 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 $u < v$.*

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

[22] See [Wirth, 2004, § 2.1.2, p. 18] for the equivalence to the Principle of Dependent Choice, which is found in [Rubin and Rubin, 1985, p. 19] and analyzed in detail as Form 43 (p. 30) in [Howard and Rubin, 1998].

[23] Cf. [Geser, 1995].

Writing "$\mathsf{Wellf}(<)$" again for "$<$ is well-founded", we can formalize the Theorem of Noetherian Induction (N) as follows:[24]

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

The term "Noetherian induction" is a tribute to the famous German female mathematician Emmy Noether (1882–1935). It occurs as the "Generalized principle of induction (Noetherian induction)" in [Cohn, 1965, p. 20]. It also occurs in Proposition 7 ("Principle of Noetherian Induction") (p. 190) of § 6.5 in [Bourbaki, 1968a, Chapter III], which is a translation of the French original in its second edition [Bourbaki, 1967], where it occurs in Proposition 7 ("principe de récurrence nœthérienne")[25] of § 6.5. We do not know whether the today most common term "Noetherian Induction" occurred already before 1965;[26] in particular, it does not occur in the first French edition [Bourbaki, 1956] of [Bourbaki, 1967].[27]

3.3 The Natural Numbers

A most familiar field of application of induction is the domain of the natural numbers 0, 1, 2, ... Let us formalize the natural numbers with the help of two constructors: the constant symbol

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

for zero, and the function symbol

$$\mathsf{s} : \mathsf{nat} \rightarrow \mathsf{nat}$$

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

[24]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.

[25]The peculiar French spelling "Nœthér" in "nœthérienne" tries to imitate the German pronunciation of "Noether", where the "oe" is to be pronounced neither as a long "o" (which would be 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 Noether nor her father, the mathematics professor Max Noether (1844–1921), ligated the "oe" in their name, which occurs, however, in some of their official German documents.

[26]Even in 1967, "Noetherian Induction" was not generally used as a name for the Theorem of Noetherian Induction: For instance, in [Schoenfield, 1967, p. 205], this theorem (instantiated with the ordering of the natural numbers) is called the *principle of complete induction*, which is a most confusing name that should be avoided because "complete induction" looks like the straightforward translation of the German "vollständige Induction", which traditionally means structural induction (cf. Note 28), and which nowadays has become a very vague notion that is best translated as "mathematical induction", as done already in [Heijenoort, 1971, p.130] and as it is standard today, cf. [Hilbert and Bernays, 2011, Note 23.4].

[27]Indeed, the main text of § 6.5 in the first edition [Bourbaki, 1956] ends (on Page 98) three lines before the text of Proposition 7 begins in the second edition [Bourbaki, 1967] (on Page 76 of § 6.5).

After the definition ($\mathsf{Wellf}(<)$) and the theorem (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 on* $\mathsf{0}$ *and* s holds; formally:

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

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

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

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

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” (“complete induction”).[28]

Now we can go on by defining — in two equivalent ways — the destructor function $\mathsf{p}:\ \mathsf{nat}\rightarrow\mathsf{nat}$, which returns the predecessor of a positive natural number, first in *constructor style* (where constructor terms may occur as arguments of the function being defined) in ($\mathsf{p1}$), and then in *destructor style* (where only variables may occur as arguments) in ($\mathsf{p1}'$). Moreover, we define some recursive functions over the natural numbers, such as addition and multiplication $+, *:\ \mathsf{nat},\mathsf{nat}\rightarrow\mathsf{nat}$ (for which we use infix notation), the irreflexive ordering of the natural numbers $\mathsf{less}:\ \mathsf{nat},\mathsf{nat}\rightarrow\mathsf{bool}$ (see § 3.4 for the data type bool of Boolean values), and the Ackermann function $\mathsf{ack}:\ \mathsf{nat},\mathsf{nat}\rightarrow\mathsf{nat}$:[29]

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

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

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

$$\begin{array}{lll}
(\mathsf{less1}) & \mathsf{less}(x,\mathsf{0}) & =\mathsf{false}\\
(\mathsf{less2}) & \mathsf{less}(\mathsf{0},\mathsf{s}(y)) & =\mathsf{true}\\
(\mathsf{less3}) & \mathsf{less}(\mathsf{s}(x),\mathsf{s}(y)) & =\mathsf{less}(x,y)
\end{array}$$

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

[28]The first occurrence of the name “vollständige Induction” with the meaning of mathematical induction seems to be on Page 46f. in [Fries, 1822]. See also Note 26.

[29]Rósza Péter (1905–1977) published a simplified version [1951] of the first recursive, but not primitive recursive function developed by Wilhelm Ackermann (1896–1962) [Ackermann, 1928]. What today is simply called *the Ackermann function* is actually Péter’s version.

An immediate consequence of the axiom $(\mathsf{nat1})$ and the definition $(\mathsf{p1})$ is the lemma $(\mathsf{s1})$ and its flattened version $(\mathsf{s1}')$

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

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

On the basis of these axioms we can most easily show

$(\mathsf{less4})$ $\quad \mathsf{less}(x, \mathsf{s}(x)) \quad = \mathsf{true}$

$(\mathsf{less5})$ $\quad \mathsf{less}(x, \mathsf{s}(x+y)) = \mathsf{true}$

by *structural induction on* x, i.e. by taking the predicate variable P in the Axiom of Structural Induction (S) to be $\lambda x.\ (\mathsf{less}(x, \mathsf{s}(x)) = \mathsf{true})$ in case of $(\mathsf{less4})$, and $\lambda x.\ (\mathsf{less}(x, \mathsf{s}(x+y)) = \mathsf{true})$ in case of $(\mathsf{less4})$. Moreover — to see the necessity of doing induction upon several variables in parallel — we will do the more complicated proof of the *strengthened transitivity of the irreflexive ordering of the natural numbers*, i.e. of

$(\mathsf{less7})$ $\quad \mathsf{less}(\mathsf{s}(x), z) = \mathsf{true} \;\Leftarrow\; \big(\ \mathsf{less}(x, y) = \mathsf{true} \;\wedge\; \mathsf{less}(y, z) = \mathsf{true}\ \big)$

twice, once in once in Example 3 in § 3.6, and again in Example 9 in § 5.2.6. Twice we also prove the commutativity lemma

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

once in Example 2 in § 3.6, and again in Example 4 in § 3.7.1. Moreover, in Example 5 in § 3.8, we prove the most simple fact about the Ackermann function:

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

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 $\mathsf{Wellf}(\lambda x, y.\ (\mathsf{less}(x, y) = \mathsf{true}))$.

Now the natural numbers can be specified up to isomorphism either by[30]

- $(\mathsf{nat2})$, $(\mathsf{nat3})$, and (S) — following Guiseppe Peano (1858–1932),

or else by

- $(\mathsf{nat1})$ and $\mathsf{Wellf}(\lambda x, y.\ (\mathsf{s}(x) = y))$ — following Mario Pieri (1860–1913).[31]

[30]Cf. [Wirth, 2004, § 1.1.2].

[31]Pieri [1908] stated these axioms informal and showed their equivalence to the version of the Peano axioms of Alessandro Padoa (1868–1937). For a discussion and an English translation see [Marchisotto and Smith, 2007]. Pieri has also a version where instead of the symbol $\mathsf{0}$, there is only the statement that there is a natural number, and where $(\mathsf{nat1})$ is replaced with the weaker $\neg\exists y_0.\ (x_0 = \mathsf{s}(y_0)) \wedge \neg\exists y_1.\ (x_1 = \mathsf{s}(y_1)) \;\Rightarrow\; x_0 = x_1$. Non-standard natural numbers cannot exist in Pieri's specification: For every natural number x we can from the set of all elements that can be reached from x by the inverse of the successor relation. By well-foundedness, this set contains a minimal element, which is $\mathsf{0}$; so $x = \mathsf{s}^n(\mathsf{0})$ for some meta-level natural number n.

3.4 Standard Data Types

As we are interested in the verification of hard- and software, more important for us than natural numbers are standard data types, such as those well-known from their occurrence in any higher-level programming language.

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 here the specification of the data type bool of Boolean values and the data type $\mathsf{list}(\mathsf{nat})$ of the lists over natural numbers. We will need these data types for our further examples as well.

A special case is the data type bool of the Boolean values given by the two constructors $\mathsf{true}, \mathsf{false} : \mathsf{bool}$ without any arguments, for which we globally declare the variable $b : \mathsf{bool}$, but for which we get only the two following axioms by analogy to the axioms for the natural numbers and the axioms for lists:

(bool1) $\quad b = \mathsf{true} \quad \vee \quad b = \mathsf{false}$

(bool2) $\quad \mathsf{true} \neq \mathsf{false}$

Note that the analogy of the axioms of Boolean values to the axioms of the natural numbers (cf. § 3.3) is not perfect: An axiom (bool3) analogous to (nat3) cannot exist because there are no constructors for bool that take arguments. Moreover, (boolS) is superfluous because it is implied by (bool1).

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

(and1) $\quad \mathsf{and}(\mathsf{false}, b) \quad = \mathsf{false}$

(and2) $\quad \mathsf{and}(b, \mathsf{false}) \quad = \mathsf{false}$

(and3) $\quad \mathsf{and}(\mathsf{true}, \mathsf{true}) = \mathsf{true}$

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

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

which takes two arguments, 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. Moreover, let us assume that the variables k, l always range over lists of natural numbers and globally declare $k, l : \mathsf{list}(\mathsf{nat})$.

In analogy to the axioms of the natural numbers, the axioms of this data type are the following.

$$
\begin{array}{ll}
(\mathsf{list(nat)}1) & l = \mathsf{nil} \ \vee\ \exists y,k.\ (\ l = \mathsf{cons}(y,k)\ ) \\
(\mathsf{list(nat)}2) & \mathsf{cons}(x,l) \neq \mathsf{nil} \\
(\mathsf{list(nat)}3_1) & \mathsf{cons}(x,l) = \mathsf{cons}(y,k) \quad \Rightarrow \quad x = y \\
(\mathsf{list(nat)}3_2) & \mathsf{cons}(x,l) = \mathsf{cons}(y,k) \quad \Rightarrow \quad l = k \\
(\mathsf{list(nat)S}) & \forall P.\ \left(\ \forall l.\ P(l)\ \Leftarrow \left(\begin{array}{c} P(\mathsf{nil}) \\ \wedge\ \ \forall x,k.\ (\ P(\mathsf{cons}(x,k))\ \Leftarrow\ P(k)\ ) \end{array}\right)\right)
\end{array}
$$

Moreover, let us define the recursive functions giving the length and the size of a list $\mathsf{length}, \mathsf{size} :\ \mathsf{list(nat)} \rightarrow \mathsf{nat}$, which are helpful when specifying list over natural numbers up to isomorphism:

$$
\begin{array}{lll}
(\mathsf{length(list(nat))}1) & \mathsf{length(nil)} & = \mathsf{0} \\
(\mathsf{length(list(nat))}2) & \mathsf{length}(\mathsf{cons}(x,l)) & = \mathsf{s}(\mathsf{length}(l))
\end{array}
$$

$$
\begin{array}{lll}
(\mathsf{size(list(nat))}1) & \mathsf{size(nil)} & = \mathsf{0} \\
(\mathsf{size(list(nat))}2) & \mathsf{size}(\mathsf{cons}(x,l)) & = \mathsf{s}(x + \mathsf{size}(l))
\end{array}
$$

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 $(\mathsf{list(nat)}3_1)$, which has no analog among the axioms of the natural numbers.
2. None of the axioms $(\mathsf{list(nat)}3_1)$ and $(\mathsf{list(nat)}3_2)$ is implied by the axiom $(\mathsf{list(nat)}1)$ together with the axiom
$$\mathsf{Wellf}(\lambda l,k.\ \exists x.\ (\mathsf{cons}(x,l) = k)),$$
which is the analog to the second axiom of Pieri's specification of the natural numbers.[32]
3. The latter axiom, however, is weaker than each of
$$\mathsf{Wellf}(\lambda l,k.\ (\mathsf{less}(\mathsf{length}(l), \mathsf{length}(k)) = \mathsf{true})),$$
$$\mathsf{Wellf}(\lambda l,k.\ (\mathsf{less}(\mathsf{size}(l), \mathsf{size}(k)) = \mathsf{true})),$$
simply because they state the well-foundedness of relations bigger[33] than $\lambda l,k.\ \exists x.\ (\mathsf{cons}(x,l) = k)$. The well-foundedness of these relations, however, is already implied by the well-foundedness that Pieri used for his specification of the natural numbers.

[32]See § 3.3 for Pieri's specification of the natural numbers. The axioms $(\mathsf{list(nat)}3_1)$ and $(\mathsf{list(nat)}3_2)$ are not implied simply because all axioms beside $(\mathsf{list(nat)}3_1)$ or $(\mathsf{list(nat)}3_2)$ 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 lengths or in their sizes, respectively.

Therefore, the lists of natural numbers can be specified up to isomorphism by a specification of the natural numbers up to isomorphism (see § 3.3), plus the axioms $(\mathsf{list(nat)}3_1)$ and $(\mathsf{list(nat)}3_2)$, plus one of the following sets of axioms:

- $(\mathsf{list(nat)}2)$, $(\mathsf{list(nat)S})$ — in the style of Peano,
- $(\mathsf{list(nat)}1)$, $\mathsf{Wellf}(\lambda l, k.\ \exists x.\ (\mathsf{cons}(x,l)=k))$ — in the style of Pieri,[34]
- $(\mathsf{list(nat)}1)$, $(\mathsf{length(list(nat))}1)$, $(\mathsf{length(list(nat))}2)$ — refining the style of Pieri.[35]

Today it is standard to avoid higher-order axioms in the way exemplified in the last of these three items,[36] and to get along with one second-order axiom for the natural numbers or even with its first-order instances.

Moreover, as some of the most natural functions on lists, let us define the destructors $\mathsf{car} : \mathsf{list(nat)} \to \mathsf{nat}$ and $\mathsf{cdr} : \mathsf{list(nat)} \to \mathsf{list(nat)}$ both in constructor and destructor style. Furthermore, let us define the recursive member predicate

$$\mathsf{mbp} :\ \mathsf{nat},\ \mathsf{list(nat)} \to \mathsf{bool},$$

and the recursive function

$$\mathsf{delfirst} :\ \mathsf{list(nat)} \to \mathsf{list(nat)}$$

that deletes the first occurrence of a natural number in a list:

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

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

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

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

(mbp1) $\mathsf{mbp}(x,\mathsf{nil}) = \mathsf{false}$

(mbp2) $\mathsf{mbp}(x,\mathsf{cons}(y,l)) = \mathsf{true} \ \Leftarrow\ x=y$

(mbp3) $\mathsf{mbp}(x,\mathsf{cons}(y,l)) = \mathsf{mbp}(x,l) \ \Leftarrow\ x\neq y$

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

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

---

[33]Indeed, in case of $\mathsf{cons}(x,l)=k$, we have $\mathsf{less}(\mathsf{length}(l),\mathsf{length}(k)) =$ $= \mathsf{less}(\mathsf{length}(l),\mathsf{length}(\mathsf{cons}(x,l))) = \mathsf{less}(\mathsf{length}(l),\mathsf{s}(\mathsf{length}(l))) = \mathsf{true}$ because of (less4), and we also have $\mathsf{less}(\mathsf{size}(l),\mathsf{size}(k)) = \mathsf{less}(\mathsf{size}(l),\mathsf{size}(\mathsf{cons}(x,l))) = \mathsf{less}(\mathsf{size}(l),\mathsf{s}(x+\mathsf{size}(l))) = \mathsf{true}$ because of (+3) and (less5).

[34]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 $(\mathsf{list(nat)}2)$ in our formalization, and guarantees that Item (6) essentially means $\mathsf{Wellf}(\lambda l, k.\ \exists x.\ (\mathsf{cons}(x,l)=k))$. The last axiom of Item (1) guarantees $(\mathsf{list(nat)}1)$. Moreover, Item (2) guarantees $(\mathsf{list(nat)}3_1)$ and $(\mathsf{list(nat)}3_2)$.

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

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

An immediate consequence of the axiom ($\mathsf{list(nat)}1$) and the definitions ($\mathsf{car}1$) and ($\mathsf{cdr}1$) is the lemma ($\mathsf{cons}1$) and its flattened version ($\mathsf{cons}2'$):

($\mathsf{cons}2$) $\quad \mathsf{cons}(\mathsf{car}(l'), \mathsf{cdr}(l')) = l' \ \Leftarrow\ l' \neq \mathsf{nil}$

($\mathsf{cons}2'$) $\quad \mathsf{cons}(x, l) = l' \ \Leftarrow\ l' \neq \mathsf{nil} \ \wedge\ x = \mathsf{car}(l') \ \wedge\ l = \mathsf{cdr}(l')$

Furthermore, let us define the Boolean function

$$\mathsf{lexless} :\ \mathsf{list(nat)}, \mathsf{list(nat)} \rightarrow \mathsf{bool},$$

which lexicographically compares lists according to the ordering of the natural numbers, and

$$\mathsf{lexlimless} :\ \mathsf{nat}, \mathsf{list(nat)}, \mathsf{list(nat)} \rightarrow \mathsf{bool},$$

its restriction to lengths up to a given natural number:

($\mathsf{lexless}1$) $\quad \mathsf{lexless}(l, \mathsf{nil}) = \mathsf{false}$

($\mathsf{lexless}2$) $\quad \mathsf{lexless}(\mathsf{nil}, \mathsf{cons}(y, k)) = \mathsf{true}$

($\mathsf{lexless}3$) $\quad \mathsf{lexless}(\mathsf{cons}(x, l), \mathsf{cons}(y, k)) = \mathsf{lexless}(l, k) \ \Leftarrow\ x = y$

($\mathsf{lexless}4$) $\quad \mathsf{lexless}(\mathsf{cons}(x, l), \mathsf{cons}(y, k)) = \mathsf{less}(x, y) \ \Leftarrow\ x \neq y$

($\mathsf{lexlimless}1$) $\quad \mathsf{lexlimless}(x, l, k) = \mathsf{and}(\mathsf{lexless}(l, k), \mathsf{less}(\mathsf{length}(l), x))$

Such lexicographic combinations play an important rôle in well-foundedness arguments in 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:[37]

($\mathsf{lexlimless}2$) $\quad \mathsf{Wellf}(\lambda l, k.\ (\mathsf{lexlimless}(x, l, k) = \mathsf{true}))$

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

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

3.5 The Standard High-Level Method of Mathematical Induction

In 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,[38] require considerable ingenuity and have to be carried out. In case of a proof on natural numbers, the experienced mathematician engineers his proof roughly according to the following pattern:

> He starts with the conjecture and simplifies it by case analysis, typically based on the axiom ($\mathsf{nat1}$). When he realizes that the current goal becomes 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 $\mathsf{Wellf}(\lambda x, y : \mathsf{nat}.\ (\mathsf{s}(x) = y))$ 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,[39] 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.[40]

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 still easy to find when the instances for P in (S) are not.

[38] For example, such difficult inductive arguments are the proofs of Hilbert's *first ε-theorem* [Hilbert and Bernays, 1970], Gentzen's *Hauptsatz* [Gentzen, 1935], and confluence theorems such as the ones in [Gramlich and Wirth, 1996], [Wirth, 2009].

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

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

3.6 Descente Infinie

The soundness of the induction method of § 3.5 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.[41] Here is Fermat's Method of *Descente Infinie* in modern language, very roughly speaking:

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

If this method is executed successfully, we have proved $\forall x.\ P(x)$ because no counterexample can be $<$-minimal 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. § 3.2) simply by replacing

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

with its contrapositive

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

Although it was still very hard for Fermat to obtain a positive version of his counterexample method,[42] the negation 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[43] as a synonym for the modern standard high-level method of mathematical induction described in this section.

Let us now prove the lemmas (+3) and (less7) of § 3.3 (in the axiomatic context of § 3.3) by *descente infinie* seen as the standard high-level method of mathematical induction described in § 3.5.

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

[42] Fermat reported in his letter for Huygens that he had had problems to apply 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* is often simpler than 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.

[43] In general, in the tradition of [Wirth, 2004], *descente infinie* is taken nowadays as a synonym for the standard high-level method of mathematical induction. This way of using the term *"descente infinie"* is found in [Brotherston and Simpson, 2007; 2011], [Voicu and Li, 2009], [Wirth, 2005; 2010a; 2012b; 2013]. If the historical perspective before the 19th century is taken, however, 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*).

Applying the Theorem of Noetherian Induction (N) (cf. § 3.2) 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 here on the sub-formula $x+y = y+x$. Based on axiom $(\mathsf{nat1})$ we can reduce this task to the two cases $x=\mathsf{0}$ and $x=\mathsf{s}(x')$ with the two goals

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

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

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

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

$$\begin{array}{ll} \mathsf{0}=\mathsf{0}+\mathsf{0}; & \mathsf{s}(x'+\mathsf{0})=\mathsf{0}+\mathsf{s}(x'); \\ \mathsf{s}(y')=\mathsf{s}(y')+\mathsf{0}; & \mathsf{s}(x'+\mathsf{s}(y'))=\mathsf{s}(y')+\mathsf{s}(x'). \end{array}$$

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

$$\begin{array}{ll} \mathsf{0}=\mathsf{0}; & \mathsf{s}(x'+\mathsf{0})=\mathsf{s}(x'); \\ \mathsf{s}(y')=\mathsf{s}(y'+\mathsf{0}); & \mathsf{s}(x'+\mathsf{s}(y'))=\mathsf{s}(y'+\mathsf{s}(x')); \end{array}$$

respectively. Now we can make immediate progress only if we instantiate the induction hypothesis that is available in the context[44] given by our above formula in four different forms, namely we have to instantiate (x'',y'') with $(x',\mathsf{0})$ $(\mathsf{0},y')$, $(x',\mathsf{s}(y'))$, $(\mathsf{s}(x'),y')$, respectively. Rewriting with these instances, the four goals become:

$$\begin{array}{ll} \mathsf{0}=\mathsf{0}; & \mathsf{s}(\mathsf{0}+x')=\mathsf{s}(x'); \\ \mathsf{s}(y')=\mathsf{s}(\mathsf{0}+y'); & \mathsf{s}(\mathsf{s}(y')+x')=\mathsf{s}(\mathsf{s}(x')+y'); \end{array}$$

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

$$\begin{array}{ll} \mathsf{0}=\mathsf{0}; & \mathsf{s}(x')=\mathsf{s}(x'); \\ \mathsf{s}(y')=\mathsf{s}(y'); & \mathsf{s}(\mathsf{s}(y'+x'))=\mathsf{s}(\mathsf{s}(x'+y')). \end{array}$$

Now the first three goals following 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 $\lambda l,k.\ (\mathsf{lexlimless}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))),l,k)=\mathsf{true})$, which is well-founded according to $(\mathsf{lexlimless2})$ (cf. § 3.4). Then it is trivial to show that $(\mathsf{s}(x'),\mathsf{s}(y'))$ is greater than each of $(x',\mathsf{0})$ $(\mathsf{0},y')$, $(x',\mathsf{s}(y'))$, $(\mathsf{s}(x'),y')$, (x',y').

This completes the proof of our conjecture. □

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

EXAMPLE 3 (Proof of ($\mathsf{less7}$) by *descente infinie*).

The previous proof was very formal w.r.t. its underlying logic, because we wanted to make the application of the Theorem of Noetherian Induction most explicit. 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. Then we have to show

$$\mathsf{less}(\mathsf{s}(x), z) \;\Leftarrow\; \big(\; \mathsf{less}(x, y) \;\wedge\; \mathsf{less}(y, z) \;\big)$$

If we apply the axiom ($\mathsf{nat1}$) twice to both y and z with the intention to reduce the last literal, then, after reduction with ($\mathsf{less1}$), the two base cases have an atom false in their conditions, abbreviating $\mathsf{false} = \mathsf{true}$, which is false according to ($\mathsf{bool2}$), 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 ($\mathsf{less3}$) to

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

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

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

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

The first goal can be generalized to the conjecture $\mathsf{less}(x, z') \Leftarrow \mathsf{less}(\mathsf{s}(x), z')$, but the second goal cannot be generalized to a theorem simpler than our initial conjecture ($\mathsf{less7}$). This means that the application of the given instantiation of the induction hypothesis is useless. Thus, instead of this application, we apply the axiom $\mathsf{nat1}$ also to x, obtaining the cases $x = \mathsf{0}$ and $x = \mathsf{s}(x')$ with the two goals, after reduction with ($\mathsf{less2}$) and ($\mathsf{less3}$),

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

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

respectively. The first is trivial by ($\mathsf{less1}$), ($\mathsf{less2}$) after another application of the axiom $\mathsf{nat1}$ 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 take any of the variables of the original conjecture w.r.t. the irreflexive ordering on the natural numbers or the successor relation.

This completes the proof of the conjecture. Note that we have shown that the proof can only be successful. $\square$

3.7 Explicit Induction

3.7.1 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 § 3.5, a proof calculus should have the 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 (N) (cf. § 3.2), and also with the second-order variable Q in the definition of well-foundedness ($\mathsf{Wellf}(<)$) (cf. § 3.1).

At a time when the dominance of J. Alan Robinson's resolution method suggested a reduction of human-oriented reasoning to the most elementary forms of machine-oriented first-order reasoning, Boyer and Moore were lucky not to implement these human-oriented and higher-order features, because they would probably have failed on the basis of the logic calculi and the computer technology of the 1970s. Instead, they found a way 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.

The induction rule immediately instantiates the higher-order variables P and $<$ in the Theorem of Noetherian Induction with first-order predicates.

This is comparatively straightforward for the predicate variable P, which simply becomes the (simplified) quantifier-free first-order conjecture that is to be proved by induction, and its argument is simply the tuple of the free first-order variables of this conjecture.

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

At the first glance, we would not expect that this induction rule could implement all applications of the Theorem of Noetherian Induction within a successful proof-search heuristic, simply because it has to solve the two hard tasks of an induction proof, namely the Induction-Hypotheses Task and the Induction-Ordering Task (cf. § 3.5), right at the beginning of the proof, before the proof attempt has been sufficiently developed to exhibit its structural difficulties.

As a matter of fact, however, the induction rule is most successful in realizing all applications of the Theorem of Noetherian Induction required within the proof-search heuristic of the Boyer–Moore waterfall (cf. Figure 1). This has to do partly with the poorness of the quantifier-free first-order logic. More important, however, is the possibility of having additional lemmas that can then again be proved by induction.

EXAMPLE 4 (Proof of (+3) by explicit induction).

Let us prove (+3) in the context of § 3.3, just as we have done already in Example 2 (cf. § 3.6), 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 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 suggests to instantiate $<$ to $\lambda x'', y'', x, y.\ (\mathsf{s}(x'')=x)$. As this relation is known to be well-founded, the induction rule reduces this task again based on axiom $(\mathsf{nat1})$ 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, and these two goals must now be shown without the possibility to apply further instances of the induction hypotheses. Because of the very limited look-ahead that the induction rule can have on the whole proof, the induction rule is not able to find the many instances we applied in the proof of Example 2. In general, this would require an arbitrary look-ahead, depending on the size of the structure of the *descente infinie* proof.

Note that the two above goals are exactly the ones that result from a structural induction on x. Nevertheless, the induction rule is in general able to produce much more complicated base and step cases than a simple structural induction on x.

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 — this is the famous *"cross-fertilization"* of the Boyer–Moore waterfall (cf. Figure 1) — and then throw away the condition completely, with the intention to obtain a stronger induction hypothesis. The simplified second goal reduces to

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

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

All in all, although the induction rule does not find the more complicated induction hypotheses of the *descente infinie* proof of Example 2, it well suffices to prove our original conjecture with the help of the additional lemmas $y = y+\mathsf{0}$ and $\mathsf{s}(y+x') = y+\mathsf{s}(x')$. From a logical viewpoint, these lemmas are redundant because they follow from the original conjecture and the definition of $+$. From a heuristic viewpoint and oriented for rewriting from right to left, however, they are much more useful than the original conjecture, because their application tends to terminate in the context of the overall simplification by symbolic evaluation, which constitutes the first stage in the Boyer–Moore waterfall (cf. Figure 1). □

The two proofs of the very simple conjecture (+3) given in Examples 2 and 4, respectively, nicely demonstrate how the induction rule manages to prove simple theorems very efficiently and with additional benefits for the further performance of the simplification procedure.

Moreover, if the overall waterfall heuristic fails, the user can help any Boyer–Moore theorem prover except the PURE LISP THEOREM PROVER by stating additional lemmas with additional notions, that will finally help the induction rule to prove also very hard theorems.

These two proofs of (+3), however, can only give a very rough idea on the advantage of *descente infinie* for hard induction proofs, but see Example 9 in § 5.2.6. For a more difficult higher-order proof by descente infinie see § 3.4 of [Wirth, 2004] for a complete formal proof of M. H. A. Newman's famous lemma, where the reverse of a well-founded relation is shown to be confluent in case of local confluence by *induction w.r.t. its own ordering*, a situation where explicit induction cannot even be applied.[45]

3.7.2 Theoretical Viewpoint

Under the theoretical aspect, we should be aware that the models of explicit induction, say for the natural numbers, include also non-standard models,[46] where there are **Z**-chains in addition to the natural numbers **N**, contrary to the higher-order specifications of Peano and Pieri up to isomorphism with **N**. These **Z**-chains cannot be excluded simply because all applications of the Theorem of Noetherian Induction are confined to the induction rule, which does not use any higher-order properties, but only first-order instances of the Theorem of Noetherian Induction, and the remaining inference rule realize only first-order deductive reasoning.

[45]Note that, though confluence is the Church–Rosser property, the Newman Lemma has nothing to do with the Church–Rosser Theorem for untyped λ-calculus. This theorem was actually executed with a Boyer–Moore theorem prover in the first half of the 1980s by Shankar [1988], following the Tait/Martin-Löf proof found e.g. in [Barendregt, 2012]. Unlike the Newman Lemma, Shankar's proof proceeds by structural induction on the λ-terms and not by Noetherian induction w.r.t. the reverse of the reduction relation; indeed, untyped λ-calculus in nonterminating.

[46] See also Note 104.

3.7.3 Practical Viewpoint

Under the practical aspect, we have to be aware that application of the induction rule of explicit induction does not refer to the Theorem of Noetherian Induction anymore, but is actually confronted with 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 to the input conjecture reduces, i.e. whose validity implies the validity of the input conjecture.

Therefore, the induction rule of explicit induction has to solve three tasks:

1. Choose some of the variables in the conjecture as *induction variables.*

2. Split the conjecture into several base and step cases, based on the induction variables' demand on which governing conditions (and constructors)[47] have to be added to unfold the recursive function calls on the induction variables without further case analysis. This means that we successively apply axioms of the form of (nat1) (cf. § 3.3) and (list(nat)1) (cf. § 3.4) to the induction variables.

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

3.8 Generalization

Contrary to merely deductive, analytic theorem proving, an input conjecture for an inductive proof 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. The 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.

Generalization is hardly needed when humans supply the input conjecture. As we have seen in § 3.8, explicit induction often has to start another induction during the proof. In this case, however, the secondary, machine-generated input conjecture often requires generalization for its proof attempt to be successful.

Generalization typically proceeds by the replacement of all occurrences of a non-variable term with a fresh variable. This is especially promising for a subsequent induction if the same non-variable term t has multiple occurrences in the conjecture, and becomes even more promising if these occurrences are found on

[47]This applies when constructor style is either found in the recursive function definitions or or to be used for the step cases. In the Pure LISP Theorem Prover, only the latter is the case. In Thm, none of this is the case.

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*, i.e. the reduction to an invalid new conjecture, sub-terms are to be preferred to super-terms, and one should never generalize if t is 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 or its direct arguments, or the first argument of a conditional (`IF`). In any of these cases, the information loss by generalization would typically be so high that it probably results in an invalid conjecture.

EXAMPLE 5 (Proof of (ack4) by Explicit Induction and Generalization).

Let us prove (ack4) in the context of § 3.3 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. Thus, according to the induction rule of explicit induction we should try to prove the the goals

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

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

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

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

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

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

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

Now the first case is simply an instance of our lemma (less4). Let us simplify the two remain step cases a bit by introducing variables for their common subterms:

$$\mathsf{less}(0, z) = \mathsf{true} \;\Leftarrow\; \left(\begin{array}{ll} & \mathsf{less}(\mathsf{s}(0), z) = \mathsf{true} \\ \wedge & z = \mathsf{ack}(x', \mathsf{s}(0)) \end{array}\right)$$

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

Now the first follows from from applying (nat1) to z. Before we can prove the second by another induction, however, we have to generalize it to the lemma (less7) of § 3.3 by deleting the last to 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 ($\mathsf{less7}$), 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 NQTHM is able to do this whole proof and generalize the second step case to the lemma ($\mathsf{less7}$) completely automatically, even when it works with a redefined empty arithmetic theory. Moreover, as shown in § 3.3 of [Wirth, 2004], in a slightly richer logic, the heuristics of NQTHM would additionally admit to synthesize the lower bound in the first argument of less from an input conjecture like $\exists z.\ (\mathsf{less}(z, \mathsf{ack}(x, y)) = \mathsf{true})$, simply because less does not contribute to the choice of the base and step cases.

3.9 Proof-Theoretical Peculiarities of Mathematical Induction

The following two proof-theoretical peculiarities of induction compared to first-order deduction may be considered noteworthy:[48]

- A calculus for arithmetic cannot be complete, simply because the theory of the arithmetic of natural numbers is not enumerable.[49] [50]

- According to Gentzen's Hauptsatz,[51] 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.[52] 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.[53]

[48]Note, however, that these peculiarities of induction do not make a difference to first-order deductive theorem proving *in practice*. See Notes 53 and 50.

[49]Cf. [Gödel, 1931].

[50]In practice, however, we have to extend our proof search to additional lemmas and notions anyway, and it does not really matter 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]).

[51]Cf. [Gentzen, 1935].

[52]Cf. [Kreisel, 1965].

[53]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.

3.10 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 from the top-level. 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, which again can be seen as a global application of the Theorem of Noetherian Induction.

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

To express the relevant notions in these data types, we need *recursion*, a peculiar method of definition, which have often used in this section, based on its informal intuitiveness, but we did not actually discuss its formal admissibility requirements. We make up for this omission in § 4, with a focus on modes of recursion that admit an effective consistency test.

Moreover, a mathematical proof method cannot be completely captured by its non-procedural logic formalization; and so we still have to develop effective heuristics for actually finding proofs by induction. This issues is treated in § 5, with a focus on heuristics that can be automated completely.

This Conclusion needs revision.

4 RECURSION

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 §§ 3.3 and 3.4, such as the ones of +, less, length, and size, where these function symbols occurred in some of the right-hand sides of the equations of their own definitions; for instance, the function symbol + occurs in the right-hand side of (+2) in § 3.3.

Beginning with the PURE LISP THEOREM PROVER in 1972, recursive functions were to be defined in all Boyer–Moore theorem provers in the style of the function definitions of the programming language LISP.[54]

EXAMPLE 6. To avoid the association of routine knowledge, let us not consider a function definition over lists (as standard in LISP), but over the natural numbers. For example, instead of our two equations (+1), (+2) for +, we find the following single equation on Page 53 of the standard reference in [Boyer and Moore, 1979]:

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

The primary difference here is that PLUS is defined in *destructor style* instead of the *constructor style* of our equations (+1), (+2) in § 3.3. As there is no essential difference between these two styles, let us transform our definition of + from (+1), (+2) into destructor style.

In place of the untyped destructor SUB1, let us use the typed destructor p defined by either by (p1) or by (p1′) of § 3.3, 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:

$$(+1') \qquad x+y = y \qquad \Leftarrow \quad x = 0$$

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

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 here; 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.[55] □

[54] Cf. [McCarthy *et al.*, 1965].

There are three kinds of restriction explicitly required for any function definition in the Boyer–Moore theorem provers.

The first restriction is on *confluence* of the rewrite relation that results from reading the defining equations as rewrite rules that admit to replace occurrences of left-hand sides of instantiated equations with their respective right-hand sides, provided that their conditions are fulfilled.[56] This restriction guarantees that no distinct objects of the data types can be equated by the recursive function definitions. If we assume the axioms $(\mathsf{nat2})$ and $(\mathsf{nat3})$ (see § 3.3), then this restriction is essential for consistency.

The two other restrictions, *reducibility* of all ground terms w.r.t. the rewrite relation and *termination* of the rewrite relation, are not essential; when they are both satisfied, as required in all Boyer–Moore theorem provers, then — similar to the classical case of explicitly defined notions — we can get rid of the recursively defined function symbols, but only if all arguments of these function symbols are concrete objects of data types (i.e. terms without variables, consisting solely of constructor function symbols).

Let us consider these three kinds of restriction in more detail.

4.1 Confluence

If we consider the $(+2)$ together with $(+2')$, then we can rewrite $\mathsf{s}(x)+y$ on the one hand with $(+2)$ into $\mathsf{s}(x+y)$, and, on the other hand, with $(+2')$, into $\mathsf{s}(\mathsf{p}(\mathsf{s}(x))+y)$, provided that $x\neq\mathsf{0}$ is fulfilled. This case does not seem to be problematic, because the latter result can be rewritten to the former one by $(\mathsf{p1})$. In general, however, confluence is undecidable and criteria sufficient for confluence are extremely hard to develop.

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 add $\mathsf{p}(x)=\mathsf{0}$ to $(\mathsf{p1})$, then we get $\mathsf{s}(\mathsf{0})=\mathsf{p}(\mathsf{s}(\mathsf{s}(\mathsf{0})))=\mathsf{0}$, in contradiction to the axiom $(\mathsf{nat2})$ of § 3.3.

[55] It is indeed easy to see this, at least with the additional information that `IF X Y Z` is nothing but "`IF X` 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. Note that contrary to modern LISP dialects, such as COMMON LISP [Steele Jr., 1990], in the LISP 1.5 of [McCarthy *et al.*, 1965] there was no way to restrict the variables of the definition of `PLUS` to the type of natural numbers.

[56] For the technical meaning of *fulfilledness* in the recursive definition of the rewrite relation see [Wirth, 2009], where it is also explained why the positive/negative-conditional equation equations simple respect the straightforward purely logical semantics in the specification.

For the recursive definitions admissible in the Boyer–Moore theorem provers, confluence results from the absence of divergence, which is an immediate consequence of the LISP definition style; namely that there is only one defining equation for each new function symbol,[57] and that all variables occurring on the right-hand side of the definition also occur on the left-hand side of the defining equation.[58] [59]

4.2 Reducibility

It is not only so that we can check the soundness of $(+1')$ and $(+2')$ independently from each other, we can also omit one of the equations, resulting in a partial definition of the function $+$. Indeed, for the function p we did not specify any value for $\mathsf{p}(0)$; so $\mathsf{p}(0)$ is not reducible in the term-rewrite system 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 any concrete elements of the data types (in our case the natural numbers) must be reducible. This reducibility immediately results from the LISP definition style, which requires all arguments of the function symbol on left-hand side of its definition to be disjoint variables.[60]

4.3 Termination

In addition to the LISP definition style, the Boyer–Moore theorem provers require termination of the rewriting with function definitions as reduction rules in the sense that — *before* the definition of a new function is admitted to the specification — a proof of termination of the following form is required:[61]

[57] 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].

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

[59] The most general applicable confluence criterion for positive/negative-conditional equations is found in [Wirth, 2008] and avoids divergence in a similar way by a criterion that can be easily checked, although the proof that it guarantees confluence is very involved.

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

[61] Cf. Item (d) of the "definition principle" of [Boyer and Moore, 1979, p. 44f.]. Note that in the Pure LISP Theorem Prover, however, this termination was not proved; instead, the soundness of the induction proofs came with the *proviso* that all the rewrite relation of all defined function symbols terminate.

> For every occurrence of the new function symbol f on the right-hand side of the defining equation, show that its arguments — when given as arguments to a newly defined *weight function*[62] w_f (taking exactly the arguments of the new function symbol) — are smaller in a well-founded relation than the variables occurring as arguments of f on the left-hand side given as arguments to w_f; and all this, of course, is required only if the conditions governing this occurrence are all satisfied.

EXAMPLE 7. For our definition of $+$ with $(+1')$, $(+2')$ in Example 6, the only occurrence of $+$ on the right-hand side is in $(+2')$, and it has the arguments $\mathsf{p}(x)$ and y, governed by the condition $x \neq 0$. If we define the weight function $w_+ : \mathsf{nat}, \mathsf{nat} \rightarrow \mathsf{nat}$ by $w_+(x, y) = x$, and choose the well-founded ordering $\lambda x, y.\ (\mathsf{less}(x, y) = \mathsf{true})$, then we have to prove

$$\mathsf{less}(w_+(\mathsf{p}(x), y), w_+(x, y)) = \mathsf{true} \;\Leftarrow\; x \neq 0,$$

i.e.

$$\mathsf{less}(\mathsf{p}(x), x) = \mathsf{true} \;\Leftarrow\; x \neq 0,$$

which should be no problem for our theorem prover. □

4.4 Constructor Variables

In this section we have presented the general admissibility conditions for the definition of recursive functions, namely confluence (which is essential for consistency) and the two additional ones of the Boyer–Moore theorem provers, reducibility and termination.

Note that the two additional restrictions imply that only *total recursive* functions are admissible in the Boyer–Moore theorem provers.[63] Contrary to that, the inductive theorem prover QuodLibet admits also *partial recursive* functions.

It is an interesting question why Boyer and Moore brought up these two additional restrictions. A possible 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 as follows: First, a function $\mathsf{russell} : \mathsf{bool} \rightarrow \mathsf{bool}$ is defined, which is essentially the following:

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

(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. This, however, is the case only if the variable b of the axiom (bool1) (cf. § 3.4) can be instantiated with the term $\mathsf{russell}(b)$, which we do not have

[62]Or "measure function" as it is called in the tradition of Boyer and Moore.

[63]There is an explicit reference to [Schoenfield, 1967] as the basis for the logic of the Pure LISP Theorem Prover on Page 93 of [Moore, 1973].

to permit: If all variables we have introduced so far are *constructor variables*[64] in the sense that they can only be instantiated with terms formed from constructor function symbols (incl. constructor constants) and constructor variables, then the values of the function russell can exist as a *junk objects* different from true and false, and no inconsistency arises.[65]

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

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 reads ∃`Z(COND X (COND 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.

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

[65]For the resulting semantics see in particular [Wirth and Gramlich, 1994b], and [Kühler and Wirth, 1997].

5 AUTOMATED EXPLICIT INDUCTION

5.1 The Application Context of Automated Explicit Induction

Since the upcoming of programmable computing machinery in the middle of the 20th century until today, a major problem of hardware and software is the uncertainty whether it actually does what it should do.

To simplify matters, let us assume that none of the components of the physical hardware is broken and that the machinery situated at a place where we can neglect the influence of the environment. Under this assumption, the task to show that the hard- and software actually implements the intended functionality can be made concrete, at least in principle, in the following way:

We specify the intended functionality in a language of formal logic, and then we supply a formal proof that the program actually satisfies the specification.

This approach requires a formal specification of the hardware and the involved programming languages. Moreover, to be complete, this approach would also require a verification that the implementation of the programming languages (via interpreters or compilers) actually follows this specification. The crucial problem, however, are the costs of the many proofs of the huge amounts of application software in a market-oriented economy.[66]

To reduce the complexity of the approach, we can restrict ourselves to a single programming languages which is very close to formal logics and admits a simple implementation. Moreover, to reduce the costs of the software verification, we can hope to automate it with automated theorem-proving systems. This means a mechanization of mathematical induction because most data types applied in programming, such as natural numbers, pairs, arrays, lists, and trees, require induction for the verification of their properties.

This section on Application Context needs revision.

[66]Although the costs for verification of a central processing unit are economically maintainable, a company in a market-oriented economy may have problems with these relatively small extra costs, because their competitors may be able to offer slightly lower prices and push that company out of the market. Even the economical ruin of some of these competitors because of the liabilities for their faulty processing units does not guarantee a future success of verification in the business.

5.2 The PURE LISP THEOREM PROVER

From the historical perspective the big question is how Robert S. Boyer and J Strother Moore could invent their outstanding solutions to the hard heuristic problems in the automation of induction, implemented in the sophisticated theorem prover THM (cf. [Boyer and Moore, 1979]), starting virtually from zero[67] in the middle of 1972.

As already described in § 1, the main breakthrough in the heuristics for automated theorem proving was achieved with the "PURE LISP THEOREM PROVER", developed and implemented by Boyer and Moore. It was presented by Moore at the third IJCAI, which took place in Stanford (CA) in August 1973,[68] but it is best documented in Part II of Moore's PhD thesis [1973], defended in November 1973.

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

To understand the achievements a bit better, let us now discuss the material of Part II of Moore's PhD thesis in more detail, because it provides some explanation on how Boyer and Moore could be so surprisingly successful. Especially helpful for understanding the process of creation are those procedures of the PURE LISP THEOREM PROVER that are provisional w.r.t. to their refinement in later Boyer–Moore theorem provers. Indeed, these provisional procedures help to decompose the giant leap from nothing to THM, which has no parallel in the history of automated theorem proving, to steps of a more comprehensible size.

As W. W. Bledsoe (1921–1995) was Boyer's PhD advisor, it is likely that Boyer was building the PURE LISP THEOREM PROVER on the advanced standpoint Bledsoe had developed in the automation of theorem proving.[69]

---

[67] No heuristics were explicitly described in Rod M. Burstall's considerations of the year 1968 on program verification by induction over recursive functions in [Burstall, 1969]; 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]

Still in 1972, all known implementations of automated inductive theorem provers based on recursive functions (we know only of the ones of W. W. Bledsoe and Robert S. Boyer) were just able to apply the Axiom of Structural Induction (S) of § 3.3 to a randomly picked variable of type nat, which is not worth mentioning in comparison with the Boyer–Moore theorem provers.

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

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

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

On the method of Boyer and Moore for the development of their induction heuristics in late 1972 and early 1973, in particular of their famous waterfall (cf. Figure 1), they report that they were doing proofs on list data structures on the blackboard and verbalizing to each other the heuristics behind their choices on how to proceed with the proof.[70] This means that the heuristics of the Pure LISP Theorem Prover are learned from human heuristics, although explicit induction itself is not the approach humans would choose for non-trivial induction tasks.

Note that their method of learning computer procedures from their own human behavior in mathematical logic also meant a step of two young men against the spirit of the time when the dominance of J. Alan Robinson's resolution method suggested the application of vast amounts of computational power to most elementary forms of "machine-oriented" (i.e. not human like) first-order reasoning. It may well 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,[71] but the major influence of Boyer and Moore is here again W. W. Bledsoe.[72]

The source code of the Pure LISP Theorem Prover was written in the programming language POP–2.[73] 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 10 s.[74]

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 the scratch and does not improve its behavior with the help of previously established lemmas.

Moreover, the induction orderings in the Pure LISP Theorem Prover are exclusively recombinations of constructor relations, such that all inductions it can perform 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.[75]

[70]Interview of Boyer and Moore by Wirth on Thursday, Oct. 7, 2012.

[71]Cf. e.g. the quotation from [Burstall, 1969] in Note 67.

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

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

[74]This timing result is hard to believe and strongly indicates that Boyer and Moore were as great in coding as they were in creating heuristics for theorem proving. 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 microseconds each."

Nevertheless, the soundness of the PURE LISP THEOREM PROVER depends directly on the termination of the recursive function definitions, but only in one single aspect: It simplifies and evaluates expressions under the assumption of termination. For instance, both `(IF`[76] 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.[77]

The termination of the recursively defined functions, however, is not at all 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 (or pure) subset of the logic of LISP.[78] The only *destructors* in this logic are `CAR` and `CDR`, which take apart the only *constructors* `NIL` and `CONS` according to the equations `(CAR (CONS` a d`)) =` a and `(CDR (CONS` a d`)) =` d. Moreover, to enforce totality, the destructors are overspecified via `(CAR NIL) = NIL` and `(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 in place of an argument 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`.[79] Moreover, the natural number `0` is represented by `NIL` and the successor function $\mathsf{s}(d)$ is represented by `(CONS NIL` d`)`.[80]

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.

[75]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 confluence is sufficient here as discussed in § 4 along the results of [Wirth, 2009].

[76]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 write "`IF`" for every "`COND`" of the PURE LISP THEOREM PROVER.

[77]Contrary to later Boyer–Moore theorem provers, the PURE LISP THEOREM PROVER does not require termination for (the non-existing) lemma application.

[78]Beside the imperative commands of LISP, such as variants of `PROG`, `SET`, `GO`, and `RETURN`, this *pure* subset of LISP also lacks all functions or special forms that depend on the memory representation of the structures, such as `EQ`, `RPLACA`, and `RPLACD`, which can be used in LISP to realize circular structures or to save memory space.

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

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

5.2.1 Simplification in the Pure LISP Theorem Prover

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

"Normalization" tries find sufficient conditions for a given expression to have the soft type "Boolean" and to normalize logical expressions. Contrary to clausal logic over equational atoms, LISP admits `EQUAL` and `IF` to appear not only on top level, but in arbitrary mutually 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*, based on the obvious fact that in

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

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: If we know that c is of soft type "Boolean", then we rewrite all occurrences of c in p actually to `T`. Moreover, 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. 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 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. The Pure LISP Theorem Prover, however, does not do this at all. Instead, it applies the normalization only if one of l, r is a ground term.[81] If this is the case then the other is not 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 constructor 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". Note, however, that further contextual rewriting with equalities is applied in a later stage of the Boyer–Moore waterfall, named *cross-fertilization.*

"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.

[81] Actually this ground term is always a *constructor* ground term because the previously applied "evaluation" procedure has reduced any ground term to a constructor ground term, provided that the termination *proviso* is satisfied.

By "evaluation", ground terms are completely evaluated to their normal forms. Terms containing (implicitly universally quantified) variables, however, have to be handled in addition. Surprisingly, the considered rewrite relation is not necessarily terminating on non-ground terms, although the LISP evaluation of ground terms terminates because of the assumed termination condition for recursive function definitions (cf. § 4.3). The reason for this non-termination is the following: Because of the LISP definition style via *un*conditional equations, the positive/negative conditions are 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, 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.[82] Later Boyer–Moore theorem provers also use lemmas for rewriting during symbolic evaluation, which is another source of possible non-termination. As the final aim is a form that provides a sufficiently concise and strong induction hypothesis, symbolic evaluation must be prevented from unfolding function definitions unless the context lets us expect an effect of simplification. Today this is typically done by *contextual rewriting* where evaluation stops when the governing conditions cannot be established from the context.[83]

Because the main function of "evaluation" in the PURE LISP THEOREM PROVER, however, 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 completely different 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 matching of 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.[84]

[82]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 case 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.

[83]See, for instance, the evaluation in QUODLIBET as described in [Schmidt-Samoa, 2006b; 2006c].

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.

"Evaluation" provides an interesting link between symbolic evaluation and the induction rule, which helps to understand how it was possible to develop the more sophisticated recursion analysis of later Boyer–Moore theorem provers stepwise. So 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 quite different question "Where to destructors like `CAR` and `CDR` block the further symbolic evaluation?".

5.2.2 Destructor Elimination in the Pure LISP Theorem Prover

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

5.2.3 (Cross-) Fertilization in the Pure LISP Theorem Prover

Fertilization is just contextual rewriting with an equality, described before for the "reduction" that is part of the simplification of the Pure LISP Theorem Prover, but now with an equation between *two non-ground* terms.

The most important case of fertilization is called *"cross-fertilization"*. It occurs very often in step cases of induction proofs of equational theorems, and we have seen it already in Example 4 of § 3.7.1.

Neither Boyer nor Moore ever explained why cross-fertilization is cross. 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 helps to remember that only the side (i.e. left- or right-hand) of the induction conclusion that 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. 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 memorized by the fact that — in the image — only one (diploid) genetic code remains.

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

[84]Actually, "evaluation" also informs which occurrences of `CAR` or `CDR` beside the arguments of recursive occurrences of `PLUS` were permanently introduced during that trial to rewrite. In general, such occurrences indicate which additional case analysis is to be generated by the induction rule.

5.2.4 *Generalization in the* Pure LISP Theorem Prover

Generalization in the Pure LISP Theorem Prover works as described in § 3.8. 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.[85]

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.

5.2.5 *Elimination of Irrelevance in the* Pure LISP Theorem Prover

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

5.2.6 *Induction in the* Pure LISP Theorem Prover

The reader is reminded of the short § 3.7.3 and the three tasks or steps it displays, which the induction stage of the Boyer–Moore waterfall has to solve when it applies the induction rule to its input formula.

EXAMPLE 8 (Induction Rule in the Explicit Induction Proof of (ack4)).
Let us see how these three steps look like for the induction formula of Example 5 of § 3.8. According to the definition of ack in (ack1), (ack2), (ack3) (cf. § 3.3), we should take x and y as induction variables (Step 1), and generate our base and step cases according to the substitutions

$$\begin{array}{l}\{x \mapsto \mathsf{0}\},\\ \{x \mapsto \mathsf{s}(x'),\ \ y \mapsto \mathsf{0}\},\\ \{x \mapsto \mathsf{s}(x'),\ \ y \mapsto \mathsf{s}(y')\}.\end{array}$$

The definition of less does not suggest any induction because its second argument is blocked in the input conjecture. The Pure LISP Theorem Prover gets its information on these steps from its already described "evaluation". It requires the axioms (ack1), (ack2), (ack3) to be in destructor instead of constructor style:[86]

[85]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.

[86]We ignore the here irrelevant fact that the Pure LISP Theorem Prover actually has a list representation of the natural numbers.

(ack1′)	$\mathsf{ack}(x,y) = \mathsf{s}(y)$	$\Leftarrow\ x = 0$
(ack2′)	$\mathsf{ack}(x,y) = \mathsf{ack}(\mathsf{p}(x),\mathsf{s}(0))$	$\Leftarrow\ x \neq 0 \ \wedge\ y = 0$
(ack3′)	$\mathsf{ack}(x,y) = \mathsf{ack}(\mathsf{p}(x),\mathsf{ack}(x,\mathsf{p}(y)))$	$\Leftarrow\ x \neq 0 \ \wedge\ y \neq 0$

"Evaluation" would not rewrite the input conjecture with this definition, but write a fault description for the permanent occurrences of p as arguments of the three occurrences of 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 less essentially consists of the pocket $(\mathsf{p}(y),\mathsf{p}(\mathsf{ack}(x,y)))$. This fault description does not suggest any induction because one of the arguments of 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 p in all pockets as induction variables. As this is the only suggestion, no subsumption or merging of suggested inductions is required here. So the PURE LISP THEOREM PROVER picks the right set of induction variables in Step 1.

It fails, however, with Step 2, where it generates the base and step cases according to the substitutions[87]

$\{x \mapsto 0\}$,
$\{y \mapsto 0\}$,
$\{x \mapsto \mathsf{s}(x'),\ \ y \mapsto \mathsf{s}(y')\}$.

This turns the first step case the proof of Example 5 into a base case, for which the proof can only fail. The PURE LISP THEOREM PROVER also fails with the step case it actually generates:

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

This step case has only one hypothesis, which is none of the two we need. □

Thus, regarding the three tasks or steps described in § 3.7.3, the PURE LISP THEOREM PROVER does well in Step 1, but fails[88] with Steps 2 and 3 in Example 8. As seen from that example, the PURE LISP THEOREM PROVER shows a bad performance on the rarely occurring *non-trivial inductions on several variables.* Moreover, as also seen from that example, the main weakness of the PURE LISP THEOREM PROVER is that all information it considers on the right-hand sides of the recursive function definitions comes from the fault descriptions of previous applications of "evaluation", which ignore too much information required for the proper instances of the induction hypotheses. As a consequence, all these hypotheses instantiate the input conjecture exclusively with constructor terms (though multiple hypotheses for a step case can be generated).

[87] 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].

[88] There are indications that Steps 2 and 3 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, an is not handled in its full generality by the program."

Compared to the sophisticated *recursion analysis* of the later Boyer–Moore theorem provers, the induction stage of the PURE LISP THEOREM PROVER is pretty straightforward and undeveloped, although it somehow contains all essential later ideas in a rudimentary form, such as recursion analysis, the three above steps, and the merging of step cases. To see the latter we have to consider another example.

EXAMPLE 9 (Proof of (less7) by Explicit Induction, incl. Merging of Step Cases).

Let us write $T(x, y, z)$ for (less7) of § 3.3. From the proof of (less7) in Example 3 of § 3.6 we can learn the following: The proof becomes simpler when we take

$$T(0, \mathsf{s}(y'), \mathsf{s}(z'))$$

as base case (beside say $T(x, y, 0)$ and $T(x, 0, \mathsf{s}(z'))$), instead of any of

$$T(0, y, \mathsf{s}(z')), \qquad T(0, \mathsf{s}(y'), z), \qquad T(0, y, z).$$

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 induction rule of explicit induction looks ahead only one rewrite step, separately for each occurrence of a recursive function in the conjecture.

This means that there is no way for explicit induction 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 gets its information on these steps from its already described "evaluation". It requires the axioms (less1), (less2), (less3) to be in destructor style:[89]

(less1′)	$\mathsf{less}(x, y) = \mathsf{false}$	$\Leftarrow\ y = 0$
(less2′)	$\mathsf{less}(x, y) = \mathsf{true}$	$\Leftarrow\ y \neq 0 \ \wedge\ x = 0$
(less3′)	$\mathsf{less}(x, y) = \mathsf{less}(\mathsf{p}(x), \mathsf{p}(y))$	$\Leftarrow\ y \neq 0 \ \wedge\ x \neq 0$

"Evaluation" does not rewrite any of the occurrences of less 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 less 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 $T(0, y, z)$, which requires some considerable extra work, but does not result in a failure. □

[89]We ignore the here irrelevant fact that the PURE LISP THEOREM PROVER actually has a list representation of the natural numbers.

5.3 Thm

Boyer and Moore never gave names to their theorem provers.[90] The names "Thm" (for "theorem prover"), "Qthm" (for "quantified Thm"), and "Nqthm" (for "new quantified Thm") were actually the directory names under which the different versions of their theorem provers were developed and maintained.[91] 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. Thm appeared yet in publication only as a mode in Nqthm,[92] 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.[93] The name "Nqthm" appeared in publication first as a name of a reference in [Boyer and Moore, 1987],[94] and then in [Boyer and Moore, 1988b] as a mode in Nqthm.

In this section we describe the enormous heuristic improvements documented in [Boyer and Moore, 1979] as compared to [Moore, 1973] (cf. § 5.2). In case of the minor differences of the logic described in [Boyer and Moore, 1979] and of the later released version that is simulated by the `THM` mode in Nqthm as documented in [Boyer and Moore, 1988b; 1998], we try to follow the latter description, partly because of its elegance, partly because Nqthm is an easily 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]".

Note the clearness, precision, and detail of the natural-language descriptions of heuristics in [Boyer and Moore, 1979] is unique and unrivaled.[95] To the best of our knowledge, there is similarly broad treatment of heuristics in theorem proving.

[90]The only exception seems to be [Moore, 1975], 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, 1975] only in [Boyer and Moore, 1975], Boyer has had an equal share in contributing to the Pure LISP Theorem Prover right from the start.

[91]Cf. [Boyer, 2012].

[92]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].

[93]Cf. [Boyer, 2012].

[94]The actual occurrence at the end of § 3.3 of the technical report [Boyer and Moore, 1987] is the name of a bibliographic reference that was not resolved properly. In the journal version [Boyer and Moore, 1988a] and in the proceedings version [Boyer and Moore, 1989], this link is resolved and just reads "9".

[95]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."

From 1973 to 1981 Boyer and Moore were researchers at Xerox Palo Alto Research Center (Moore only) and at SRI International in Menlo Park (CA) just a few miles apart. Since 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 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.[96]

The approach for developing the heuristics has not changed. Just as in the PURE LISP THEOREM PROVER, the heuristics of THM are still learned from the heuristics of the human mathematicians Boyer and Moore.

The logic of THM has changed a bit compared to the PURE LISP THEOREM PROVER. By means of the new *shell principle*,[97] 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.[98] In addition, for each argument of the shell, a destructor[99] has to be described, together with a default value (for the sake of totality). As the logic remains untyped, the user also has to supply a name for the predicate that tests for being an object of the new data type.

`NIL` has lost its elementary status and is now an element of the shell `PACK` of symbols.[100] `T` and `F` now abbreviate the nullary function calls `(TRUE)` and `(FALSE)`, respectively, which are the only Boolean values. Any argument with

[96]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."

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

[98]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 record types. 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:

$$\begin{array}{rlrl} \mathsf{empty\text{-}or\text{-}tree} : & \mathsf{or\text{-}tree}; & \mathsf{or} : & \mathsf{and\text{-}tree},\ \mathsf{or\text{-}tree} \rightarrow \mathsf{or\text{-}tree}; \\ \mathsf{empty\text{-}and\text{-}tree} : & \mathsf{and\text{-}tree}; & \mathsf{and} : & \mathsf{or\text{-}tree},\ \mathsf{and\text{-}tree} \rightarrow \mathsf{and\text{-}tree}. \end{array}$$

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

[100]The 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 `(EQUAL (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].

Boolean intention beside `F` is taken to be `T` (including `NIL`).

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

1. The shell `(ADD1 X1)` of the *natural numbers*, with
   - type restriction `(NUMBERP X1)`,
   - base object `(ZERO)`, abbreviated by `0`,
   - destructor `SUB1` with default value `0`, and
   - type predicate[99] `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 the following logical axioms to the theory, respectively:

#	Axioms Generated by Shell `ADD1`	Axioms Generated by Shell `CONS`
0.1	`(NUMBERP X)` = `T` $\lor$ `(NUMBERP X)` = `F`	`(LISTP X)` = `T` $\lor$ `(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` $\lor$ `(NUMBERP X)` = `F`
1	`(ADD1 (SUB1 X))` = `X` $\Leftarrow$ `X` $\neq$ `0` $\land$ `(NUMBERP X)` = `T`	`(CONS (CAR X) (CDR X))` = `X` $\Leftarrow$ `(LISTP X)` = `T`
2	`(ADD1 X1)` $\neq$ `0`	

3	`(SUB1 (ADD1 X1))` = `X1` $\Leftarrow$ `(NUMBERP X1)` = `T`	`(CAR (CONS X1 X2))` = `X1`
		`(CDR (CONS X1 X2))` = `X2`

4	`(SUB1 0)` = `0`	

5.1	`(SUB1 X)` = `0` $\Leftarrow$ `(NUMBERP X)` = `F`	`(CAR X)` = `0` $\Leftarrow$ `(LISTP X)` = `F`
		`(CDR X)` = `0` $\Leftarrow$ `(LISTP X)` = `F`

5.2	`(SUB1 (ADD1 X1))` = `0` $\Leftarrow$ `(NUMBERP X1)` = `F`	
L1[101]	`(ADD1 X)` = `(ADD1 0)` $\Leftarrow$ `(NUMBERP X)` = `F`	
L2[102]	`(NUMBERP (SUB1 X))` = `T`	

[101]Proof of L1 from 0.2, 1–2, 5.2: We show `(ADD1 X)` = `(ADD1 (SUB1 (ADD1 X)))` = `(ADD1 0)` under the assumption of `(NUMBERP X)` = `F`. The first step is a backward application of the conditional equation 1 via {`X` $\mapsto$ `(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.

[102]Proof of Lemma L2 from 0.1–0.3, 1–4, 5.1–5.2 by *reductio ad absurdum*:
For a counterexample `X`, we get `(SUB1 X)` $\neq$ `0` by 0.3, as well as `(NUMBERP (SUB1 X))` = `F` by 0.1. From the first we get `X` $\neq$ `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.

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

Let us discuss the axioms generated by declaration of the shell `ADD1`. Roughly speaking, Axioms 0.1–0.3 are return-type declarations, Axioms 0.4–0.6 are about disjointness of types, Axiom 1 and Lemma L2 imply the axiom (nat1) from § 3.3, Axioms 2 and 3 imply axioms (nat2) and (nat3), respectively. Axioms 4 and 5.1–5.2 overspecify `SUB1`. Note that 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.

To the contrary, the arguments of the shell `CONS` (just as every shell argument without type restriction) are not forced like this, such that even objects of shells defined later (such as `PACK`) can be properly paired by the shell `CONS`; for instance, we have `(CAR (CONS NIL NIL))` = `NIL`, though `(CDR NIL)` = `0` (because `NIL` belongs to the shell `PACK`).

On the other hand, the untyped approach of the shell principle also admits to declare a shell `(LISTN X1 X2)` of the *lists of natural numbers only* — similar to the ones of § 3.4 — with a type predicate `LISTNP`, base object `(NILN)`, type restrictions `(NUMBERP X1)`, `(LISTNP X2)`, destructors `CARN`, `CDRN`, and default values `0`, `(NILN)`, respectively.

We have already discussed this *definition principle*[103] in detail in § 4. The definition of recursive functions has not changed compared to the Pure LISP Theorem Prover beside that a function definition is admissible now only after a termination proof, which proceeds as explained in § 4.3. To this end, Thm can applies the theorem of the well-foundedness of the lexicographic combination of two well-founded orderings; moreover, Thm axiomatically assumes the well-foundedness of the irreflexive ordering on the natural numbers.[104]

[103][Boyer and Moore, 1979, p. 44f.].

[104]See Page 52f. of [Boyer and Moore, 1979] for this informal axiomatic statement of the well-foundedness of `LESSP`.

As Thm is able to prove `(LESSP X (ADD1 X))` together with Axiom 1 and Lemma L2, the latter well-foundedness of `LESSP` would imply that Thm admits only the standard model of the natural numbers, as explained in Note 31.

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

Note that in the argument of Note 31, it is not possible to replace the reflexive transitive closure of the successor relation with the Thm-definable predicate

$$\{\ \texttt{Y} \mid \texttt{(NUMBERP Y)} = \texttt{T} \ \wedge\ \texttt{(LESSP Y X)} = \texttt{T}\ \},$$

because the latter will contain `0` as a minimal element even for a non-standard natural number `X`, which turns `LESSP` into a *proper* super-relation of that transitive closure.

Let us 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. The most relevant change is that previously established theorems have an effect on the current proof, provided that they have been activated for the purpose they can serve, in which case they are a applied in the reverse order of activation. This means that the performance of the the prover is not doomed to slow down when progressing to less and less elementary results, provided that the user develops the theory stepwise and activates the theorems properly. Moreover, there is also the chance to help the theorem prover directly with a new lemma that becomes necessary during a proof but is not found by the prover. This is actually the most crucial application of lemmas, because humans have a truly fascinating ability to *understand* a theory *semantically*, and because they can apply this ability to *"see"* why a sub-goal happens to be true. This ability to see the missing lemmas is actually the only aspect where humans still top the machine in the heuristics of typical induction proofs.

5.3.1 Simplification in Thm

The simplification in the Pure LISP Theorem Prover was described with the simplification in Thm already in mind, and so the reader should read that section of the description of the Pure LISP Theorem Prover before reading this.

Simplification is covered in Chapters VI–IX of [Boyer and Moore, 1979], and the reader interested in the details are strongly encouraged to read these very well-written descriptions of heuristic procedures.

To compensate for the extra complication of the untyped approach in a system that has many more interesting types than the Pure LISP Theorem Prover, Thm computes soft-typing rules 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.[105] These type computations are pervasively applied in all stages of the theorem prover, which we cannot discuss here in detail. If the return type of a function does not depend on the type of its arguments, the typing result can be expressed in the LISP logic language as a theorem. Let us see two examples on this.

EXAMPLE 10. *(continuing Example 6)*
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`.

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

EXAMPLE 11. 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))                                    □
```

The standard representation of an expression of propositional logic 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 definition 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 is not yet carrying that term. It is probably the first description of contextual rewriting in the history of logic, unless one counts the rudimentary contextual rewriting in the Pure LISP Theorem Prover as such.

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. The free universal variables of a clause have to be treated as constants during contextual rewriting. 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 $(t \neq \mathtt{F})$.[106]

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`-term 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,[107] it is moved to the top position (outside-in), by replacing each case

[106]Knuth–Bendix [Knuth and Bendix, 1970] completion of such ground term-rewrite systems terminates for any ground-complete term ordering and can be applied for an equivalence transformation of the context and to normalize the focus. This, however, is not discussed in [Boyer and Moore, 1979].

[107]The rewrite rules for `IF` whose soundness depends on termination are the following: `(IF X Y Y) = Y`, `(IF X X F) = X`, and, for Boolean `X` only, `(IF X T F) = X`. They are tested for applicability in the given order.

with itself in the `IF`'s context, such that the literal $C[$`(IF` t_0 t_1 t_2`)`$]$ is intermediately replaced with `(IF` t_0 $C[t_1]$ $C[t_2]$`)`, and then this literal splits its clause in two: one with the two literals `(NOT` t_0`)` and $C[t_1]$ in place of the old one, and one with t_0 and $C[t_2]$ instead.

Thm already eagerly removes variables in solved form: If the variable `X` does not occur in the term t, but the literal $(\mathtt{X}=t)$ occurs in a clause, then we can remove that literal after rewriting all occurrences of `X` in the clause to t. This removal is an equivalence transformation, because we have more than two objects and `X` is universally quantified over both of them, so for one instance $(\mathtt{X}=t)$ must be false.

It remains to describe the rewriting with function definitions and with lemmas activated for rewriting. The context of the clause is involved in the both evaluation processes.

Non-recursive function definitions are always expanded by Thm. Recursive function definitions are treated in a very similar style as in the Pure LISP Theorem Prover. The criteria on the expansion 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 the expansion is rejected not anymore by destructor terms introduced by the body and not removed by simplification, but actually by the occurrence of terms that (after simplification) do not occur as sub-terms already in the rest of the clause. This means that a new term `(CDR` t`)` in the simplified argument of a recursive function call, where `CDR` originates an argument of that recursive call in the body and not in the arguments of the tentative expansion, does not block expansion anymore if `(CDR` t`)` occurs already as a sub-term elsewhere in the clause. There are also further less important criteria to unblock expansion of recursive function definitions, such an increase of the number arguments that are constructor ground terms.[108]

Rewriting with lemmas activated 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 activated them explicitly for rewriting, and because rewrite lemmas have the form of conditional equations instead of unconditional ones. Simplification with lemmas activated for rewriting and the heuristics behind the process are nicely described in [Schmidt-Samoa, 2006c], where a rewrite lemma is not just activated for rewriting, but where the user can also mark the literals of the condition on how there are to be treated. In Thm there is no lazy rewriting with rewrite lemmas, i.e. no case splits are introduced to be able to apply the lemma. This means that all conditions of the rewrite lemma have to be shown to be fulfilled in the current context. As a compensation there is a process of backward chaining, i.e. 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

[108]See Page 119 of [Boyer and Moore, 1979] for the details and the remaining criteria.

which the previous conditions can be homeomorphically embedded.[109] There are provisions to instantiate extra variables of conditions eagerly, which is necessary because there are no existential variables.[110] Non-termination of evaluation via rewrite lemmas is achieved by simple term orderings, which in particular avoid looping with commutative lemmas.[111]

5.3.2 Destructor Elimination in Thm

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

EXAMPLE 12 (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 § 3.3:

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

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 ($\mathsf{p1}'$) (of p) (cf. § 3.3) twice.

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

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

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

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

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

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

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

Delete (Introduce) the leftmost equations of the condition by applying lemma ($\mathsf{s1}'$) (cf. § 3.3) 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.

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

Up to renaming of the variables, this is the destructor-style rule ($\mathsf{ack3}'$) of Example 8 (cf. § 5.2.6). □

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

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

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

Our data types are defined inductively over constructors.[112] Therefore constructors play the main rôle in our semantics, and practice shows that step cases of induction proofs work out much better with constructors than with the respective destructors, which are secondary defined operators in our semantics and have a more complicated case analysis in application.

There are two further positive effects of destructor elimination:

1. It tends to standardize the representation of a clauses in the sense that the numbers of occurrences of identical sub-terms tend to be increased.

2. Destructor elimination also brings the sub-term property in line with the sub-structure property; e.g., `Y` is both a sub-structure of `(CONS X Y)` and a sub-term of it, whereas `(CDR Z)` is a sub-structure of `Z` in case of `(LISTP Z)`, but not a sub-term.

Both effects improve the chances that the clause passes the follow-up stages of cross-fertilization and generalization with good success.[113]

For these reasons, the Pure LISP Theorem Prover did induction using step cases with constructors, such as $P(\mathsf{s}(x)) \Leftarrow P(x)$. Thm, however, does induction using step cases with destructors, such as

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

Therefore, destructor elimination was not so urgent in the Pure LISP Theorem Prover, simply because there were less 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, 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 chooses the style that is always applicable.

EXAMPLE 13 (A Generalized Destructor Without Constructor).
A generalized constructor that does not have a straightforward constructor is the function $\mathsf{delfirst}$ as defined in § 3.4. To verify the correctness of a deletion-sort algorithm based on $\mathsf{delfirst}$, a useful step case for an induction proof[114] is of the

[112] 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 car, cdr of the list of natural numbers $\mathsf{list(nat)}$ of § 3.4, we co-inductively obtain the data type of infinite streams over natural numbers.

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

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

form

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

A constructor version of this induction scheme would need something like an insertion function with an additional free variable for a natural number indicating the position of insertion, resulting in a step case that is so complicated that the proof would become much harder. □

Proper destructor functions take only one argument. The generalized destructor delfirst we have seen in Example 13 has actually two arguments; the second one is the *proper destructor argument* and the first is a *parameter*. After the elimination for a set of destructors, the terms at the parameter positions of the destructors are typically still present, where terms at the proper destructor argument are removed.

EXAMPLE 14 (Division with Remainder as a pair of Generalized Destructors).
In case of $y \neq 0$, we can construct each natural number in the form of $(q*y)+r$ with $\mathsf{less}(r,y)=\mathsf{true}$. The related generalized destructors are the 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 place that the definition $(\mathsf{p1}')$ and the lemma $(\mathsf{s1}')$ of § 3.3 have in Example 12, and which the definitions $(\mathsf{car1}')$, $(\mathsf{cdr1}')$ and the lemma $(\mathsf{cons1}')$ of § 3.4 have in the equivalence transformations between constructor and destructor style for lists, is here taken by the following definitions of div and rem and the following lemma on the constructor $\lambda q,r.\ ((q*y)+r)$:

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

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 reverse direction of Example 12 — 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 also 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.[115] □

To enable the form of elimination of generalized destructors described in Example 14, THM admits the user to activate lemmas of the form $(\mathsf{s1}')$, $(\mathsf{cons1}')$ or $(+9')$ as *elimination lemmas* to perform destructor elimination.

For clauses, this form is in general the following: The first literal of the clause is of the form $(t^{\mathrm{c}}=x)$, 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 side of the first literal and on left-hand sides of the

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

last $n{+}1$ literals of the clause, which are actually of the form

$$(y_0 \neq t^{\mathrm{d}}_0), \quad \ldots, \quad (y_n \neq t^{\mathrm{d}}_n),$$

for distinct *generalized destructor terms* $t^{\mathrm{d}}_0, \ldots, t^{\mathrm{d}}_n$. Thm adds one more restriction, namely that the generalized destructor terms have to consist of a function symbol applied to the disjoint list of all variables of the clause, beside $y_0, \ldots y_n$.

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^{d}_i, and then removing the literal $(y_i \neq t^{\mathrm{d}}_i)$. Thus, Thm would accept as elimination lemmas only the non-flattened versions of our elimination lemmas, such as $(\mathsf{s}1)$ instead of $(\mathsf{s}1')$ (cf. § 3.3) and such as $(\mathsf{cons}1)$ instead of $(\mathsf{cons}1')$ (cf. § 3.4).

The idea of application for destructor elimination in a given clause is, of course, the following: If the non-mentioned literals can be matched to literals of the given clause, and if $t^{\mathrm{d}}_0, \ldots, t^{\mathrm{d}}_n$ occur in the given clause as sub-terms, rewrite all their occurrences with $(y_0 \neq t^{\mathrm{d}}_0), \ \ldots, \ (y_n \neq t^{\mathrm{d}}_n)$ from right to left and then use the first literal of the elimination lemma from right to left for further normalization.

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.

In the destructor elimination of Thm, the most simple destructor (actually: the one defined first) is removed first, possibly in several steps. Then the clauses are re-introduced to the waterfall, but — to avoid looping — the terms with variables introduced by destructor elimination are blocked for simplification until the next induction.[116]

For more sophisticated details of destructor elimination in Thm, we have to refer the reader to Chapter X of [Boyer and Moore, 1979].

5.3.3 *(Cross-) Fertilization in* Thm

This stage has already been described in § 5.2.3 because there is no noticeable difference between the Pure LISP Theorem Prover and Thm here, beside some heuristic fine tuning.[117]

[116]See Page 139 of [Boyer and Moore, 1979].

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

5.3.4 Generalization in Thm

Thm adds only one new rule to the universally applicable heuristic rules for generalization on a term t mentioned already in § 3.8. This new rule makes sense in particular after the preceding stage of destructor elimination: Never generalize on a term t consisting in the application of a destructor function symbol to a list of distinct variables!

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. Now the synthesis of the predicate p that is to express essential properties of the term t symbolically, which now would result in the generalized clause $T[z]$, (NOT (p z)), is completely abandoned, because it could not meet its design intentions. Literals such as (NOT (p z)) are now added by two different means:

1. Assuming all literals of the clause $T[t]$ to be false, the bit-vector describing the soft type of t is computed and if only one bit is set, then, for the respective type predicate, say `NUMBERP`, a new literal is added to the clause, such as `(NOT (NUMBERP` t`))`.
2. The user can activate certain theorems as *generalization lemmas.* For instance, one of the disappointments of Boyer and Moore with the Pure LISP Theorem Prover was that it was not able to synthesize a predicate expressing sortedness for t being `(SORT X)`, for a sorting function `SORT`. Now, in Thm, the user can activate `(SORTEDP (SORT X))` as a generalization lemma, and then, if `(SORT X)` matches t, Thm will add the respective instance of `(NOT (SORTEDP (SORT X)))` to $T[t]$. In general, for the addition of such a literal `(NOT` t''`)`, a proper sub-term of a generalization lemma t' must match t; and the literal remains in the generalized clause only if the top function symbol of t does not occur in the literal anymore after replacing t with x.[118]

5.3.5 Elimination of Irrelevance in Thm

Before we come to the next stage "induction", we should remove irrelevant literals from clauses. After generalization, this is again a stage that may turn a valid clause into an invalid one. The main reason for taking this risk here is that the subsequent heuristic procedures for induction assume all literals to be relevant, and so they get confused by the presence of irrelevant literals and suggest the wrong step cases, which results in a failure of the induction proof. Moreover, if all literals seem to be irrelevant, then we know that we are going to prove a probably

[118]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'''`)` as well as in case of t being of the form `(FLATTEN` t'''`)`, but it will stay a literal of the generalized clause only in the former case, because in the latter case the first occurrence of `FLATTEN` is not removed by the generalization. See Page 156f. of [Boyer and Moore, 1979] for the details.

invalid clause, for which we should not do a costly induction, but had better ask the user to check whether he has supplied the theorem prover with an invalid lemma, possibly missing a side condition.[119]

Let us call two literals *connected* if they both have an occurrence of the same variable. Consider the partition of a clause into its equivalence classes w.r.t. the 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, the justification for this heuristic test is the following: If the partition 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 only of 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,[120] the justification for this heuristic test is the following: The bit-vector of this literal must contain the type `F`; otherwise the validity of the literal and the clause would have been recognized by the stage "simplification". This means that `F` is most probably a positive value for some combination of arguments.

5.3.6 Induction in THM

As we have seen in § 5.2.6, the *recursion analysis* in the PURE LISP THEOREM PROVER was very rudimentary. Compared to the recursion analysis in THM, one might not want to speak of a recursion analysis in the PURE LISP THEOREM PROVER at all, because there the whole information on the right-hand side of the recursive function definitions comes out of the poor feedback of the "evaluation" of the simplification stage at run time. Roughly speaking, this information consists only in the facts that the recursive function call is possibly reachable in the current

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

[120]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.

context and that a destructor symbol occurring as an argument of this recursive function call is not removed by the "evaluation" in the local environment.

In THM, however, the recursion analysis is done at the time the functions are defined, and the induction rule uses the results of this analysis for solving the three tasks of § 3.7.3.

The recursion analysis of THM is a perfect engineering solution of a not too difficult problem. In one way or the other, the solution found in THM has been part of all practice-oriented automated induction theorem proving systems ever since.

When the user defines a function f, any such system tries to find justifications along our description in § 4.3 why and under which conditions this function may terminate. For each such justification, the system attaches the following information to the function:

1. The set of *measured variables*, which consists of those variables of the weight-function w_f which actually occur in the body of its definition. For instance, in Example 7 of § 4.3, the measured subset is just $\{x\}$.
2. A function w'_f defined on the variables of the measured subset by the body of the definition of the weight function w_f.
3. The well-founded relation. In Example 7: less. (In THM: `LESSP`)
4. A list of pairs of the recursive function call and those governing conditions of its context that are relevant for the justification of termination. In Example 7 this list contains only the pair $\big(\ \mathsf{p}(x)+y,\ \ x\neq\mathsf{0}\ \big)$.

For the Ackermann function defined by $(\mathsf{ack1}'\text{–}3')$ in Example 8 of § 5.2.6, we get the measured subset $\{x,y\}$, the weight function $\mathsf{cons}(x,\mathsf{cons}(y,\mathsf{nil}))$ (in THM: `(CONS` x y`)`), the well-founded lexicographic ordering $\mathsf{lexlimless}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))),x,y)$, and the list consisting of the pairs $\big(\ \mathsf{ack}(\mathsf{p}(x),\mathsf{s}(\mathsf{0})),\ \ x\neq\mathsf{0}\ \big)$,
$\big(\ \mathsf{ack}(x,\mathsf{p}(y)),\ \ y\neq\mathsf{0}\ \big)$, and $\big(\ \mathsf{ack}(\mathsf{p}(x),\mathsf{ack}(x,\mathsf{p}(y))),\ \ x\neq\mathsf{0}\ \big)$.[121]

To find the justification automatically, THM admits the user to activate certain theorem 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'_f (which plays the rôle of the weight function on the measure subset), plus a condition without extra variables. Moreover, the arguments of w'_f in the second argument must be distinct variables. Certain induction lemmas are automatically activated when a shell is declared. For Example 7 of § 4.3, such a shell-declared induction lemma suffices, which is roughly

```
(LESSP (COUNT (SUB1 X)) (COUNT X))  ⇐  (NOT (ZEROP X)).
```

Note that `COUNT`, playing the rôle of w'_f, is a function that is the identity on the natural numbers and works on other shells similar to our function size from § 3.4.

This section on Induction needs revision and about 5 more pages.

[121]Note that for the definition of ack by the constructor style equations $(\mathsf{ack1}'\text{–}3')$ of § 3.3 nothing is essentially different. For definitions in constructor style, we have to take triples instead of pairs by adding the left-hand side of the defining equation. Moreover, for constructor style definitions in QUODLIBET, argument numbers take the place of the variables in the measured set.

5.4 NQTHM

Short section on quantification and that we cannot treat that here

5.5 ACL2 *and the practical challenges*

Short section

5.6 INKA *and other discontinued explicit induction projects*

Very short section similar to the one in [Bundy, 1999]

5.7 Limitations of Explicit Induction(?)

Short section, possibly to be omitted

6 ALTERNATIVE APPROACHES TO THE AUTOMATION OF INDUCTION

6.1 Proof Planning

Suggestions on how to overcome an envisioned dead end in automated theorem proving were summarized at the end of the 1980s under the keyword *proof planning*. Beside its human-science aspects,[122] the main idea of proof planning[123] is to add a smaller and more human-oriented *higher-level search space* to the theorem-proving system on top of the *low level search space* of the logic calculus. We do not cover this subject here, and refer the reader to the article by Alan Bundy and Jörg Siekmann in this volume.

6.2 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. We cannot cover this very well-documented subject here, but refer the reader to the monograph [Bundy *et al.*, 2005].[124] 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 as a lake with ripples (goal). 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, mirrored by the calmed surface of the lake.

6.3 Implicit Induction

Proof planning and rippling were applied to the automation of induction within the paradigm of explicit induction. The alternative approaches to mechanize mathematical induction *not* subsumed by explicit induction, however, are united under the name "implicit induction". Triggered by the success of Boyer and Moore [1979], work on these alternative approaches started already in the year 1980 in purely

[122]Cf. [Bundy, 1989].

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

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

equational theories.[125] A sequence of papers on technical improvements[126] 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 end of the 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.[127]

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:[128] Systems for proof by consistency run some Knuth–Bendix[129] or superposition[130] completion procedure that 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 an inconsistency.

Proof by consistency has shown to be not competitive with explicit induction in practice, mainly due to too many superfluous inferences, typically infinite runs, and too restrictive admissibility conditions. Roughly speaking, the conceptual flaw in proof by consistency is that, instead of finding a sufficient set of reasonable inferences, the research follows the paradigm of ruling out as many irrelevant inferences as possible.

Implicit Induction Ordering: In the early implicit-induction systems, induction proceeds over a syntactical term ordering, which typically cannot be made explicit in the sense that there would be some predicate in the logical syntax that denotes this ordering in the intended models of the specification. The semantical orderings of explicit induction, however, cannot depend precisely on the syntactical term structure of a weight w, but only on the value of w under an evaluation in the intended models.

The price one has to pay for the possibility to have induction orderings that can also depend on the precise syntax is surprisingly high for powerful inference systems.[131]

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

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

[127]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].

[128]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].

[129]Cf. [Gramlich and Lindner, 1991].

[130]Cf. [Ganzinger and Stuber, 1992].

[131]Cf. [Wirth, 1997].

The early implicit-induction systems needed such sophisticated term orderings,[132] 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,[133] after which the need for syntactical term orderings as induction orderings does not exist anymore.

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 termination in implicit induction need not be confined to single inference steps.[134] The induction rule of explicit induction combines several induction hypotheses in a single inference step. To the 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 § 3, the name *descente infinie* was coined for it in [Wirth, 2004]. Researchers introduced to this aspect by [Protzen, 1994] (entitled “Lazy Generation of Induction Hypotheses”) sometimes speak of “lazy induction” instead of *descente infinie.*

The interest in proof by consistency and implicit induction orderings today is either merely theoretical or merely historical, especially because these approaches cannot be combined with the paradigm of explicit induction. For more information on these viewpoints on implicit induction see the handbook article [Comon, 2001] and its partial correction [Wirth, 2005].

In § 6.4 we will show, however, how *Descente infinie* (“lazy induction”) goes well together with explicit induction and why we can hope that both the restrictions implied by induction axioms can be overcome and the usefulness of the excellent heuristic knowledge developed in explicit induction can be improved.[135]

6.4 QuodLibet

(along [Wirth, 2009] and [Schmidt-Samoa, 2006c])

[132]Cf. e.g. [Bachmair, 1988], [Steinbach, 1995].

[133]Cf. [Wirth and Becker, 1995].

[134]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.

[135]Cf. [Wirth, 2013].

7 BEYOND INDUCTION (SHORT)

7.1 Beyond Noetherian induction (Full axiom of choice instead of principle of dependent choices)

7.2 What the incredible success of NQTHM *meant for the fields of ATP and AI*

7.3 Lessons Learned beyond Induction

Boyer and Moore never gave a name to their provers and so they became most popular under the name *the Boyer–Moore theorem prover.* So here is an advice to the young researchers who want to become popular: Build a good system, but do not give it a name, so that people have to attach your name to it.

From the development of THM via the intermediate stage of the PURE LISP THEOREM PROVER, which somehow contains most essential stages, procedures, and heuristics of NQTHM in a rudimentary and partly inferior form, we can learn that it may be beneficial at an early stage of a true creation to proceed as follows:

1. Learn the essential ideas in a rudimentary form from the simpler, most frequent standard scenarios.
2. Implement and test these ideas.
3. Refine the resulting procedures and structures in later work.

This section on Lessons needs revision and possibly extension.

8 CONCLUSION

(very short)

In an interview on Thursday, Oct. 7, 2012, in their noble humbleness, Moore said the following, and Boyer agreed laughingly:

> "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."

This section on Lessons needs revision and extension.

ACKNOWLEGDGEMENTS

We would like to thank Anne O. Boyer, Robert S. Boyer, Warren A. Hunt, Matt Kaufmann, Marianeh Rezaei.

We could split the bibliography into subsections, but this seems to be inappropriate here, because alternative approaches to explicit induction have few references, and because the sections on INKA, NQTHM, ✓ERIFUN, ACL2, λCLAM would heavily overlap)

BIBLIOGRAPHY

[Abrahams *et al.*, 1980] Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors. *Conference Record of the 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.

[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.

[Aubin, 1979] Raymond Aubin. Mechanizing structural induction. *Theoretical Computer Sci.*, 9:329–345+347–362, 1979.

[Autexier *et al.*, 1999] Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. 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, 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.

[Baaz *et al.*, 1997] Matthias Baaz, Uwe Egly, and Christian G. Fermüller. Lean induction principles for tableaus. 1997. In [Galmiche, 1997, pp. 62–75].

[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 Kaufman (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, 2001] Klaus Barner. Das Leben Fermats. *DMV-Mitteilungen*, 3/2001:12–26, 2001.

[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.

[Benzmüller *et al.*, 2008] Christoph Benzmüller, Frank Theiss, Larry 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, 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].

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

[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.

[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..

[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, 1977] 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.

[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, 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. References are missing. `http://www.cs.utexas.edu/~boyer/quant.pdf`. Rev. version also in [Boyer and Moore, 1988a] and [Boyer and Moore, 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 in [Boyer and Moore, 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, 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 in [Boyer and Moore, 1988a].

[Boyer and Moore, 1990] Robert S. Boyer and J Strother Moore. A theorem prover for a computational logic. 1990. [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, 1971] Robert S. Boyer. *Locking: a restriction of resolution.* PhD thesis, The Univ. of Texas at Austin, 1971.

[Boyer, 1980] Anne O. Boyer. Sing a song of hacking (letter to the editor). *Psychology Today Magazine, One Park Avenue, New York 10016*, 14(6 (Nov.)), 1980.

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

[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].

[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.*, 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 (1993) **62**, pp. 185–253.

[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, 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.

[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].

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

[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.

[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.

[FOCS, 1980] *Proc. 21st Annual Symposium on Foundations of Computer Sci., Syracuse, 1980.* IEEE Press, 1980. `http://ieee-focs.org/`.

[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**, pp. 253–276, Academic Press (Elsevier), 1989.

[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 *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.

[Galmiche, 1997] Didier Galmiche, editor. *6th Int. Conf. on Tableaux and Related Methods, Pont-à-Mousson (France), 1997*, number 1227 in Lecture Notes in Artificial Intelligence. Springer, 1997.

[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, 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].

[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(omon) 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].

[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.

[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].

[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, 2011] David Hilbert and Paul Bernays. *Grundlagen der Mathematik I — Foundations of Mathematics I, Part A.* College Publications, London, 2011. Commented, first English translation of the 2nd edn. [Hilbert and Bernays, 1968] by Claus-Peter Wirthet.al., incl. the German facsimile and the annotation and translation of all deleted texts of 1st German edn. [Hilbert and Bernays, 1934]. 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. ISBN 978–1–84890–033–2.

[Hillenbrand and Löchner, 2002] Thomas Hillenbrand and Bernd Löchner. The next WALDMEISTER loop. 2002. In [Voronkov, 2002, pp. 486–500]. `http://www.waldmeister.org`.

[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**, pp. 239–266, Academic Press (Elsevier), 1982.

[Huet, 1980] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. of the ACM*, 27:797–821, 1980.

[Hutter and Stephan, 2005] Dieter Hutter and Werner Stephan, editors. *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg H. 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].

[Hutter, 1994] Dieter Hutter. Synthesis of induction orderings for existence proofs. 1994. In [Bundy, 1994, pp. 29–41].

[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 (1996) **16(1-2)**, pp. 79–111, Kluwer (Springer Science+Business Media).

[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**, pp. 1–33, 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 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.

[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, 1988] Robert A. Kowalski. The early years of logic programming. *Comm. ACM*, 31:38–43, 1988.

[Kraan *et al.*, 1995] Ina Kraan, David Basin, and Alan Bundy. *Middle-Out Reasoning for Synthesis and Induction.* 1995. DAI Research Paper No. 729, Dept. Artificial Intelligence, Univ. Edinburgh. Also in J. Automated Reasoning (1996) **16(1–2)**, pp. 113–145, Kluwer (Springer Science+Business Media).

[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. `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, 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`.

[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`.

[Lusk and Overbeek, 1988] Ewing Lusk and Ross Overbeek, editors. *9th Int. Conf. on Automated Deduction, 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, New Brunswick (NJ), 1996*, number 1104 in Lecture Notes in Artificial Intelligence. Springer, 1996.

[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.

[Meltzer, 1975] Bernard Meltzer. Department of A.I. – Univ. of Edinburgh. *ACM SIGART Bulletin*, 50:5, 1975.

[Moore, 1973] J Strother Moore. *Computational Logic: Structure Sharing and Proof of Program Properties.* PhD thesis, Dept. Artificial Intelligence, Univ. Edinburgh, 1973.

[Moore, 1975] 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. Received Dec. 1974, rev. March 1975.

[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].

[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.

[Pascal, 1954] Blaise Pascal. *Œuvres Complètes.* Gallimard, Paris, 1954. Jacques Chevalier (ed.).

[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].

[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].

[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] Alan Robinson and Andrei Voronkow, editors. *Handbook of Automated Reasoning.* Elsevier, 2001.

[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.

[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. *Journal of 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.

[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.

[Steele Jr., 1990] Guy L. Steele Jr.. COMMON LISP — *The Language.* Digital Press (Elsevier), 1990. 2nd edn. (1st edn. 1984).

[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. In [Kodratoff, 1988, pp. 565–570].

[Stickel, 1990] Mark E. Stickel, editor. *10th Int. Conf. on Automated Deduction, 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.

[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, København, 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, 1994] Christoph Walther. Mathematical induction. 1994. In [Gabbay *et al.*, 1994, pp. 127–228].

[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], `www.ags.uni-sb.de/~cp/p/cade94`.

[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.. Rev. extd. edn. of 1st part is [Wirth and Gramlich, 1994a], rev. edn. of 2nd part is [Wirth and Gramlich, 1994b].

[Wirth, 1991] Claus-Peter Wirth. Inductive theorem proving in theories specified by positive/negative-conditional equations. Diplomarbeit (Master's thesis), 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, `www.ags.uni-sb.de/~cp/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, 2005] 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, 2008] Claus-Peter Wirth. Hilbert's epsilon as an operator of indefinite committed choice. *J. Applied Logic*, 6:287–317, 2008. `http://dx.doi.org/10.1016/j.jal.2007.07.009`.

[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. `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. 2^{nd} edn. (1^{st} edn. 2006). `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. *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, 2012. Rev. edn., `http://arxiv.org/abs/1104.2444`.

[Wirth, 2013] Claus-Peter Wirth. Human-oriented inductive theorem proving by descente infinie — a manifesto. *Logic J. of the IGPL*, 20, 2013. Received July 11, 2011. Published online March 12, 2012, `http://dx.doi.org/10.1093/jigpal/jzr048`. To appear in print.

[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].