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 — for neither the *purely equational fragment* nor *propositional logic*.[2] Moreover, automation of theorem proving for *higher-order logic* has started becoming generally useful only during the last ten years.[3]

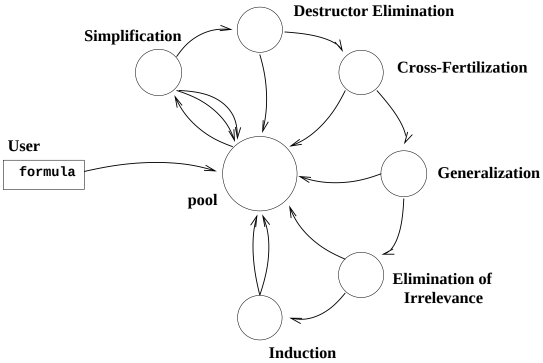

Figure 1. The Boyer–Moore Waterfall

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

*Second Readers:

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

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

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

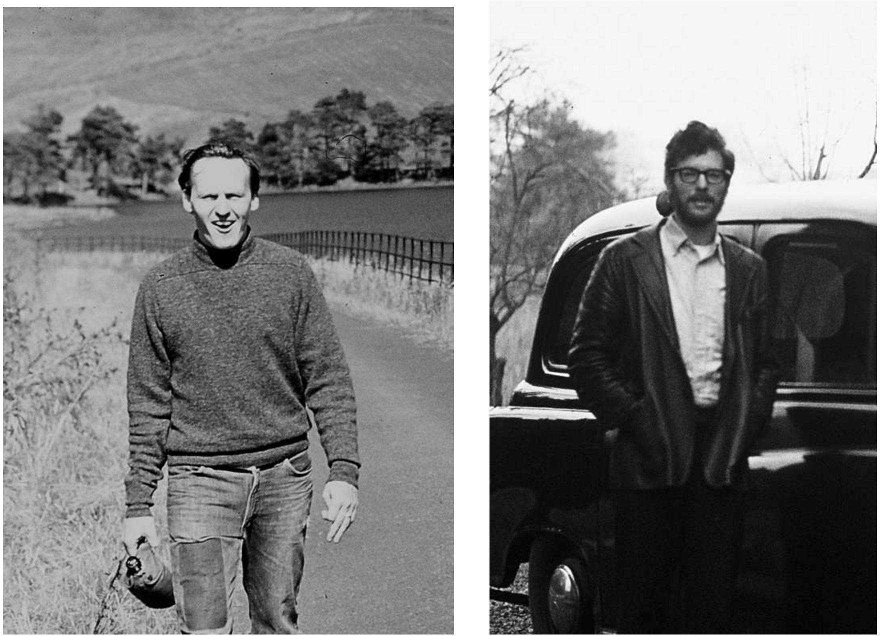

Figure 2. Robert S. Boyer (1971) (l.) and J Strother Moore (1972?) (r.)

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

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

The work on this breakthrough in the automation of inductive theorem proving was started quite deliberately in September 1972, by Robert S. Boyer and J Strother Moore, in Edinburgh, Scotland. Most of the crucial steps and their synergetic combination in the "waterfall" (cf. Figure 1) of their now famous theorem provers were developed in the span of a single year and implemented in their "PURE LISP THEOREM PROVER", presented at IJCAI in Stanford (CA) in August 1973,[5] and documented in Moore's PhD thesis [1973], defended in November 1973.

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

[4] See the last paragraph of § 5.4.

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

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

Being “warm blooded Texans”, they shared an office in the Metamathematics Unit at 9 Hope Park Square, Meadow Lane. The 18th century buildings at Hope Park Square were the center of Artificial Intelligence research in Britain at a time when the promises of AI were seemingly just on the horizon.[6] In addition to mainline work on mechanized reasoning by Rod M. Burstall, Robert A. Kowalski, Pat Hayes, Gordon Plotkin, Boyer, Moore, Mike J. C. Gordon, Alan Bundy, and (by 1973) Robin Milner, there was work on new programming paradigms, program transformation and synthesis, natural language, machine vision, robotics, and cognitive modeling. Hope Park Square received a steady stream of distinguished visitors from around the world, including J. Alan Robinson, John McCarthy, W. W. Bledsoe, Dana S. Scott, and Marvin Minsky. An eclectic series of seminars were on offer weekly to complement the daily tea times, where all researchers gathered around a table and talked about their current problems.

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

	Univ. Edinburgh	PhD	life time
	(time, Dept.)	(year, advisor)	(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	(1971–1973, CL)	(1971, Bledsoe)	(*1946)
Alan Bundy	(1971–today, CL)	(1971, Goodstein)	(*1947)
Robin Milner	(1973–1979, LFCS)	(no PhD)	(1934–2010)

CL = Metamathematics Unit (founded and headed by Bernard Meltzer)
(new name from late 1971 to Oct. 1974: Dept. of Computational logic)
(new name from Oct. 1974: Dept. of Artificial Intelligence)
MI = Experimental Programming Unit (founded and headed by Donald Michie)
(new name from 1966 to Oct. 1974: Dept. for Machine Intelligence and Perception)
(new name from Oct. 1974: Machine Intelligence Unit)
LFCS = Laboratory for Foundations of Computer Science

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

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

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

Their structure-sharing representation of derived clauses in a linear resolution system is just a stack of resolution steps. This suggests the idea of resolution being a kind of "procedure call."[7] Exploiting structure sharing, Boyer and Moore implemented a declarative LISP-like programming language called Baroque [Moore, 1973], a precursor to Prolog. They then implemented a LISP interpreter in Baroque and began to use their resolution engine to prove simple theorems about programs in LISP. However, while resolution was sufficient to prove such theorems as that "there is a list whose length is 3", the absence of a rule of induction prevented the proofs of more interesting theorems like the associativity of list concatenation.

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

Today's readers might have difficulty imagining the computing infrastructure in Scotland in the early 1970s. Boyer and Moore developed their software on an ICL–4130, with 64 kByte (128 kByte in 1972) core memory (RAM). Paper tape was used for archival storage. The machine was physically located in the Forrest Hill building of the University of Edinburgh, about 1 km from Hope Park Square. A rudimentary time-sharing system allowed several users at once to run lightweight applications from teletype machines at Hope Park Square. The only high-level programming language supported was POP–2, a simple stack-based list-processing language with an Algol-like syntax.[8]

Programs were prepared with a primitive text editor modeled on a paper tape editor: a disk file could be copied through a one byte buffer to an output file. By halting the copying and typing characters into or deleting characters from the buffer one could edit a file, a process that usually took several passes. Memory

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

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

limitations of the ICL–4130 prohibited storing large files in memory for editing. Early in their collaboration Boyer and Moore solved this problem by inventing what has come to be called the "piece table" whereby an edited document is represented by a linked list of "pieces" referring to the original file which remains on disk. Their "77-editor" [Boyer *et al.*, 1973] (written in 1971 and named for the disk track on which it resided) provided an interface like MIT's Teco, but with POP–2 as the command language.[9] It was thus with their own editor that Boyer and Moore wrote the code for the PURE LISP THEOREM PROVER.

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

2 METHOD OF PROCEDURE AND PRESENTATION

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

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

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

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

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

Because these achievements in the automation of inductive theorem proving manifest themselves mainly in the line of the Boyer–Moore theorem provers, we cannot avoid the confrontation of the reader with some more ephemeral forms of representation found in these software systems. In particular, we cannot avoid some small expressions in the list programming language LISP,[10] simply because the Boyer–Moore theorem provers we discuss in this article, namely the PURE LISP THEOREM PROVER, THM, NQTHM, and ACL2, all have *logics* based on a subset of LISP. Note that we do not necessarily refer to the *implementation language* of these software systems, but to the logic language used both for representation of formulas and for communication with the user.

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

> "We use a subset of LISP as our language because recursive list processing functions are easy to write in LISP and because theorems can be naturally stated in LISP; furthermore, LISP has a simple syntax and is universal in Artificial Intelligence. We employ a LISP interpreter to 'run' our theorems and a heuristic which produces induction formulas from information about how the interpreter fails. We combine with the induction heuristic a set of simple rewrite rules of LISP and a heuristic for generalizing the theorem being proved."[11]

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

> "The structure of the program is remarkably simple by artificial intelligence standards. This is primarily because the control structure is embedded in the syntax of the theorem. This means that the system does not contain two languages, the 'object language', LISP, and the 'meta-language', predicate calculus. They are identified. This mix of computation and deduction was largely inspired by the view that the two processes are actually identical. Bob Kowalski, Pat Hayes, and the nature of LISP deserve the credit for this unified view."[12]

This view was prevalent in the Metamathematics Unit by 1972. Indeed, "the Unit" was by then officially renamed the Department of Computational Logic.[6]

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

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

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

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

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

So a candidate to be considered today (besides a subset of LISP or of λ-calculus) is the functional programming language HASKELL.[13] 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*[14] 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 which its development is rooted,[15] were available in 1972. And still today, the applicative subset of COMMON LISP that provides the logic language for ACL2 (= $(\text{ACL})^2$ = $\underline{\text{A}}$ $\underline{\text{C}}$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 for reasons of efficiency: The applications of ACL2 in hardware verification and testing require a performance that is still at the very limits of today's computing technology. This challenging efficiency demand requires, among other aspects, that the logic of the theorem prover is so close to its own programming language that — after certain side conditions have been checked — the theorem prover can defer the interpretation of ground terms to the analogous interpretation in its own programming language.

For many of our illustrative examples in this article, however, we will use the higher flexibility and conceptual adequacy of positive/negative-conditional rewrite systems. They are so close to standard logic that we can dispense their semantics to the reader's intuition,[16] and they can immediately serve as an intuitively clear replacement of the Boyer–Moore *machines*.[17]

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

[14] See § 4.3 of this article.

[15] The general theory in 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].

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

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

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

For the same reason, we also prefer examples from arithmetic to examples from list theory, which might be considered to be especially supported by the LISP logic. The reader can find the famous examples from list theory in almost any other publication on the subject.[18]

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

Nevertheless, small LISP expression cannot be completely avoided because we have to describe the crucial parts of the historically most significant implementations and ought to show some of the advantages of LISP's untypedness.[19] The readers, however, do not have to know more about LISP than the following: A LISP term is either a variable symbol, or a function call of the form `(`$f\ \ t_1\ \cdots\ t_n$`)`, where f is a function symbol and $t_1, \ldots, t_n$ are LISP terms.

2.1 Organization of ~~T~~his Article

This article is further organized as follows.

§§ 3 and 4 offer a self-contained reference for the readers who are not familiar with the field of mathematical induction and its automation. In § 3 we introduce the essentials of mathematical induction. In § 4 we have to become more formal regarding recursive function definitions, their consistency, termination, and induction templates and schemes.

The main part is § 5, where we present the historically most important systems in automated induction, and discuss the details of software systems for explicit induction, with a focus on the 1970s. After describing the application context in § 5.1, we describe the following Boyer–Moore theorem provers: the PURE LISP THEOREM PROVER (§ 5.2) THM (§ 5.3) NQTHM (§ 5.4), and ACL2 (§ 5.5). The most noteworthy remaining explicit-induction systems are sketched in § 5.6.

Alternative approaches to the automation of induction that do not follow the paradigm of explicit induction are discussed in § 6.

After summarizing the lessons learned in § 7, we conclude with § 8.

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

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

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

3 MATHEMATICAL INDUCTION

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

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

Surprisingly, in this well-established and time-honored terminology, *mathematical induction* is not induction, but a special form of deduction for which — in the 19th century — the term "induction" was introduced and became standard in German and English mathematics.[20] In spite of this misnomer, for the sake of brevity, the term "induction" will always refer to mathematical induction in what follows.

Although it received its current name only in the 19th century, mathematical induction has been a standard method of every working mathematician at all times. It has been conjectured[21] that Hippasus of Metapontum (ca. 550 B.C.) ~~has~~ applied a form of mathematical induction that was later named *descente infinie (ou indéfinie)* by Fermat. We find another form of induction, nowadays called *structural induction*, in a text of Plato (427–347 B.C.).[22] In Euclid's famous "Elements" [ca. 300 B.C.], we find several applications of *descente infinie* and in a way also of structural induction.[23] 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).[24] Furthermore, structural induction was used by Francesco Maurolico (Messina) (1494–1575),[25] and by Blaise Pascal (1623–1662).[26] After an absence of more than one millennium (besides copying ancient proofs), *descente infinie* was reinvented by Pierre Fermat (160?–1665).[27]

[20] First in German (cf. Note 36), soon later in English (cf. [Cajori, 1918]).

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

[22] Cf. [Acerbi, 2000].

[23] An example for *descente infinie* is Proposition 31 of Vol. VII of the Elements. Moreover, we consider the proof of Proposition IX.8 of the Elements to be sound in a mathematical sense, though very poor in its linguistic and logical form. Thus, Proposition IX.8 must be a proof by structural induction. This is in accordance with [Freudenthal, 1953], but not with [Unguru, 1991] and [Acerbi, 2000]. See also [Fowler, 1994], [Wirth, 2010b, § 2.4] for a further discussion.

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

[25] Cf. [Bussey, 1917].

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

3.1 Well-foundedness and Termination

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

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

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

$<$ is an (irreflexive) *ordering* if it is irreflexive and transitive. There is not much difference between a well-founded relation and a well-founded ordering:[28]

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

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

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;[29] 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 the difference is relevant here because we cannot take the Axiom of Choice for granted in a discussion of foundations of induction, as will be explained in § 3.3.

3.2 The Theorem of Noetherian Induction

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

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

Again writing "$\mathsf{Wellf}(<)$" for "$<$ is well-founded", we can formalize the *Theorem of Noetherian Induction* as follows:[30]

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

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

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

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

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

The today commonly used term “Noetherian induction” is a tribute to the famous female German mathematician Emmy Noether (1882–1935). It occurs as the “Generalized principle of induction (Noetherian induction)” in [Cohn, 1965, p. 20]. Moreover, it occurs as Proposition 7 (“Principle of Noetherian Induction”) in [Bourbaki, 1968a, Chapter III, § 6.5, p. 190] — a translation of the French original in its second edition [Bourbaki, 1967, § 6.5], where it occurs as Proposition 7 (“principe de récurrence nœthérienne”).[31] We do not know whether “Noetherian” was used as a name of an induction principle before 1965;[32] in particular, it does not occur in the first French edition [Bourbaki, 1956] of [Bourbaki, 1967].[33]

3.3 An Induction Principle Stronger ~~T~~han Noetherian Induction?

Let us try to find a weaker replacement for the precondition of well-foundedness in Noetherian induction, in the sense that we try to replace “$\mathsf{Wellf}(<)$” in the Theorem of Noetherian Induction (N) in § 3.2 with some weaker property, which we will designate with “$\mathsf{Weak}(<,P)$” (such that $\forall P.\ \mathsf{Weak}(<,P)\ \Leftarrow\ \mathsf{Wellf}(<)$). This would result in the formula

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

If we assume (N'), however, we get the reverse $\forall P.\ \mathsf{Weak}(<,P)\ \Rightarrow\ \mathsf{Wellf}(<)$.[34] This means that a proper weakening is possible only w.r.t. *certain* P, and the Theorem of Noetherian Induction *is the strongest among those induction principles of the form* (N') *where* $\mathsf{Weak}(<,P)$ *does not depend on* P.

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

For every non-empty $<$-chain C [without a $<$-minimal element]:

$$\exists v\,{\in}\,C.\ P(v) \quad\Leftarrow\quad \forall u{<}C.\ P(u).$$

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

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

[32] In 1967, “Noetherian Induction” was not generally used yet 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*. “Complete induction”, however, is a most confusing name hardly used in English, typically referring to *course-of-values induction*, cf. e.g. `http://en.wikipedia.org/wiki/Mathematical_induction#Complete_induction`. Moreover, “complete induction” is the literal translation of the German technical term “vollständige Induction”, which traditionally means structural induction (cf. Note 36) — and these three kinds of mathematical induction are different from each other.

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

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

This means that the axiomatic status of induction principles ranges from the Theorem of Noetherian Induction up to the Axiom of Choice. If we took the Axiom of Choice for granted, this difference in status between a theorem and an axiom would collapse and our discussion of the axiomatic status of mathematical induction would be deteriorated. So the care with which we distinguished termination of the reverse relation from well-foundedness in § 3.1 is justified.

3.4 The Natural Numbers

The field of application of mathematical induction most familiar in mathematics is the domain of the natural numbers 0, 1, 2, ... Let us formalize the natural numbers with the help of two 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 in this article that the variables x, y always range over the natural numbers, and that free variables in formulas are implicitly universally quantified (as standard in mathematics), such that, for example, a formula with the free variable x can be seen as having the implicit outermost quantifier $\forall x : \mathsf{nat}$.

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

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

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

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

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

[34] *Proof.* Let $<\!\restriction_A$ denote the range restriction of $<$ to A (i.e. $u{<}\!\restriction_A v$ if and only if $u < v \in A$). Let us take $P(w)$ to be $\mathsf{Wellf}(<\!\restriction_{A(w)})$ for $A(w) := \{\ w' \mid w' <^* w\ \}$. Then the reverse implication follows from (N') because $P(v) \Leftarrow \forall u{<}v.\ P(u)$ holds for any v,[35] and $\forall w.\ P(w)$ implies $\mathsf{Wellf}(<)$.

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

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

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

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

Richard Dedekind (1831–1916) proved the Axiom of Structural Induction (S) for his model of the natural numbers in [Dedekind, 1888], where he states that the proof method resulting from the application of this axiom is known under the name "vollständige Induction".[36]

Now we can go on by defining — in two equivalent[37] ways — the destructor function $\mathsf{p} : \mathsf{nat} \to \mathsf{nat}$, returning the $\underline{\text{p}}$redecessor of a positive natural number:

1. In $(\mathsf{p}1)$ in *constructor style*, where constructor terms may occur on the left-hand side of the positive/negative-conditional equation as arguments of the function being defined.
2. In $(\mathsf{p}1')$ in *destructor style*, where only variables may occur as arguments on the left-hand side.

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

$$(\mathsf{p}1) \qquad \mathsf{p}(\mathsf{s}(x)) = x$$

$$(\mathsf{p}1') \qquad \mathsf{p}(x') = x \ \Leftarrow\ x' = \mathsf{s}(x)$$

Let us define some recursive functions over the natural numbers, such as addition and multiplication $+, * : \mathsf{nat}, \mathsf{nat} \to \mathsf{nat}$, the irreflexive ordering of the natural numbers $\mathsf{lessp} : \mathsf{nat}, \mathsf{nat} \to \mathsf{bool}$ (see § 3.5 for the data type bool of Boolean values), and the Ackermann function $\mathsf{ack} : \mathsf{nat}, \mathsf{nat} \to \mathsf{nat}$:[38]

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

$$\begin{array}{lll}
(\mathsf{lessp}1) & \mathsf{lessp}(x, 0) & = \mathsf{false} \\
(\mathsf{lessp}2) & \mathsf{lessp}(0, \mathsf{s}(y)) & = \mathsf{true} \\
(\mathsf{lessp}3) & \mathsf{lessp}(\mathsf{s}(x), \mathsf{s}(y)) & = \mathsf{lessp}(x, y)
\end{array}$$

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

[36] In the tradition of Aristotelian logic, the technical term "vollständige Induction" (in Latin: "inductio completa", cf. e.g. [Wolff, 1740, Part I, § 478, p. 369]) denotes a complete case analysis, cf. e.g. [Lambert, 1764, Dianoiologie, § 287; Alethiologie, § 190]. Its misuse as a designation of structural induction originates in [Fries, 1822, p. 46f.], and was perpetuated by Dedekind. Its literal translation "complete induction" is misleading, cf. Note 32. By the 1920s, "vollständige Induction" had become a very vague notion that is best translated as "mathematical induction", as done in [Heijenoort, 1971, p.130] and as it is standard today, cf. e.g. [Hilbert and Bernays, 2011, Note 23.4].

[37] For the equivalence transformation between constructor and destructor style see Example 15 in § 5.3.2.

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

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

Now the natural numbers can be specified up to isomorphism either by[39]

- $(\mathsf{nat2})$, $(\mathsf{nat3})$, 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).[40]

Immediate consequences of the axiom $(\mathsf{nat1})$ and the definition $(\mathsf{p1})$ are the lemma $(\mathsf{s1})$ and its flattened version $(\mathsf{s1}')$:

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

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

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

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

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

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

Moreover — to see the necessity of doing induction on several variables in parallel — we will present[41] the more complicated proof of the strengthened transitivity of the irreflexive ordering of the natural numbers, i.e. of

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

We will also prove the commutativity lemma $(+3)$[42] and the simple lemma $(\mathsf{ack4})$ about the Ackermann function:[43]

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

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

[39] Cf. [Wirth, 2004, § 1.1.2].

[40] Pieri [1908] stated these axioms informally 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 [1908] 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 statement that there is at most one s-minimal element:

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

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

[41] We will prove $(\mathsf{lessp7})$ twice: once in Example 3 in § 3.7, and again in Example 12 in § 5.2.6.

[42] We will prove $(+3)$ twice: once in Example 2 in § 3.7, and again in Example 4 in § 3.8.1.

[43] We will prove $(\mathsf{ack4})$ in Example 5 in § 3.9.

3.5 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 today from their occurrence in higher-level programming languages.

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

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 get only the following two axioms by analogy to the axioms for the natural numbers. We globally declare the variable $b : \mathsf{bool}$; so b will always range over the Boolean values.

(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. § 3.4) is not perfect: An axiom (bool3) analogous to (nat3) cannot exist because there are no constructors for bool that take arguments. Moreover, an axiom analogous to (S) 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

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

for the empty list, and the function symbol

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

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

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

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

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

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

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

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

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

(length1) $\quad \mathsf{length}(\mathsf{nil}) \qquad = 0$

(length2) $\quad \mathsf{length}(\mathsf{cons}(x, l)) = \mathsf{s}(\mathsf{length}(l))$

(count1) $\quad \mathsf{count}(\mathsf{nil}) \qquad = 0$

(count2) $\quad \mathsf{count}(\mathsf{cons}(x, l)) = \mathsf{s}(x + \mathsf{count}(l))$

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

1. There is an additional axiom ($\mathsf{list(nat)}3_1$), which has no analog among the axioms of the natural numbers.
2. Neither 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 Pieri's second axiom for the natural numbers.[44]
3. The latter axiom is weaker than each of the two axioms
$$\mathsf{Wellf}(\lambda l, k.\ (\mathsf{lessp}(\mathsf{length}(l), \mathsf{length}(k)) = \mathsf{true})),$$
$$\mathsf{Wellf}(\lambda l, k.\ (\mathsf{lessp}(\mathsf{count}(l), \mathsf{count}(k)) = \mathsf{true})),$$
which state the well-foundedness of bigger[45] relations. In spite of their relative strength, the well-foundedness of these relations is already implied by the well-foundedness that Pieri used for his specification of the natural numbers.

Therefore, the lists of natural numbers can be specified up to isomorphism by a specification of the natural numbers up to isomorphism (see § 3.4), 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,[46]
- ($\mathsf{list(nat)}1$), ($\mathsf{length}1$–2) — refining the style of Pieri.[47]

Today it is standard to avoid higher-order axioms in the way exemplified in the last of these three items,[48] and to get along with one second-order axiom for the natural numbers, or even with the first-order instances of that axiom.

[44] See § 3.4 for Pieri's specification of the natural numbers. The axioms ($\mathsf{list(nat)}3_1$) and ($\mathsf{list(nat)}3_2$) are not implied because all axioms besides ($\mathsf{list(nat)}3_1$) 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 sizes.

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

[46] 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) implies $\mathsf{Wellf}(\lambda l, k.\ \exists x.\ (\mathsf{cons}(x,l) = k))$.

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

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

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

$$
\begin{array}{ll}
(\mathsf{car}1) & \mathsf{car}(\mathsf{cons}(x,l)) = x \\
(\mathsf{cdr}1) & \mathsf{cdr}(\mathsf{cons}(x,l)) = l \\
(\mathsf{car}1') & \qquad \mathsf{car}(l') = x \ \Leftarrow\ l' = \mathsf{cons}(x,l) \\
(\mathsf{cdr}1') & \qquad \mathsf{cdr}(l') = l \ \Leftarrow\ l' = \mathsf{cons}(x,l)
\end{array}
$$

$$
\begin{array}{llll}
(\mathsf{mbp}1) & \mathsf{mbp}(x,\mathsf{nil}) & = \mathsf{false} & \\
(\mathsf{mbp}2) & \mathsf{mbp}(x,\mathsf{cons}(y,l)) & = \mathsf{true} & \Leftarrow\ x = y \\
(\mathsf{mbp}3) & \mathsf{mbp}(x,\mathsf{cons}(y,l)) & = \mathsf{mbp}(x,l) & \Leftarrow\ x \neq y
\end{array}
$$

$$
\begin{array}{llll}
(\mathsf{delfirst}1) & \mathsf{delfirst}(x,\mathsf{cons}(y,l)) & = l & \Leftarrow\ x = y \\
(\mathsf{delfirst}2) & \mathsf{delfirst}(x,\mathsf{cons}(y,l)) & = \mathsf{cons}(y,\mathsf{delfirst}(x,l)) & \Leftarrow\ x \neq y
\end{array}
$$

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

$$
\begin{array}{ll}
(\mathsf{cons}1) & \mathsf{cons}(\mathsf{car}(l'),\mathsf{cdr}(l')) = l' \ \Leftarrow\ l' \neq \mathsf{nil} \\
(\mathsf{cons}1') & \qquad \mathsf{cons}(x,l) = l' \ \Leftarrow\ l' \neq \mathsf{nil}\ \wedge\ x = \mathsf{car}(l')\ \wedge\ l = \mathsf{cdr}(l')
\end{array}
$$

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

$$
\begin{array}{llll}
(\mathsf{lexless}1) & \mathsf{lexless}(l,\mathsf{nil}) & = \mathsf{false} & \\
(\mathsf{lexless}2) & \mathsf{lexless}(\mathsf{nil},\mathsf{cons}(y,k)) & = \mathsf{true} & \\
(\mathsf{lexless}3) & \mathsf{lexless}(\mathsf{cons}(x,l),\mathsf{cons}(y,k)) & = \mathsf{lexless}(l,k) & \Leftarrow\ x = y \\
(\mathsf{lexless}4) & \mathsf{lexless}(\mathsf{cons}(x,l),\mathsf{cons}(y,k)) & = \mathsf{lessp}(x,y) & \Leftarrow\ x \neq y
\end{array}
$$

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

Such lexicographic combinations play an important rôle in well-foundedness arguments of induction proofs, because they combine given well-founded orderings into new well-founded orderings, provided there is an upper bound for the length of the list:[49]

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

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

[49] 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 lists of different lengths typically arises in mutual induction proofs where the two induction hypotheses have a different number of free variables at measured positions. See [Wirth, 2004, § 3.2.2] for a nice example.

3.6 The Standard High-Level Method of Mathematical Induction

A mathematical proof method cannot be completely captured by its non-procedural logic formalization; and so we need effective heuristics for actually finding proofs by 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,[50] require considerable ingenuity and have to be carried out.

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

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

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

(Induction-Hypotheses Task)
to find the numerous induction hypotheses;[51] 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.[52]

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

[50] 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].

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

[52] 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].

3.7 Descente Infinie

The soundness of the induction method of § 3.6 is most easily seen when the argument is structured as a proof by contradiction, assuming a counterexample. For Fermat's historic reinvention of the method, it is thus just natural that he developed the method in terms of assumed counterexamples.[53] Here is Fermat's Method of *Descente Infinie* in modern language, very roughly speaking:

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

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

Nowadays every logician immediately realizes that a formalization of the method of *descente infinie* is obtained from the Theorem of Noetherian Induction (N) (cf. § 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,[54] 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[55] as a synonym for the modern standard high-level method of mathematical induction described in § 3.6.

Let us now prove the lemmas (+3) and (lessp7) of § 3.4 (in the axiomatic context of § 3.4) by *descente infinie*, seen as the standard high-level method of mathematical induction described in § 3.6.

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

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

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

[55] In general, in the tradition of [Wirth, 2004], *descente infinie* is nowadays taken as a synonym for the standard high-level method of mathematical induction as described in § 3.6. This way of using the term *"descente infinie"* is found in [Brotherston and Simpson, 2007; 2011], [Voicu and Li, 2009], [Wirth, 2005a; 2010a; 2013; 2012c].

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

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

By application of the Theorem of Noetherian Induction (N) (cf. § 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 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 $y = \mathsf{s}(y')$, which leaves us with the 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 instantiate the induction hypothesis that is available in the context[56] given by our above formula in four different forms, namely we instantiate (x'',y'') with $(x',\mathsf{0})$, $(\mathsf{0},y')$, $(x',\mathsf{s}(y'))$, and $(\mathsf{s}(x'),y')$, 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 follow directly from the reflexivity of equality, whereas the last goal needs also an application of our induction hypothesis: This time we have to instantiate (x'',y'') with (x',y').

Finally, we instantiate our induction ordering $<$ to the lexicographic combination of length less than 3 of the ordering of the natural numbers. If we read our pairs as two-element lists, i.e. (x'',y'') as $\mathsf{cons}(x'',\mathsf{cons}(y'',\mathsf{nil}))$, then we can set $<$ to $\lambda l,k.\ (\mathsf{lexlimless}(l,k,\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) = \mathsf{true})$, which is well-founded according to $(\mathsf{lexlimless2})$ (cf. § 3.5). 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. □

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

EXAMPLE 3 (Proof of ($\mathsf{lessp7}$) by *descente infinie*).
In the previous proof in Example 2 we made the application of the Theorem of Noetherian Induction most explicit, and so its presentation was rather formal w.r.t. the underlying logic.

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

To prove the strengthened transitivity of lessp as expressed in lemma ($\mathsf{lessp7}$) in the axiomatic context of § 3.4, we then have to show

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

If we apply the axiom ($\mathsf{nat1}$) — with the intention to reduce the last literal — once to y and once to z, then, after reduction with ($\mathsf{lessp1}$), 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 ($\mathsf{lessp3}$) to

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

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

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

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

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

$$\begin{aligned} \mathsf{lessp}(\mathsf{0}, z') &\;\Leftarrow\; \mathsf{lessp}(y', z') \\ \mathsf{lessp}(\mathsf{s}(x'), z') &\;\Leftarrow\; \mathsf{lessp}(x', y') \wedge \mathsf{lessp}(y', z'), \end{aligned}$$

respectively. The first is trivial by ($\mathsf{lessp1}$), ($\mathsf{lessp2}$) 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 select any of the variables of the original conjecture w.r.t. the irreflexive ordering on the natural numbers or w.r.t. the successor relation.

This completes the proof of the conjecture.

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

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

3.8 Explicit Induction

3.8.1 From the Theorem of Noetherian Induction to Explicit Induction

To admit the realization of the standard high-level method of mathematical induction as described in § 3.6, a proof calculus should have an explicit concept of an induction hypothesis. Moreover, it has to cope in some form with the second-order variables P and $<$ in the Theorem of Noetherian Induction (N) (cf. § 3.2), and with the second-order variable Q in the definition of well-foundedness $(\mathsf{Wellf}(<))$ (cf. § 3.1).

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

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

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

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

At a first glance, it seems highly unlikely that there could be any framework of proof-search heuristics in which such an induction rule could succeed in implementing all applications of the Theorem of Noetherian Induction, simply because this rule has to solve the two hard tasks of an induction proof, namely the Induction-Hypotheses Task and the Induction-Ordering Task (cf. § 3.6), right at the beginning of the proof attempt, before the proof has been sufficiently developed to exhibit its structural difficulties.

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

- No new symbols have to be introduced during the proof, such as the ones of quantifier elimination. Therefore, the required instances of the induction hypothesis can already be denoted when the induction rule is applied.[57]
- A general peculiarity of induction,[58] namely that the formulation of lemmas often requires the definition of new recursive functions, is aggravated by the weakness of the logic; and the user actually provides further guidance for the induction rule via these new function definitions.[59]

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

EXAMPLE 4 (Proof of (+3) by explicit induction).
Let us prove (+3) in the context of § 3.4, just as we have done already in Example 2 (cf. § 3.7), but now with the induction rule as the only way to apply the Theorem of Noetherian Induction.

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

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

Based, roughly speaking, on a termination analysis for the function $+$, the heuristic of the induction rule of explicit induction suggests to instantiate $<$ to $\lambda(x'',y''),(x,y).\ (\mathsf{s}(x'')=x)$. As this relation is known to be well-founded, the induction rule reduces the task 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. Thus, instances of the induction hypothesis can no longer be applied in the further proof.

The induction rules of the Boyer–Moore theorem provers are not able to find the many instances we applied in the proof of Example 2. This is different for a theoretically more powerful induction rule suggested by Christoph Walther (*1950), which actually finds the proof of Example 2.[60] In general, however, for harder conjectures, a simulation of *descente infinie* by the induction rule of explicit induction

[57] Cf. Note 61.

[58] See item 2 of § 3.10.

[59] Cf. § 8.

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

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

In general, however, Walther's induction rule is less successful within the heuristic framework of the Boyer–Moore waterfall (cf. Figure 1).

would require an arbitrary look-ahead into the proofs, depending on the size of the structure of these proofs; thus, because the induction rule is understood to have a limited look-ahead into the proofs, such a simulation would not fall under the paradigm of explicit induction anymore. Indeed, the look-ahead of induction rules into the proofs is typically not more than a single unfolding of a single occurrence of a recursive function symbol, for each such occurrence in the conjecture.

Note that the two above goals of the base and the step case can also be obtained by reducing the input conjecture with an instance of axiom (S) (cf. § 3.4), i.e. with the Axiom of Structural Induction over $\mathsf{0}$ and s. Nevertheless, the induction rule is in general able to produce much more complicated base and step cases than reduction with simple instances of (S).

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

The second goal is simplified to

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

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

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

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

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

Although the two proofs of the very simple conjecture (+3) given in Examples 2 and 4 can only give a very rough idea on the advantage of *descente infinie* for hard induction proofs,[61] these two proofs nicely demonstrate how the induction rule of explicit induction manages to prove simple theorems very efficiently and with additional benefits for the further performance of the simplification procedure.

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

3.8.2 Theoretical Viewpoint on Explicit Induction

From a theoretical viewpoint, we should be aware of the possibility that the intended models of specifications in explicit-induction systems, say for the natural numbers, also include non-standard models, where — contrary to the higher-order specifications of Peano and Pieri — there may be **Z**-chains in addition to the natural numbers **N**.[62] These **Z**-chains cannot be excluded because the inference rules realize only first-order deductive reasoning, except for the induction rule to which all applications of the Theorem of Noetherian Induction are confined and which does not use any higher-order properties, but only well-founded orderings that are defined in the first-order logic of the explicit-induction system; see also Note 131.

3.8.3 Practical Viewpoint on Explicit Induction

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

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

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

1. Choose some of the variables in the conjecture as *induction variables*, and split the conjecture into several base and step cases, based on the induction variables' demand on which governing conditions and constructor substitutions[63] have to be added to be able to unfold — without further case analysis — some of the recursive function calls that contain the induction variables as direct arguments.

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

[61] For some of the advantages of *descente infinie*, see Example 12 in § 5.2.6, and especially the more difficult, complete formal proof of M. H. A. Newman's famous lemma in [Wirth, 2004, § 3.4], where the reverse of a well-founded relation is shown to be confluent in case of local confluence *by induction w.r.t. this well-founded relation itself.* The induction rule of explicit induction cannot be applied here because an eager induction hypothesis generation is not possible: The required instances of the induction hypothesis contain δ-variables that can only be generated later during the proof by quantifier elimination.

Though confluence is the Church–Rosser property, the Newman Lemma has nothing to do with the Church–Rosser Theorem stating the confluence of the rewrite relation of $\alpha\beta$-reduction in untyped λ-calculus, which has actually been verified with a Boyer–Moore theorem prover in the first half of the 1980s by Shankar [1988] (see the last paragraph of § 5.4 and Note 166) following the short Tait/Martin-Löf proof found e.g. in [Barendregt, 2012, p. 59ff.]. Unlike the Newman Lemma, Shankar's proof proceeds by structural induction on the λ-terms, not by Noetherian induction w.r.t. the reverse of the rewrite relation; indeed, untyped λ-calculus is not terminating.

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

The actual realization of these tasks in the induction rule, however, is quite different from these expectations: Induction variables play only a very minor rôle toward the end of the procedure (in the deletion of flawed induction schemes, cf. § 5.3.8), the focus is on complete step cases including eagerly generated induction hypotheses, and the complementing bases case are generated only ~~in~~ the very end.[64]

3.9 Generalization

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

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

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

Generalization is hardly needed when input conjectures are supplied by humans. As we have seen in Example 4 of § 3.8.1, however, explicit induction often has to start another induction during the proof, and then the secondary, machine-generated input conjecture often requires generalization.

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

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

The removal of irrelevant side conditions is intuitively clear. For formulas in clausal form, it simply means to remove irrelevant literals. More interesting are the heuristics of its realization, which we discuss in § 5.3.5.

The less clear process of 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 both sides of the same positive equation or in literals of different polarity, say in a conclusion and a condition of an implication.

To avoid *over-generalization*, subterms are to be preferred to their super-terms, 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, a direct argument of a logical operator, or the first argument

[63] This adding of constructor substitutions refers to the application of axioms like (`nat1`) (cf. § 3.4), and is required whenever constructor style either is found in the recursive function definitions or is to be used for the step cases. In the Pure LISP Theorem Prover, only the latter is the case. In Thm, none of this is the case.

[64] See, e.g., Example 10 of § 4.7.

of a conditional (IF). In any of these cases, the information loss by generalization is typically so high that it probably results in an invalid conjecture.

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

EXAMPLE 5 (Proof of (ack4) by Explicit Induction and Generalization).
Let us prove (ack4) in the context of § 3.4 by explicit induction. It is obvious that such a proof has to follow the definition of ack in the three cases (ack1), (ack2), (ack3), using the termination ordering of ack, which is just the lexicographic combination of its arguments. So the induction rule of all Boyer–Moore theorem provers except the Pure LISP Theorem Prover reduces the input formula (ack4) to the following goals:[65]

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

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

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

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

$$\begin{array}{l}
\mathsf{lessp}(0, z) = \mathsf{true} \;\Leftarrow\; \left(\; \mathsf{lessp}(\mathsf{s}(0), z) = \mathsf{true} \;\wedge\; z = \mathsf{ack}(x', \mathsf{s}(0)) \;\right);\\
\mathsf{lessp}(\mathsf{s}(y'), z_2) = \mathsf{true} \;\Leftarrow\; \left(\begin{array}{ll} & \mathsf{lessp}(y', z_1) = \mathsf{true} \;\wedge\; \mathsf{lessp}(z_1, z_2) = \mathsf{true}\\ \wedge & z_1 = \mathsf{ack}(\mathsf{s}(x'), y') \;\wedge\; z_2 = \mathsf{ack}(x', z_1)\end{array}\right).
\end{array}$$

Now the first follows from applying (nat1) to z. Before we can prove the second by another induction, however, we have to generalize it to the lemma (lessp7) of § 3.4 by deleting the last two literals from the condition. □

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

It should be noted that all Boyer–Moore theorem provers except the Pure LISP Theorem Prover are able to do this whole proof completely automatically and invent the lemma (lessp7) by generalization of the second step case; and they do this even when they work with an arithmetic theory that was redefined, so that no decision procedures or other special knowledge on the natural numbers can be used by the system. Moreover, as shown in § 3.3 of [Wirth, 2004], in a slightly richer

[65] See Example 10 of § 4.7 on how these step cases are actually found in explicit induction.

logic, these heuristics would additionally admit to synthesize the lower bound in the first argument of lessp from the input conjecture $\exists z.\ (\mathsf{lessp}(z, \mathsf{ack}(x, y)) = \mathsf{true})$, simply because lessp does not contribute to the choice of the base and step cases.

3.10 Proof-Theoretical Peculiarities of Mathematical Induction

The following two proof-theoretical peculiarities of induction compared to first-order deduction may be considered noteworthy:[66]

1. A calculus for arithmetic cannot be complete, simply because the theory of the arithmetic of natural numbers is not enumerable.[67]
2. According to Gentzen's Hauptsatz,[68] 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.[69] 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.[70]

3.11 Conclusion

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

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

To express the relevant notions in these data types, we need *recursion*, a method of definition, which we have often used in this section intuitively. We did not discuss its formal admissibility requirements, however, which we will do in § 4, with a focus on modes of recursion that admit an effective consistency test, including termination aspects such as induction templates and schemes.

[66] Note, however, that these peculiarities of induction do not make a difference to first-order deductive theorem proving *in practice*. See Notes 67 and 70.

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

[68] Cf. [Gentzen, 1935].

[69] Cf. [Kreisel, 1965].

[70] 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]).