Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern

SEKI - REPORT

Completion Based Inductive Theorem Proving: A Case Study in Verifying Sorting Algorithms

B. Gramlich

SEKI Report SR-90-04

Completion Based Inductive Theorem Proving - A Case Study in Verifying Sorting Algorithms

Bernhard Gramlich
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
6750 Kaiserslautern
West Germany
E-mail: gramlich@uklirb.uucp

This work was supported by the Deutsche Forschungsgemeinschaft, SFB 314 (D4)

Completion Based Inductive Theorem Proving - A Case Study in Verifying Sorting Algorithms

Bernhard Gramlich
Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
6750 Kaiserslautern
West Germany

Abstract

The focus of this paper lies on practical aspects of completion based inductive theorem proving in equational theories. As domain of interest we concentrate on the mechanically supported verification of a couple of sorting algorithms. The experiments have been performed with UNICOM, an inductive theorem prover based on refined unfailing completion techniques. We summarize these experiments, point out important technical and conceptual aspects and illustrate them by numerous examples. In particular we discuss and exemplify the kind of intelligent proof engineering required for succeeding in non-trivial verification problems.

1. Introduction

Equational reasoning is fundamental for many fields of computer science like functional programming, abstract data type specifications, program synthesis and verifications. Within these applications one is usually interested in a standard initial model of a given equational axiom system and not in all its models. Proof methods for this initial model are usually based on induction schemas, e.g. for structural induction (cf., [Bu 69], [BM 79], [Au 79]).

Within the last decade an alternative approach based on rewriting and completion techniques has been developed ([Mu 80], [Go 80], [HH 80]) and refined in many ways in the meanwhile (cf. [JK 86], [Fr 86], [Gö 87], [Kü 87], [Ba 88]). This approach has some very attractive and promising characteristics compared to classical approaches for explicit inductive theorem proving based on induction schemas, which will be pointed out later on. We shall present an implementation of this proof technique and its practical application to a couple of non-trivial verification problems.

The rest of this paper is organized as follows. In chapter 2 we recall the main theoretical results for completion based inductive theorem proving and provide an overview of UNICOM, an

implementation of the method. A detailed case study in verifying various sorting algorithms is presented in chapter 3. Then we summarize the experiences gained from the experiments pointing out important technical and conceptual aspects of intelligent proof engineering and discuss open problems.

The appendix contains a complete listing of defining rules and inductive verification properties for various sorting algorithms as well as of the underlying data types for boolean algebra and natural numbers and lists.

2. Completion Based Inductive Theorem Proving

First let us recall some basic definitions and notations and the main theoretical results for completion based inductive theorem proving. Missing details can be found in [Gr 89] (cf. also [JK 86], [Fr 86], [Kü 87], [Ba 87] and [Ba 88]).

We are dealing with first order terms over some set of operator symbols F and some set of variables V. We assume that F contains at least one constant. Thus the set of ground terms T is non-empty. By t/p we denote the subterm of t at position p and by σt the result of applying a substitution σ to t. We write u[s] to indicate that the term u contains s as a subterm and (ambiguously) denote by u[t] the result of replacing a particular occurrence of s in u by t. An equation is a pair of terms, written $s = t$. By $s == t$ we ambiguously denote $s = t$ or $t = s$. A rewrite rule is a directed equation, written $s \to t$. A term rewriting system (TRS) R is a set of rewrite rules. For a set E of equations we denote by $E^{\leftrightarrow}$ the symmetric closure of E. For a binary relation $\to$ on terms the symbols $-=\!\!\to$, , $\xrightarrow{+}$, $\xrightarrow{*}$ and $\leftrightarrow$ stand for the reflexive, transitive, reflexive-transitive and symmetric closure of $\to$, respectively. The relation $\leftarrow$ is the inverse of $\to$. By $\to_R$ we denote the reduction relation generated by R and by $\leftrightarrow_E$ the (one-step) equality relation induced by E. The equational theory of E is defined to be $Th(E) := \{s = t \mid s \xleftrightarrow{*}_E t\}$ and the inductive theory is given by $ITh(E) := \{s = t \mid \sigma s \xleftrightarrow{*}_E \sigma t$ for all ground substitutions $\sigma\}$. Equations from Th(E) and ITh(E) are called equational and inductive theorems of E, respectively.

By CP(R,R') we denote the set of critical pairs obtained from overlapping the rules of R into those of R'. If p is a non-variable position of a term l then $CP(R,p,l \to r)$ stands for the set of critical pairs obtained by overlapping R into $l \to r$ at position p. Accordingly $CP(R,P,l \to r)$ denotes the set of critical pairs obtained by overlapping R into $l \to r$ at all non-variable positions from P. We speak of a critical peak $(\sigma l[\sigma r']_{l' \to r} \leftarrow \sigma l[\sigma l'] \to_{l \to r} \sigma r)$ when taking into account the corresponding superposition term $\sigma l[\sigma l']$ belonging to the critical pair $(\sigma l[\sigma r'], \sigma r)$, too.

A TRS R is terminating if $\xrightarrow{+}_R$ is a well-founded (strict partial) ordering. R is confluent if $_R\!\xleftarrow{*} \circ \xrightarrow{*}_R \subseteq \xrightarrow{*}_R \circ {}_R\!\xleftarrow{*}$ and Church-Rosser if $\xleftrightarrow{*}_R \subseteq \xrightarrow{*}_R \circ {}_R\!\xleftarrow{*}$. It is ground confluent (ground Church-Rosser) if it is confluent (Church-Rosser) on ground terms. Note that the (ground) Church-Rosser property is equivalent to (ground) confluence. R is (ground) convergent if it is terminating and (ground) confluent. A reduction ordering $>$ is a well-founded ordering on terms which is monotonic w.r.t. to replacement ($s > t \Rightarrow u[s] > u[t]$) and substitution ($s > t \Rightarrow \sigma s > \sigma t$). A simplification ordering is a reduction ordering that satisfies the subterm property ($s[t] \geq t$).

By a proof in E we mean a sequence of equational replacements $t_0 \longleftrightarrow_E t_1 \longleftrightarrow_E \ldots \longleftrightarrow_E t_n$. A proof in $E \cup R$ consists of proof steps of the form $t_i \longleftrightarrow_E t_{i+1}$, $t_i \rightarrow_R t_{i+1}$ or $t_i {}_R\!\leftarrow t_{i+1}$. If E and R are unimportant or clear from the context we also denote a proof of the form $t_0 \overset{*}{\longleftrightarrow}_{E\cup R} t_n$ by the sequence $(t_0,\ldots,t_n)$ of its intermediate results. Two proofs of the form $s \overset{*}{\longleftrightarrow}_{E\cup R} t$ and $u \overset{*}{\longleftrightarrow}_{E'\cup R'} v$ are said to be equivalent if $s \equiv u$ and $t \equiv v$ ($\equiv$ means syntactical equality). Proofs of the form $\circ \overset{*}{\rightarrow}_R \circ {}_R\!\overset{*}{\leftarrow} \circ$ are called rewrite proofs. For a proof P we denote by σP the proof obtained from P by instantiating all intermediate results with σ. The notion P[P'] ambiguously indicates that P contains P' as a subproof. For a proof $P = (t_0,\ldots,t_n)$ and a term c we denote by c[P] the proof $(c[t_0],\ldots,c[t_n])$. A proof ordering is a (strict partial) ordering >> on proofs. It is said to be a proof reduction ordering if it is well-founded and monotonic w.r.t. replacement ($P >> P' \Rightarrow c[P] >> c[P']$), substitution ($P >> P' \Rightarrow \sigma P >> \sigma P'$) and embedding ($P >> P' \Rightarrow Q[P] >> Q[P']$). A proof P (in $E \cup R$) of the form $s \overset{*}{\longleftrightarrow}_{E\cup R} t$ is said to be (equivalently) simplifiable into a proof P' (in $E' \cup R'$) if P' is of the form $s \overset{*}{\longleftrightarrow}_{E'\cup R'} t$ with $P >> P'$ (see [Ba 87] for a more detailed introduction of equational proofs).

For a given TRS R a term s is inductively (R-) reducible iff all its ground instances are (R-) reducible. An equation s = t is inductively (R-) reducible iff σs or σt is (R-) reducible for every ground instance $\sigma s = \sigma t$ of s = t with $\sigma s \not\equiv \sigma t$. For a given ground convergent TRS R which is >-ordered by some reduction ordering > we say that a set C of equations (inductive conjectures) is provably inconsistent iff C contains an equation s = t with s > t and s not inductively reducible or an equation s = t with $s \not\equiv t$ and s = t not inductively reducible.

Using these notions inductive validity can be characterized as follows (cf. [Gr 89]):

Theorem

Let > be a reduction ordering, R a >-ordered ground convergent TRS, L a set of inductive theorems, i.e. $L \subseteq ITh(R)$ and C a set of inductive conjectures. Then:

$C \subseteq ITh(R)$ iff

(1) C is not provably inconsistent, and

(2) all ground instances of critical peaks corresponding to $CP(R,C^{\leftrightarrow})$ are equivalently simplifiable with $R \cup C \cup L$ (w.r.t $>^{pos}$).

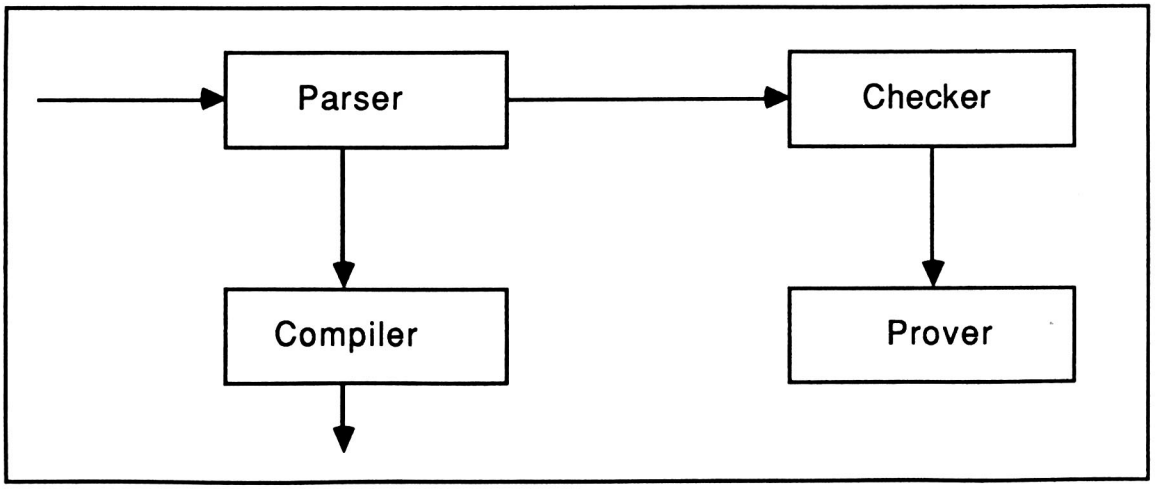
Here $>^{pos}$ is a proof reduction ordering which compares complexities of proofs using $R \cup C \cup L$ by comparing the corresponding multisets of the complexities of all occurring C-steps using the double multiset extension >> >> of the reduction ordering > (cf. [Gr 89]). Condition (1) is decidable and can be easily tested in theories with free constructors. $F_0 \subseteq F$ is a set of free constructors (w.r.t. R) iff every ground term $s \in T(F)$ is R-equivalent to a unique ground constructor term $s' \in T(F_0)$. A sufficient operational criterion for condition (2) is to verify simplifiability of the critical peaks corresponding to $CP(R, C^{\longleftrightarrow})$ instead of ground simplifiability. Note that for constructing such simpler proofs also non-reducing steps (w.r.t >) are possible as long as the resulting proof is smaller than the original one. In particular this enables reduction modulo some set of non-orientable inductive properties, e.g. AC axioms, provided that the reduction ordering used is compatible with the underlying theory (cf. [JK 86b]). Moreover, for

conjectures s = t with s > t it suffices to consider CP(R, s → t) instead of CP(R, {s → t, t → s}). And furthermore, if for s = t ∈ C, ¬(t > s), p is an inductively complete position in S (i.e. for all ground substitutions with σx irreducible for all x ∈ V(S), σs is reducible at position p), then the critical pairs to be considered for s = t can be restricted to CP(R, p, s → t) (cf. [Fr 86]). All other critical pairs, e.g. CP(C,C), are in a sense inessential ones, but can lead to potentially useful new conjectures and may be computed, too. Further refinements, optimizations and generalizations of the basic technique are summarized and discussed in [Gr 89].

Completion based inductive proving roughly spoken proceeds as follows: Given a ground convergent base system R and a set C of inductive conjectures, C is first tested on provable inconsistency. If it is not provably inconsistent (ground) simplifiability of critical pairs obtained by overlapping R into C is tried to be established. For that purpose it may (seem to) be necessary to add new equations (conjectures) which are recursively processed in the same way. This process may stop successfully (all original and deduced conjectures are inductive theorems), detect a contradiction, i.e. provable enconsistency (at least one original conjecture is not inductively valid) or run forever.

UNICOM: A Refined Completion Based Inductive Theorem Prover

Based on the presented theoretical framework we have implemented UNICOM, a system for refined **UN**failing **I**nductive **COM**pletion which is described in detail in [Sc 88] and [Gr 89a]. UNICOM is built on top of TRSPEC, a term rewriting based system for investigating hierarchically structured many-sorted algebraic specifications (cf. [AGGMS 87]. TRSPEC and UNICOM are able to treat hierarchically structured many-sorted specifications of functions (rewrite programs) and inductive conjectures (properties to be proved). Input specifications have to satisfy the following conditions. Constructors have to be declared and are required to be free. The left hand sides of the definition rules for a (n-ary) non-constructor symbol f have to be of the form $f(t_1,...,t_n)$, where all t_i are constructor terms. The system comprises the following tools (see figure).



The **parser** checks the syntax of input specifications, imports used subspecifications and produces

an internal representation.

The **checker** tests the function definitions for completeness and consistency using the syntactical restrictions mentioned above. In particular, termination of the definition rules is established by automatically generating a suitable recursive path ordering with status which is also used for subsequent inductive proofs. Ground convergence is established by investigating critical pairs. A special feature of the implemented completeness test allows to identify minimal complete sets of defining rules for the same function symbol, i.e. alternative but equivalent function definitions.

The **compiler** provides a means for rapid prototyping by translating correct specifications (only the definition part) into executable LISP code.

The central part of the system is the **prover** which tries to prove or disprove the inductive conjectures of the actual specification. The original completion based prover of TRSPEC was already able to handle certain non-orientable conjectures leading to globally finite instead of terminating rewriting systems (cf. [Gö 85], [Gö 87], [Gö 88]). UNICOM now admits arbitrary possibly non-orientable conjectures. The following features are characteristic for UNICOM:

- parallel independent inductive proofs can be performed according to the different possibilities of choosing complete positions in conjectures together with corresponding minimal complete function definitions
- inessential critical pairs may be computed as potentially useful auxiliary conjectures
- simplification (reduction) may be performed modulo the AC-properties of some operators (cf. [GD 88])
- a simple generalization technique (looking for minimal non-variable common subterms) is available
- elimination of subsumed non-orientable conjectures is integrated
- non-equational inductive knowledge about free constructors is used for speeding up the proof or disproof of conjectures
- various user interface parameters allow for switching on/off optional features (e.g. generalization, computation of inessential critical pairs) and enable a fully automatic or more or less strongly user-controlled running mode.

3. A Case Study in Verifying Sorting Algorithms with UNICOM

We shall deal with the specification and verification of a couple of sorting algorithms for lists of natural numbers equipped with their usual ordering relation $\leq$. The considered algorithms comprise sorting by insertion, minimum-sort, bubble-sort, quick-sort and merge-sort. For an equational specification of these algorithms and their (partial) correctness properties we need some basic data types, namely for boolean algebra, for natural numbers with ordering relation and equality predicate and for lists of natural numbers. Moreover, since sorting algorithms inherently require some kind of conditional reasoning, we have to provide means for specifying conditional operations and

properties. Within this case study we restrict ourselves to purely equational reasoning by encoding conditional constructs into unconditional ones using ternary if-then-else operations. Thus we avoid the difficulties of properly conditional term rewriting and completion techniques. Of course, the problems concerning conditional reasoning do not simply disappear but emerge in another form as we shall see. If not indicated accordingly equations specifying defined functions and inductive conjectures will always be interpreted as directed from left to right, i.e. as rewrite rules. Furthermore we tacitly assume (sufficient) completeness and consistency of the considered specifications. In particular, we assume the existence of appropiate reduction orderings for the corresponding rewrite rule systems. For the sake of readability we shall make free use of a mixed prefix and infix notation with usual priority rules for interpreting missing brackets.

3.1 Basic Data Types

Let us start with an equational specification of boolean algebra. The functions $\wedge$ (and), $\vee$ (or) and $\neg$ (not) are completely and consistently specified over the constant constructors t (true) and f (false) by the following set of equations (where b is a boolean variable):

E_{bool}: $\neg f = t$ $\quad f \vee b = b$ $\quad f \wedge b = f$

$\neg t = f$ $\quad t \vee b = t$ $\quad t \wedge b = b$

The usual properties of boolean algebra like associativity and commutativity (AC) of $\wedge$ and $\vee$ are easily shown to be inductively valid w.r.t. E_{bool} (cf. appendix).

Next we need a specification of natural numbers built over the constructors o (zero) and s (successor) with the functions $=_{nat}$ (equality), $\leq$ (less-or-equal) and + (addition).

E_{nat}: $o =_{nat} o = t$ $\quad o \leq n = t$ $\quad o + n = n$

$o =_{nat} s(n) = f$ $\quad s(m) \leq o = f$ $\quad s(m) + n = s(m+n)$

$s(m) =_{nat} o = f$ $\quad s(m) \leq s(n) = m \leq n$

$s(m) =_{nat} s(n) = m =_{nat} n$

Again the non-primitive functions $=_{nat}$, $\leq$ and + are completely and consistently specified by E_{nat}. For notational convenience we shall sometimes abbreviate $\neg\, m \leq n$ by $n < m$. Note that one could also explicitely introduce a function $< : nat \times nat \rightarrow bool$ by an appropiate recursive definition. Then properties relating $\leq$, $<$ and $=_{nat}$ like $\neg(m \leq n) = n < m$ or $m \leq n = m < n \vee m =_{nat} n$ could indeed be easily verified by UNICOM. Basic inductive properties of the above specification include the AC-axioms for +, reflexivity and transitivity of $=_{nat}$ and $\leq$, symmetry of $=_{nat}$ and anti-symmetry of $\leq$ as well as totality of $\leq$. All these properties (and even more) are easily verified with UNICOM. As an example let us consider transitivity of $\leq$ expressed by

(1) $n_1 \leq n_2 \wedge n_2 \leq n_3 \wedge n_1 \leq n_3 = n_1 \leq n_2 \wedge n_2 \leq n_3$

Note that we have omitted the brackets for the argument of $\wedge$, because as soon as the AC-properties for some operator have been proved, we shall always work modulo these properties, i.e. equality and rewriting steps within the simplification process are performed modulo AC.

Now, the critical pairs obtained by superposing the defining rules for $\leq$ into (1) at the subterm $n_1 \leq n_2$ occurring at an inductively complete position are

$$t \wedge n_2 \leq n_3 \wedge o \leq n_3 = t \wedge n_2 \leq n_3$$
$$f \wedge o \leq n_3 \wedge s(n_1) \leq n_3 = f \wedge o \leq n_3 \text{ and}$$
$$n_1 \leq n_2 \wedge s(n_2) \leq n_3 \wedge s(n_1) \leq n_3 = n_1 \leq n_2 \wedge s(n_2) \leq n_3 .$$

The first two equations can be simplified into trivial ones. For the last one which cannot be simplified any more we choose the inductively complete position at which the subterm $s(n_2) \leq n_3$ occurs for critical pair construction and obtain

$$n_1 \leq n_2 \wedge f \wedge s(n_1) \leq o = n_1 \leq n_2 \wedge s(n_2) \leq o$$
$$n_1 \leq n_2 \wedge n_2 \leq n_3 \wedge s(n_1) \leq s(n_3) = n_1 \leq n_2 \wedge s(n_2) \leq s(n_3)$$

The first critical pair is again eliminated by simplification and the latter is reduced to

$$n_1 \leq n_2 \wedge n_2 \leq n_3 \wedge n_1 \leq n_3 = n_1 \leq n_2 \wedge n_2 \leq n_3$$

and finally, using the original conjecture (1), to the trivial equation

$$n_1 \leq n_2 \wedge n_2 \leq n_3 = n_1 \leq n_2 \wedge n_2 \leq n_3$$

which concludes the proof.

Lists of natural numbers are built up by the empty list constructor e: $\rightarrow$ list and the binary cons-operation c: nat $\times$ list $\rightarrow$ list. Concatenation of lists is specified by the append-operation app: list $\times$ list $\rightarrow$ list with

$$app(e,l) = l$$
$$app(c(n,l_1),l_2) = c(n,app(l_1,l_2))$$

Easily provable inductive properties of app are for instance

$$app(l,e) = l$$
$$app(app(l_1,l_2),l_3) = app(l_1,app(l_2,l_3)).$$

For realizing conditional reasoning we further need ternary if-then-else operations if-s: bool $\times$ s $\times$ s $\rightarrow$ s for alternatives of sort s (here: for s = bool,nat,list). These operations can

be defined schematically by

$$\text{if-s}(f,x,y) = y$$
$$\text{if-s}(t,x,y) = x.$$

Basic and easily provable properties of these operations include among others the following

$$\text{if-s}(b,x,x) = x$$
$$\text{if-s}(b,\ \text{if-s}(b,x,y),z) = \text{if-s}(b,x,z)$$
$$\text{if-s}(b,x,\text{if-s}(b,y,z)) = \text{if-s}(b,x,z)$$
$$\text{if-s}(b,\text{if-s}(\neg b,x,y),z) = \text{if-s}(b,y,z)$$
$$\text{if-s}(b,x,\text{if-s}(\neg b,y,z)) = \text{if-s}(b,x,y)$$

and

$$f(\ldots,\text{if-s}'(b,u,v),\ldots) = \text{if-s}(b,f(\ldots,u,\ldots),f(\ldots,v,\ldots))$$
$$g(\ldots,\text{if-s}'(b,u,v),\ldots,\text{if-s}''(b,x,y),\ldots) = \text{if-s}(b,g(\ldots,u,\ldots,x,\ldots),g(\ldots,v,\ldots,y,\ldots))$$

for every f and g with corresponding arity. For boolean sort we have in particular

$$\text{if-bool}(b_1,b_2,b_3) = b_1 \wedge b_2 \vee \neg b_1 \wedge b_3\ .$$

3.2 Specifying Partial Correctness of Sorting Algorithms

Assume that we are given a sufficiently complete and consistent equational specification for a function sort: list → list for sorting lists of natural numbers (in ascending order). Then we may ask for a formal specification of (partial) correctness of sort. Intuitively it is clear, that correctness is characterized by the following two properties which must hold for every list l of natural numbers:

(I) sort(l) is ordered (ascendingly).
(II) l and sort(l) have the same multiset of elements.

These two properties may be formalized in quite different ways as we shall point out and discuss now. For property (I) it seems natural to introduce an ordered-predicate which is modelled by a boolean valued function ord: list → bool and to require

$$\text{ord}(\text{sort}(l)) = \text{true}.$$

A first specification of being an ordered list then consists in defining recursively

(Ia) $\text{ord}(e) = t$
$\text{ord}(c(n,e)) = t$

$$ord(c(m,c(n,l))) = m \leq n \wedge ord(c(n,l))$$

But other specifications may also be conceivable and - concerning proof technical aspects - even more advantageous. For instance we may in some sense redundantly require that the first element of a given list is not only less than or equal to the next element (provided there exists one) but less than or equal to all remaining list elements. This leads to the following specification with an auxiliary operation $\leq_{nl}$: nat × list → bool.

(Ib) $ord(e) = t$
$ord(c(n,l)) = n \leq_{nl} l \wedge ord(l)$
$m \leq_{nl} e = t$
$m \leq_{nl} c(n,l) = m \leq n \wedge m \leq_{nl} l$

Still another less explicit and not constructor based definition using the auxiliary operations $\leq_{nl}$ (as above) and $\leq_{ll}$ would be

(Ic) $ord(app(l_1,l_2)) = ord(l_1) \wedge ord(l_2) \wedge l_1 \leq_{ll} l_2$
$e \leq_{ll} l = t$
$c(n,l_1) \leq_{ll} l_2 = n \leq_{nl} l_2 \wedge l_1 \leq_{ll} l_2$

It is rather straightforward to prove that the definitions (Ia), (Ib) and (Ic) are indeed equivalent. We shall make use of (Ia) and (Ib) and exploit (Ic) for an appropriate decomposition of the verification problem for merge-sort.

The second (partial) correctness property (II) of sorting algorithms may be formalized in quite different ways. A first possibility is to introduce the property that a list is a permutation of another list via the operation perm: list × list → bool which uses auxiliary functions del: nat × list → list and $\in$: nat × list → bool for deleting one occurrence of an element in a given list and for membership test, respectively:

(IIa) $perm(e,e) = t$
$perm(e,c(n,l)) = f$
$perm(c(m,l_1),l_2) = m \in l_2 \wedge perm(l_1,del(m,l_2))$
$m \in e = f$
$m \in c(n,l) = m =_{nat} n \vee m \in l$
$del(m,e) = e$
$del(m,c(n,l)) = \text{if-list}(m =_{nat} n,l,c(n,del(m,l)))$

The correctness property (II) for sorting algorithms may then be formalized by

$perm(l,sort(l)) = true.$

The above definition of perm is in a sense a strongly algorithmic one since the computation of $perm(l_1,l_2)$ for concrete lists l_1,l_2 of natural numbers proceeds by successively considering the first element m of l_1, testing for membership of m in l_2 and deleting (the first occurrence of) m in l_1 and l_2. Moreover the structure of the definition does not reflect the symmetrical aspect of permutative equivalence, i.e. the property that two lists are the same up to a permutation of their arguments. This symmetry stated by

$$perm(l_1,l_2) = perm(l_2,l_1)$$

does indeed hold for the above definition, but a formal proof of it seems to be difficult.

As an alternative where symmetry becomes obvious consider the following definition using a function oc: nat × list → nat which counts the number of occurrences of some natural number in a given list.

(IIb) $oc(m,e) = o$
$oc(m,c(n,l)) = \text{if-nat}(m =_{nat} n, s(o),o) + oc(m,l)$

Then permutative equivalence of two lists l_1, l_2 is formalized by

$$oc(n,l_1) = oc(n,l_2)$$

To ensure that this definition is indeed equivalent to the former version using perm we have to verify that

(L) $perm(l_1,l_2) = t \Leftrightarrow \forall n: oc(n,l_1) = oc(n,l_2)$

holds in the initial model I(E) of E with E consisting of the defining equations for all function symbols involved. Note that (L) does not have equational form. Hence the method of completion based inductive theorem proving is not applicable here. Instead we shall give a classical proof for (L) based on induction over the list structure of l_1. Denoting the conjecture by

$$P(l_1) := \forall\, l_2 : [\, perm(l_1,l_2) = t \Leftrightarrow \forall n: oc(n,l_1) = oc(n,l_2)\,]$$

we thus have to prove:

(IB) P(e) and
(IS) $\forall l_1: P(l_1) \Rightarrow P(c(m,l_1))$

For the induction basis (IB) we distinguish two cases, namely $l_2 = e$ and $l_2 = c(m',l_2')$: For $l_2 = e$ we obtain

$$perm(e,e) = t \Leftrightarrow \forall n: oc(n,e) = oc(n,e)$$

which simplifies to the trival equivalence

$$t = t \;\Leftrightarrow\; 0 = 0.$$

The second case yields

$$perm(e,c(m',l_2')) = t \;\Leftrightarrow\; \forall n\colon oc(n,e) = oc(n,c(m',l_2'))$$

which is reduced to

$$f = t \;\Leftrightarrow\; \forall n\colon o = \text{if-nat}(n =_{nat} m', s(o),o) + oc(n,l_2')$$

Choosing $n = m'$ on the right-hand side $o = \text{if-nat}(m' =_{nat} m', S(o),o) + oc(m',l_2')$ simplifies to $o = s(oc(m',l_2'))$ which is unsatisfiable. Thus, due to unsatisfiability of $f = t$, we are done. For the induction step (IS) we have to prove

$$(*) \quad perm(c(m,l_1),l_2) = t \;\Leftrightarrow\; \forall n\colon oc(n,c(m,l_1)) = oc(n,l_2)$$

for some arbitrary but fixed l_1 under the induction hypothesis

$$\forall\, l_2\ [\ perm(l_1,l_2) = t \;\Leftrightarrow\; \forall n\colon oc(n,l_1) = oc(n,l_2)\ ]\ .$$

Using the definitions of perm and oc $(*)$ simplifies to

$$m \in l_2 \;\wedge\; perm(l_1,del(m,l_2)) = t \;\Leftrightarrow\; \forall n\colon \text{if-nat}(n =_{nat} m, S(o),o) + oc(n,l_1) = oc(n,l_2).$$

Splitting the left-hand side conjunction and applying the induction hypothesis yields

$$m \in l_2 = t \text{ and } \forall n\colon oc(n,l_1) = oc(n,del(m,l_2)) \;\Leftrightarrow\; \forall n\colon \text{if-nat}(n =_{nat} m,S(o),o) + oc(n,l_1) = oc(n,l_2).$$

For the "$\Rightarrow$"-direction we now use the auxiliary lemma

$$(L_1) \quad m \in l = t \;\Rightarrow\; oc(n,l) = \text{if-nat}(n =_{nat} m,s(o),o) + oc(n,del(m,l))$$

for $l = l_2$ such that

$$\forall n\colon \text{if-nat}(n =_{nat} m,s(o),o) + oc(n,l_1) = oc(n,l_2)$$

is transformed into

$$\forall n\colon \text{if-nat}(n =_{nat} m,s(o),o) + oc(n,l_1) = \text{if-nat}(n =_{nat} m,s(o),o) + oc(n,del(m,l_2))$$

which is trivially satisfied using the second assumption

$\forall n$: $oc(n,l_1) = oc(n,del(m,l_2))$.

For the "$\Leftarrow$"-direction we first deduce $m \in l_2 = t$ by substituting m for n and using lemma

(L_2) $s(oc(m,l_1)) = oc(m,l_2) \Rightarrow m \in l_2 = t$.

Applying (L_1) and the cancellation law for +

(L_3) $x + y = x + z \Rightarrow y = z$

to the right hand side then produces

$\forall n$: $oc(n,l_1) = oc(n,del(m,l_2))$

which remained to be shown. ■

Of course, the inductive validity of the auxiliary lemmas (L_1) - (L_3) involved in the above proof also has to be established. This is possible by again using structural induction.

From (L) we can easily deduce now that perm is reflexive, symmetric and transitive.
Comparing the definitions (IIa) and (IIb) concerning proof technical aspects the first one seems to be better for such cases where we know something about the first elements of the two lists to be compared. This comes true for instance for sorting by insertion and min-sort. Definition (IIb) however is better suited for verifying quick-sort and merge-sort, where the first element of a sorted list is not directly visible from the definition of sorting.

Finally, let us mention that the property of permutative equivalence could also be specified by explicitely computing the multiset of list elements and defining inclusion and equality for multisets (of elements).

3.3 Completion Based Verification of Sorting Algorithms

In the following we shall present equational specifications of some sorting algorithms as well of their corresponding correctness properties. In all cases the sets of defining equations oriented from left to right constitute ground convergent rewrite systems. The termination property which is tacitly assumed can be established by choosing appropiate semantical recursive path orderings (s.r.p.o.'s, cf. [KL 80]). Roughly spoken, the semantical component of the s.r.p.o. is necessary for taking into account the fact that the length of list arguments in recursive calls of some sorting functions is strictly decreasing (e.g. for quick-sort).

We summarize essential steps of the verification process performed by our completion based

inductive theorem prover UNICOM and guided by the human user. A typical phenomenon arising in all the examples is that a whole hierarchy of auxiliary lemmas is necessary to succeed in the original proof. The central parts of the lemma hierarchies for the examples are depicted graphically leaving out basic lemmas concerning boolean reasoning and the ordering relation $\leq$ (cf. the appendix for a complete listing). Interesting steps of problem reduction by applying auxiliary lemmas are motivated and discussed. Let us start with

3.3.1 Sorting by Insertion

The main algorithm isort: list $\rightarrow$ list needs an auxiliary operation ins: nat $\times$ list $\rightarrow$ list which inserts a given natural number into a list at the "right" place w.r.t. the ordering $\leq$. For modelling the correctness properties we choose the variants (Ia), (IIa) (cf. section 3.2). The resulting specification which implicitely uses the given specifications for boolean algebra and natural numbers as subspecifications looks as follows:

(1) isort(e) = e
(2) isort(c(m,l)) = ins(m,isort(l))
(3) ins(m,e) = c(m,e)
(4) ins(m,c(n,l)) = if-list(m $\leq$ n,c(m,c(n,l)),c(n,ins(m,l)))
(5) ord(e) = t
(6) ord(c(m,e)) = t
(7) ord(c(m,c(n,l))) = m $\leq$ n $\wedge$ ord(c(n,l))
(8) perm(e,e) = t
(9) perm(e,c(m,l)) = f
(10) perm(c(m,l_1),l_2) = m $\in$ l_2 $\wedge$ perm(l_1,del(m,l_2))
(11) m $\in$ e = f
(12) m $\in$ c(n,l) = m $=_{nat}$ n $\vee$ m $\in$ l
(13) del(m,e) = e
(14) del(m,c(n,l)) = if-list(m $=_{nat}$ n, l, c(n,del(m,l)))

The corretness properties for isort to be proved are

(L_1) ord(isort(l)) = t, and
(L_2) perm(l,isort(l) = t.

The essential part of the lemma hierarchy successfully handled by UNICOM is depicted in figure 1. The notation $L_1 \rightarrow L_2$ indicates that L_2 is used for the proof of L_1.

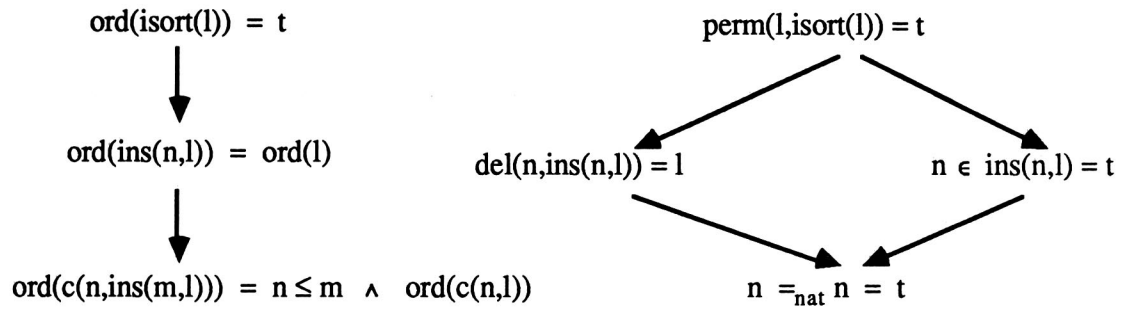


Figure 1: Lemma hierarchy for isort

For illustration purposes let us pick out the left part of the lemma hierarchy. For proving (L_1) we have to consider the two critical pairs

CP(1,L_1) ord(e) = t

which easily turns out to be convergent, and

CP(2,L_1) ord(ins(n,isort(l))) = t.

Using the auxiliary lemma

(L_3) ord(ins(n,l)) = ord(l)

the left hand side ord(ins(n,isort(l))) can be simplified to ord(isort(l)) and by applying (L_1) to t thus yielding inductive validity of (L_1). One possibility to generate the auxiliary lemma (L_3) is to use the heuristic of cross-fertilization and subsequent generalization (cf. [BM 79], [Gr 85]). Roughly spoken this heuristic proceeds as follows in the example. For succeeding in the "induction step" we have to find a proof for

ord(ins(n,isort(l))) = t

under the "induction hypothesis"

ord(isort(l)) = t.

Making use of the "induction hypothesis" by applying it in reverse direction for substituting the right hand side of the "induction step" we obtain

ord(ins(n,isort(l))) = ord(isort(l))

which would suffice for a proof. Now this auxiliary equation again cannot be proved directly but is amenable to a non-trivial generlization step, namely by replacing the common subterm isort(l) on the left and right hand side by a new variable. This leads exactly to (L_3). Note that the original equation

$$ord(ins(n,isort(l))) = t$$

could not be generalized in such a non-trivial way because it does not have common subterms on both sides. Of course, such generalization steps do not preserve inductive validity in general. But when used in combination with the cross-fertilzation technique they often yield conjecures which are indeed inductively valid and easier to prove than the original conjecture. For proving (L_3) in the example we additionally need the key lemma

$$(L_4) \qquad ord(c(n,ins(m,l))) = n \leq m \wedge ord(c(n,l))$$

the proof of which is technically more complicated. Here the interesting case is given by

$$CP(2,L_4) \quad ord(c(n,if\text{-}list(m \leq p,c(m,c(p,l)),c(p,ins(m,l))))) = n \leq m \wedge ord(c(n,c(p,l))).$$

By applying defining rules for ord, boolean and schematic if-rules the left hand side simplifies to

$$if\text{-}bool(m \leq p, n \leq m \wedge m \leq p \wedge ord(c(p,l)), n \leq p \wedge ord(c(p,ins(m,l)))).$$

Using (L_4) again and eliminating if-bool by the basic lemma $if\text{-}bool(b,b_1,b_2) = b \wedge b_1 \vee \neg b \wedge b_2$ produces

$$m \leq p \wedge n \leq m \wedge m \leq p \wedge ord(c(p,l)) \quad \vee \quad p < m \wedge n \leq p \wedge p \leq m \wedge ord(c(p,l))$$

which further simplifies to

$$m \leq p \wedge n \leq m \wedge ord(c(p,l)) \quad \vee \quad p < m \wedge n \leq p \wedge p \leq m \wedge ord(c(p,l)):$$

by means of idempotency of $\wedge$. Finally we need some inductive knowledge about $\leq$, namely the lemmas

$$p < m \wedge p \leq m = p < m$$

and

$$n \leq p \wedge p < m \vee n \leq m \wedge m \leq p = n \leq m \wedge n \leq p.$$

For being applicable in the above context we need these lemmas (as well as idempotency of $\wedge$ above) in generalized extended form, namely as

$$b \wedge p < m \wedge p \leq m \;=\; b \wedge p < m$$

and

$$b \wedge n \leq p \wedge p < m \;\vee\; b \wedge n \leq m \wedge m \leq p \;=\; b \wedge n \leq m \wedge n \leq p.$$

We shell come back to this practically important technical detail later on. Using these two lemmas the left hand side reduces further to

$$n \leq m \wedge n \leq p \wedge \text{ord}(c(p,l))$$

which coincides with the simplified right hand side of $CP(2,L_4)$ as desired.

The proof of correctness property

$$(L_2) \qquad \text{perm}(l,\text{isort}(l)) = t$$

is rather straightforword provided that the auxiliary lemmas mentioned in figure 1 are available. The only difficulty arises in verifying

$$(*) \qquad \text{del}(n,\text{ins}(n,l)) = l$$

which requires more knowldge over the if-then-else constructs as follows. The interesting case here is given by overlapping the defining rule (4) into (*) which yields the critical pair

$$\text{del}(n,\text{if-list}(n \leq p, c(n,c(p,l)), c(p,\text{ins}(n,l)))) = c(p,l).$$

Moving if-list to the top position the left hand side simplifies to

$$\text{if-list}(n \leq p, \text{del}(n,c(n,c(p,l))), \text{del}(n,c(p,\text{ins}(n,l)))).$$

Application of definition rule (14) and further simplification using reflexivity of $=_{nat}$ and the "induction hypothesis" (*) finally produces

$$\text{if-list}(n \leq p, c(p,l), \text{if-list}(n =_{nat} p, \text{ins}(n,l), c(p,l))).$$

For reducing this term to the right hand side c(p,l) we have to exploit the intuitive argument that the inner alternative ins(n,l) for the case $n =_{nat} p = t$ is never relevant. Formally this is expressed by

$$(**) \qquad \text{if-list}(n \leq p, l_1, \text{if-list}(n =_{nat} p, l_2, l_3)) = \text{if-list}(n \leq p, l_1, l_3)$$

which suffices for further reducing the left hand side to

$$\text{if-list}(n \leq p, c(p,l), c(p,l))$$

and finally to c(p,l) as desired. Note that (**) is easily provable by UNICOM, for instance by considering the critical pairs at the inductively complete position of the left hand side subterm $n =_{nat} p$. In fact (**) is a special case of a more general inductive property of if-s, namely

if-s(b,x,if-s(b',y,z)) = if-s(b,x,z)

for all b, b' such that b' implies b. Unfortunately this formulation is no langer purely equational due to the constraints on b, b' and thus cannot be handled by the current version of UNICOM. But nevertheless specific instances of such properties like (**) above can often be handed successfully within the purely equational framework.

3.3.2 Minimum-Sort

Sorting by recursively computing and extracting minimal elements is specified by the main function minsort: list → list. The computation of minimal elements is modelled by the binary function min: nat x list → list which delivers the minimal element among the first argument and all elements of the second list argument. Note that choosing this binary function min instead of the unary min1: list → list avoids the problem of partial definitions (here: min1(e) would be undefined !). The correctness properties are formalized using the variants (Ib), (IIa). Thus the specification looks as follows:

(1) minsort(e) = e
(2) minsort(c(n,l)) = c(min(n,l),minsort(del(min(n,l),c(n,l))))
(3) min(n,e) = n
(4) min(n,c(m,l)) = if-nat(n ≤ m, min(n,l),min(m,l))
(5) ord(e) = t
(6) ord(c(n,l)) = $n \leq_{nl} l \wedge$ ord(l)
(7) $n \leq_{nl} e = t$
(8) $n \leq_{nl} c(m,l) = n \leq m \wedge n \leq_{nl} l$
(9) perm(e,e) = t
(10) perm(e,c(n,l)) = f
(11) perm(c(n,l_1), l_2) = $n \in l_2 \wedge$ perm(l_1,del(n,l_2))
(12) $m \in e = f$
(13) $m \in c(n,l) = m =_{nat} n \vee m \in l$
(14) del(m,e) = e
(15) del(m,c(n,l)) = if-list($m =_{nat} n$,l,c(n,del(m,l)))

The correctness properties for minsort to be established are

(L_1) ord(minsort(l)) = t and
(L_2) perm(minsort(l),l) = t.

Figure 2 shows the essential part of the lemma hierarchy for proving these two properties

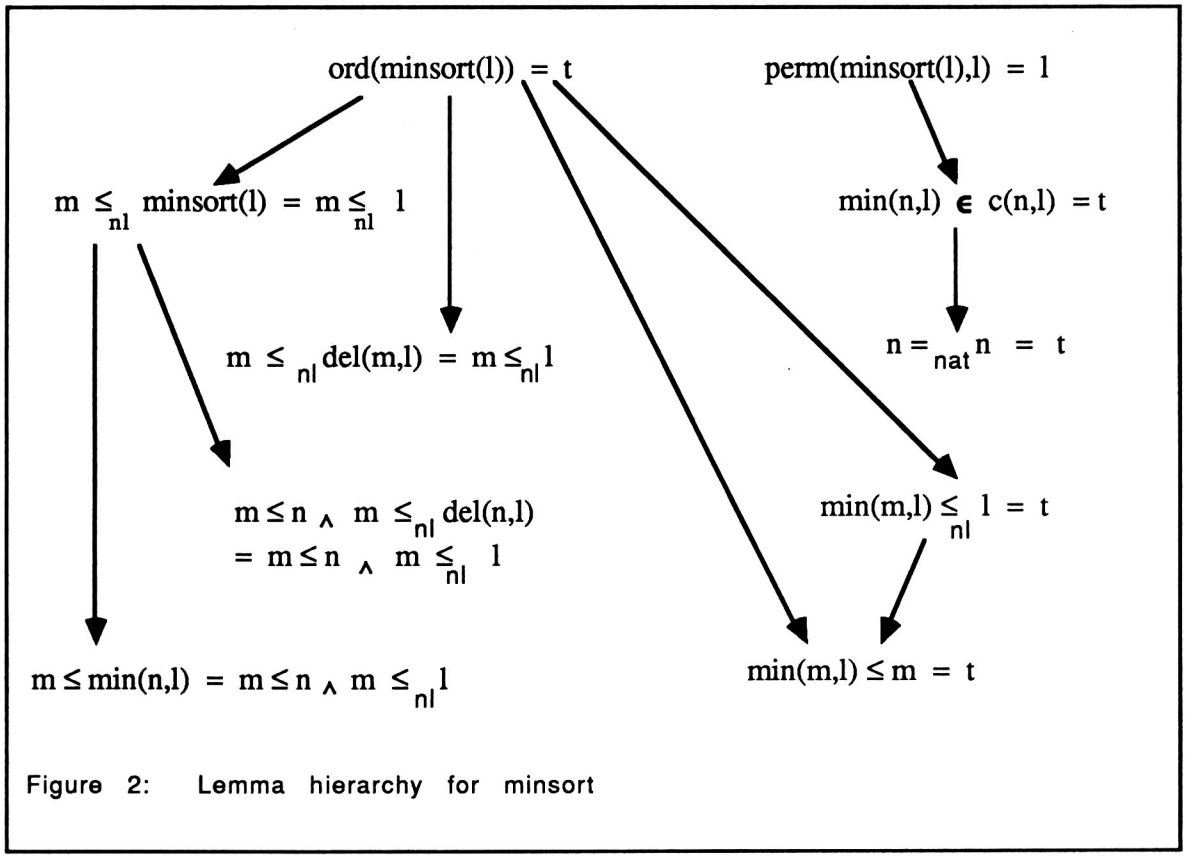


Figure 2: Lemma hierarchy for minsort

Let us comment some aspects of this lemma hierarchy. First of all, choosing

(L_2) perm(minsort(l),l) = t

instead of the symmetric version

(L_2') perm(l,minsort(l)) = t

is better suited from a proof technical point of view. In fact the only non-trivial auxiliary lemma needed is

min(n,l) ∈ c(n,l) = t

whereas a proof of the second version (L_2') would be much more complicated. For verifying property

(L_1) ord(minsort(l)) = t

a couple of non-trivial auxiliary lemmas has to be provided. Here the interesting proof step stems from the critical pair CP(2, L_1)

ord(c(min(m,l),minsort(del(min(m,l),c(m,l))))) = t.

The left-hand side successively simplifies to

min(m,l) $\leq_{nl}$ minsort(del(min(m,l),c(m,l))) $\wedge$ ord(minsort(del(min(m,l),c(m,l))))

using definition rule (6) for ord. Applying (L_1) and eliminating t yields

min(m,l) $\leq_{nl}$ minsort(del(min(m,l),c(m,l)))

which reduces to

min(m,l) $\leq_{nl}$ del(min(m,l),c(m,l))

by using the first structural simplification property (L_3) m $\leq_{nl}$ minsort(l) = m $\leq_{nl}$ l. The next auxiliary lemma (L_4) m $\leq_{nl}$ del(m,l) = m $\leq_{nl}$ l leads to

min(m,l) $\leq_{nl}$ c(m,l)

which by definition of $\leq_{nl}$ is decomposed into

min(m,l) $\leq$ m $\wedge$ min(m,l) $\leq_{nl}$ l.

Using the two lemmas (L_5) min(m,l) $\leq$ m = t and (L_6) min(m,l) $\leq_{nl}$ l = t this term immediately simplifies to t which was to be shown. From a conceptual point of view the lemmas (L_5) and (L_6) may be considered to be logical key properties describing the connection between minimum construction and ordering relation. On the other hand (L_3) and (L_4) are in a sense structural simplification properties because they eliminate certain defined function symbols. If processed in a bottom-up fashion all properties of the lemma hierarchy can be mechanically proved by UNICOM with two exceptions, namely (L_5) and (L_6) which require a more subtle analysis and currently have to be supplied by the human user. The difficulties arising in verifying (L_5) and (L_6) are instances of a more general phenomenon that will be discussed in detail in chapter 4.

3.3.3 Bubble-Sort

Using an imperative programming style a familiar specification of bubble-sort for an array A of length n could look as follows:

Procedure bubble-sort(A)
for j = n-1 step -1 until 1 do
  for i = 1 step 1 until j do
    if A[i+1] < A[i] then interchange A[i] and A[i+1].

It is obvious that after every iteration of the outer loop the element A[j+1] (and all elements with greater index) remain invariant. In order to transform this specification into a purely equational recursive one working on lists we decide to move these elements to the beginning of the list. To obtain ascendingly ordered lists we further specify the bubbling process in such a way that minimal instead of maximal elements are shifted towards the end of the list. As auxiliary functions for the main operation bsort: list → list we need the functions bubble: list → list, last: list → nat and all-but-last: list → list. The correctness properties for bsort will be formalized using variants (Ib) and (IIb). Thus we obtain

(1) bsort(e) = e
(2) bsort(c(n,l)) = c(last(bubble(c(n,l))), bsort(all-but-last(bubble(c(n,l)))))
(3) bubble(e) = e
(4) bubble(c(n,e)) = c(n,e)
(5) bubble(c(m,c(n,l))) = if-list(m ≤ n, c(n,bubble(c(m,l))), c(m,bubble(c(n,l))))
(6) last(e) = o
(7) last(c(n,e)) = n
(8) last(c(m,c(n,l))) = last(c(n,l))
(9) all-but-last(e) = e
(10) all-but-last(c(n,e)) = e
(11) all-but-last(c(m,c(n,l))) = c(m,all-but-last(c(m,l)))
(12) ord(e) = t
(13) ord(c(m,l)) = $m \leq_{nl} l \wedge$ ord(l)
(14) $m \leq_{nl} e = t$
(15) $m \leq_{nl} c(n,l) = m \leq n \wedge m \leq_{nl} l$
(16) oc(m,e) = o
(17) oc(m,c(n,l)) = oc(m,l) + if-nat($m =_{nat} n$,s(o),o)

Note that the function last has been made total by defining last(e) = o. The main correctness properties for bsort to be established are

(L_1) ord(bsort(l)) = t
(L_2) oc(n,bsort(l)) = oc(n,l).

An appropriate lemma hierarchy for proving these two properties with UNICOM is depicted in figures 3a and 3b.

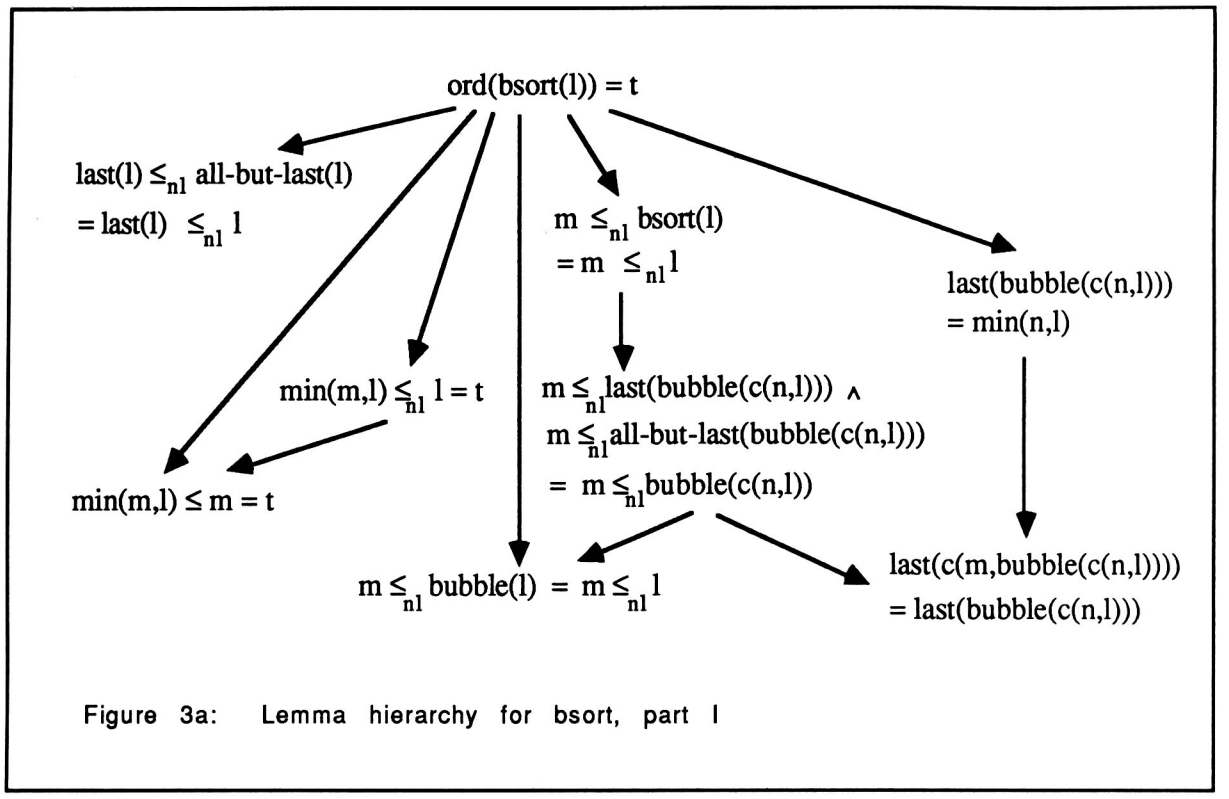


Figure 3a: Lemma hierarchy for bsort, part I

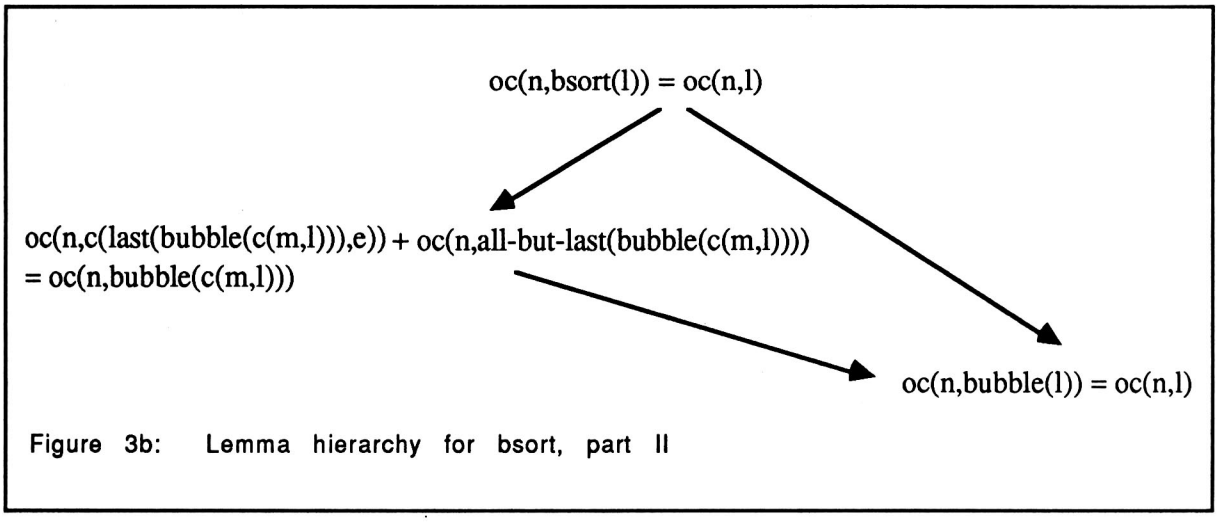


Figure 3b: Lemma hierarchy for bsort, part II

For illustration let us show how the proof of (L_1) goes through making use of the mentioned auxiliary lemmas. The only non-trivial case to be considered stems from the critical pair corresponding to CP(2,L_1), namely

ord(c(last(bubble(c(n,l))), bsort(all-but-last(bubble(c(n,l)))))) = t

Applying definition rule (13) and (L_1) the left-hand side reduces to

last(bubble(c(n,l))) $\leq_{nl}$ bsort(all-but-last(bubble(c(n,l)))).

Now we can use (L_2) m $\leq_{nl}$ bsort(l) = m $\leq_{nl}$ l yielding

last(bubble(c(n,l))) $\leq_{nl}$ all-but-last(bubble(c(n,l))).

Further simplification with (L_3) last(l) $\leq_{nl}$ all-but-last(l) = last(l) $\leq_{nl}$ l leads to

last(bubble(c(n,l))) $\leq_{nl}$ bubble(c(n,l))

and by applying (L_4) m $\leq_{nl}$ bubble(l) = m $\leq_{nl}$ l to

last(bubble(c(n,l))) $\leq_{nl}$ c(n,l).

Exploiting (L_5) last(bubble(c(n,l))) = min(n,l) we get

min(n,l) $\leq_{nl}$ c(n,l)

which simplifies to

min(n,l) $\leq$ n $\wedge$ min(n,l) $\leq_{nl}$ l

by definition of $\leq_{nl}$. Finally, by applying (L_6) min(n,l) $\leq$ n = t, (L_7) min(n,l) $\leq_{nl}$ l = t and boolean simplification we obtain t which coincides with the right-hand of the critical pair as desired. Again all lemmas presented are mechanically proved by UNICOM except (L_6) and (L_7).

3.3.4 Quick-Sort

We decide to take the first element of a non-empty list to be sorted as the splitting element for the recursive case. For specifying the main algorithm qsort: list → list we need two auxiliary functions lowers: nat x list → list and greaters: nat x list → list . Lowers(n,l) computes all elements of l which are less than or equal to n. And greaters(n,l) produces those elements of l that are greater than n. For the correctness properties of qsort we choose the variants (Ib) and (IIb). For proof technical reasons we augment the specification variant (Ib) for the property of being ordered by new relations $\leq_{nl}$ and $\leq_{ln}$. The latter relation compares whole lists whereas the first one is the symmetric version of $\leq_{nl}$. The resulting specification looks as follows.

(1) qsort(e) = e
(2) qsort(c(n,l)) = app(qsort(lowers(n,l)),c(n,qsort(greaters(n,l))))
(3) lowers(n,e) = e
(4) lowers(n,c(m,l)) = if-list(m≤n,c(m,lowers(n,l)),lowers(n,l))
(5) greaters(n,e) = e
(6) greaters(n,c(m,l)) = if-list(m≤n,greaters(n,l),c(m,greaters(n,l)))
(7) ord(e) = t
(8) ord(c(m,l)) = m$\leq_{nl}$l $\wedge$ ord(l)

(9) $m \leq_{nl} e = t$

(10) $m \leq_{nl} c(n,l) = m \leq n \wedge m \leq_{nl} l$

(11) $e \leq_{ln} m = t$

(12) $c(n,l) \leq_{nl} m = n \leq m \wedge l \leq_{ln} m$

(13) $e \leq_{ll} l = t$

(14) $c(m,l_1) \leq_{ll} l_2 = l_1 \leq_{ll} l_2$

(15) $oc(m,e) = o$

(16) $oc(m,c(n,l)) = oc(m,l) + \text{if-nat}(m =_{nat} n,s(o),o)$.

The correctness properties for qsort to be verified are

(L_1) $ord(qsort(l)) = t$ and

(M_1) $oc(n,qsort(l)) = oc(n,l)$.

In this example the mechanical proofs of (L_1) with UNICOM require a lot of auxiliary lemmas from which we separate a basic part. These basic lemmas that concern the interplay between the functions ord, app, $\leq_{nl}$, $\leq_{ln}$ and $\leq_{ll}$ are the following:

(N_1) $ord(app(l_1,l_2)) = ord(l_1) \wedge ord(l_2) \wedge l_1 \leq_{ll} l_2$

(N_2) $l_1 \leq_{ll} c(n,l_2) = l_1 \leq_{ln} n \wedge l_1 \leq_{ll} l_2$

(N_3) $l_1 \leq_{ll} app(l_2,l_3) = l_1 \leq_{ll} l_2 \wedge l_1 \leq_{ll} l_3$

(N_4) $app(l_1,l_2) \leq_{ll} l_3 = l_1 \leq_{ll} l_3 \wedge l_2 \leq_{ll} l_3$

(N_5) $m \leq_{nl} app(l_1,l_2) = m \leq_{nl} l_1 \wedge m \leq_{nl} l_2$

(N_6) $app(l_1,l_2) \leq_{ln} m = l_1 \leq_{ln} m \wedge l_2 \leq_{ln} m$.

Particularly important is (N_1) which says that a list l obtained by concatenating two sublists l_1 and l_2 is ordered if and only if l_1, l_2 are ordered and all elements of l_1 are less than or equal to all elements of l_2. Besides these easily provable properties UNICOM makes use of the lemma hierarchy as depicted in figure 4a for verifying (L_1).

The proof of (M_1) is relatively easy basing on two auxiliary lemmas, cf. figure 4b.

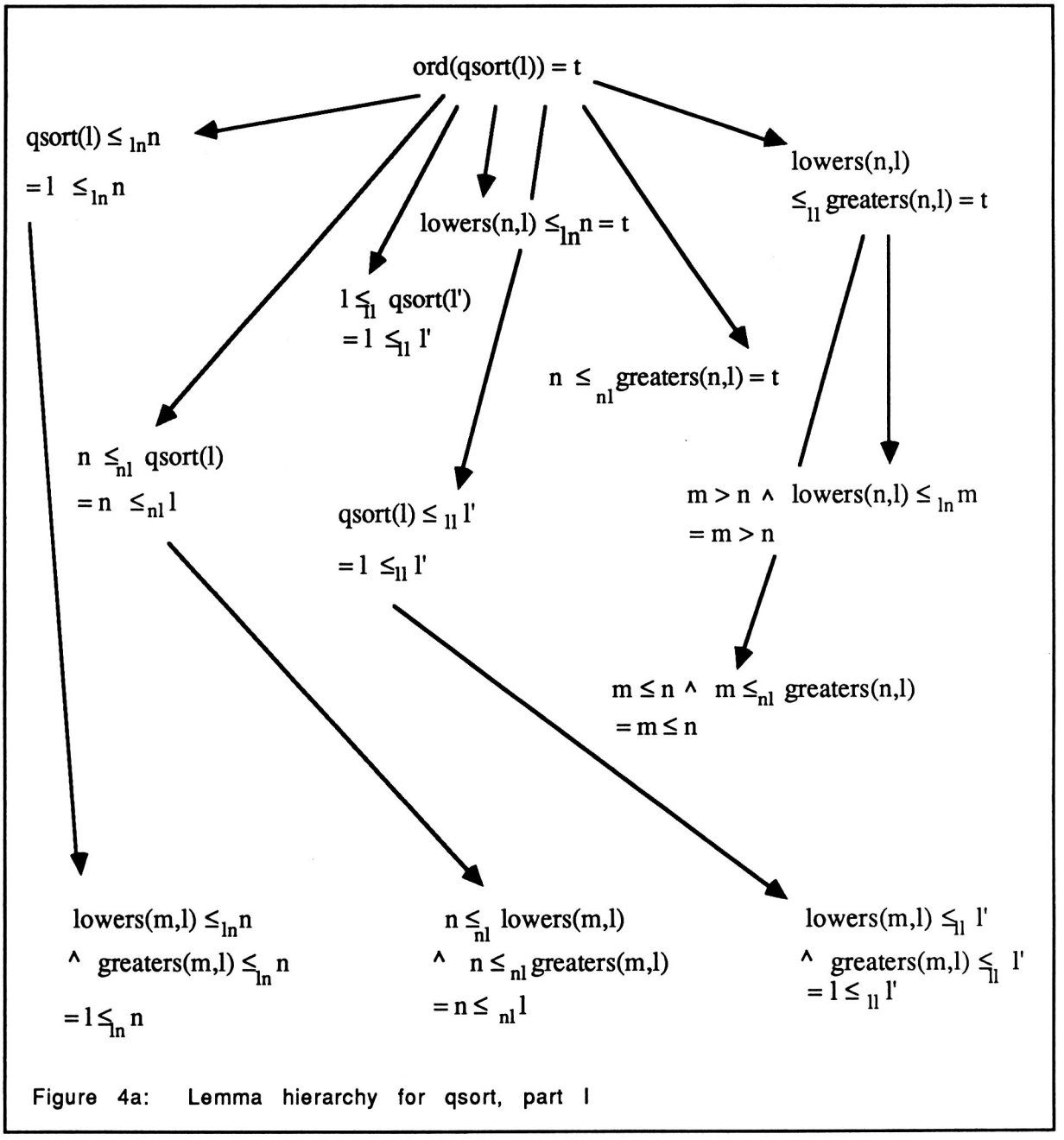


Figure 4a: Lemma hierarchy for qsort, part I

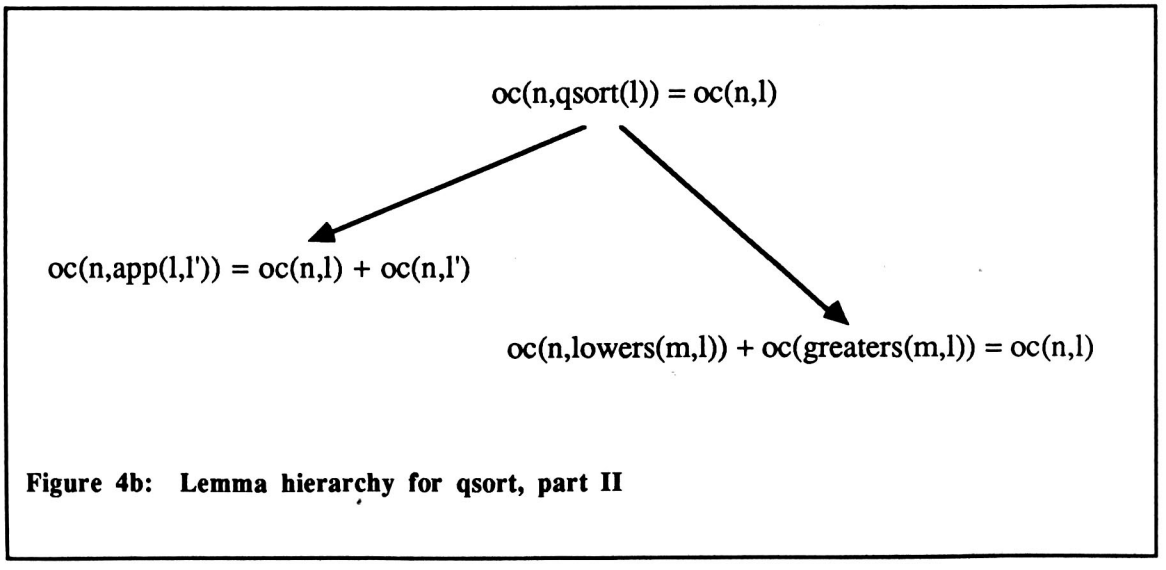


Figure 4b: Lemma hierarchy for qsort, part II

3.3.5 Merge-Sort

For sorting by merging we should partition every list with more than one element into two parts approximately containing the same number of elements. For that purpose we use two auxiliary functions split1, split2: list → list. Split1 collects the elements occurring at odd positions and split2 those which occur at even positions. The correctness properties for the main algorithm msort: list → list are specified as for quick-sort yielding

(1) msort(e) = e
(2) msort(c(n,e)) = c(n,e)
(3) msort(c(m,c(n,l))) = merge(msort(c(m,split1(l))),msort(c(n,split2(l))))
(4) merge(e,l) = l
(5) merge(l,e) = l
(6) merge (c(m,l_1),c(n,l_2)) = if-list(m≤n,c(m,merge(l_1,c(n,l_2))),c(n,merge(c(m,l_1),l_2)))
(7) split1(e) = e
(8) split1(c(m,e)) = c(m,e)
(9) split1(c(m,c(n,l))) = c(m,split1(l))
(10) split2(e) = e
(11) split2(c(m,e)) = e
(12) split2(c(m,c(n,l))) = c(n,split2(l))
(13) ord(e) = t
(14) ord(c(m,l)) = m $\leq_{nl}$ l ∧ ord(l)
(15) m $\leq_{nl}$ e = t
(16) m $\leq_{nl}$ c(n,l) = m ≤ n ∧ m $\leq_{nl}$ l
(17) oc(m,e) = o
(18) oc(m,c(n,l)) = oc(m,l) + if-nat(m $=_{nat}$ n,s(o),o).

The correctness properties to be established are

(L_1) ord(msort(l)) = t and
(L_2) oc(n,msort(l)) = oc(n,l).

The proof of (L_1) is easy provided that the lemma

(L_3) ord(merge(l_1,l_2)) = ord(l_1) ∧ ord(l_2)

is available. (L_3) corresponds closely to the intuition behind sorting by merging, namely that merging two ordered lists again produces an ordered list. (L_3) indeed contains more information because it it also states that whenever a list l which is constructed by merging l_1 and l_2 is not ordered then l_1 or l_2 is not ordered, too. For proving (L_3) we need in particular some kind of transitivity properties relating ≤ and $\leq_{nl}$ the proofs of which are straightforward (cf. figure 5a).

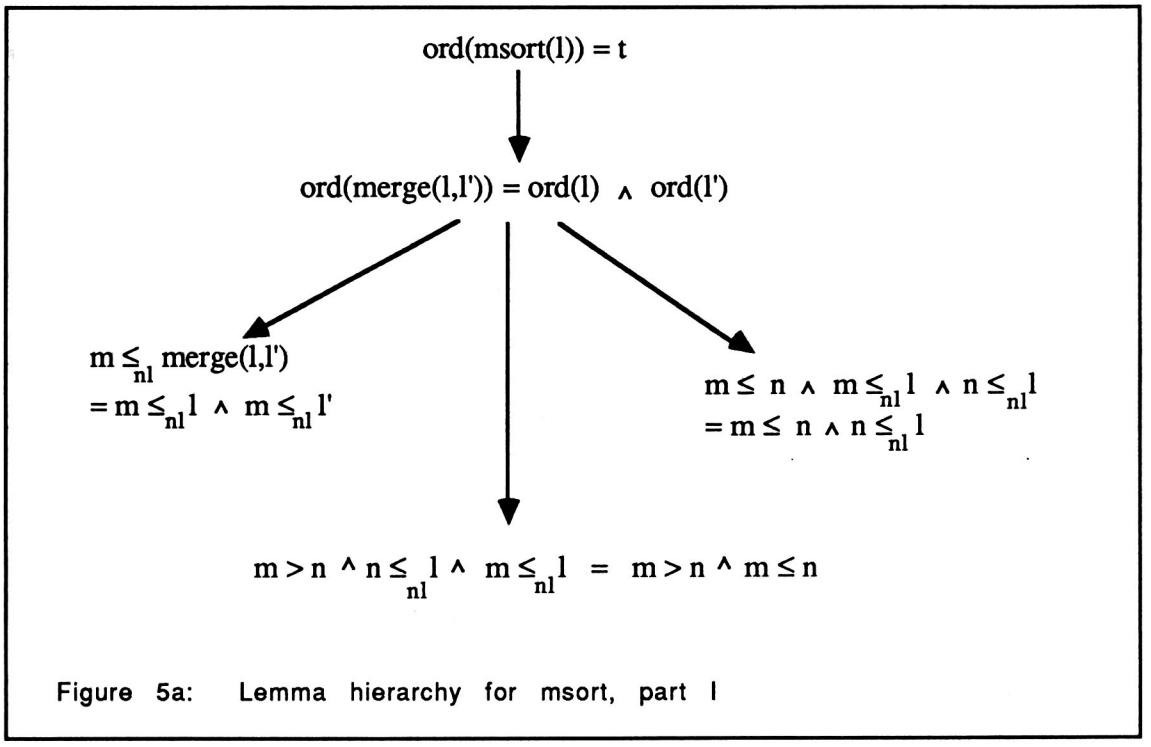


Figure 5a: Lemma hierarchy for msort, part I

The proof of

(L_2) $oc(n,msort(l)) = oc(n,l)$

requires two more lemmas (cf. figure 5b), namely the decomposition property

(L_4) $oc(n,merge(l_1,l_2)) = oc(n,l_1) + oc(n,l_2)$

and the combination property

(L_5) $oc(n,split1(l)) + oc(n,split2(l)) = oc(n,l)$.

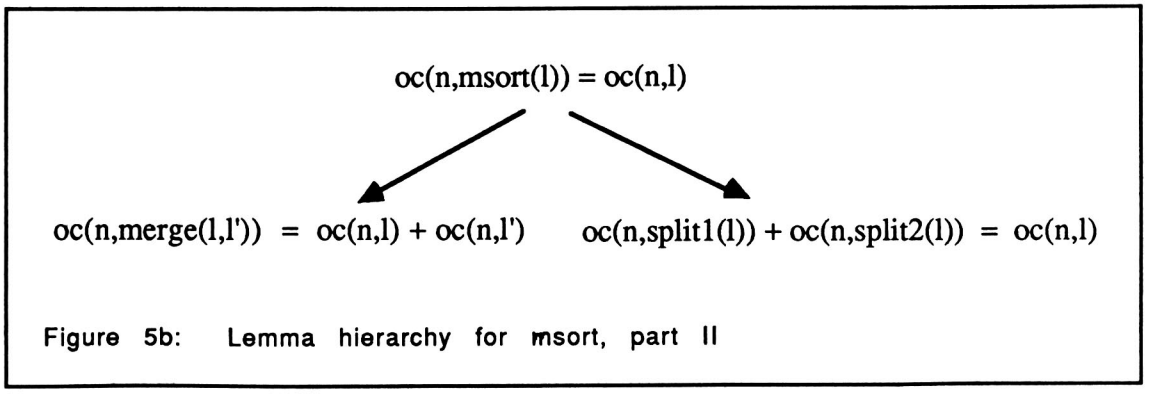


Figure 5b: Lemma hierarchy for msort, part II

4 Conceptual and Technical Aspects of Intelligent Proof Engineering

In the following we shall review and discuss some conceptual and proof technical aspects which have turned out to be crucial for successfully handling non-trivial verification problems as described. In fact we mainly presented in chapter 3 the result of an intelligent proof engineering process for a couple of non-trivial verification problems. Many fruitless efforts and impasses have not been mentioned. The problems that had to be tackled are essentially twofold. On the one hand side the specification of the involved algorithms as well as of their corresponding correctness properties has to be carefully designed. Moreover the resulting proof problems have to be adequately structured and prepared. On the other hand the actual mechanically supported proof process for specific conjectures may be technically quite challenging, even in the case that the available inductive knowledge (auxiliary lemmas) in principle suffices. This problem is due to various proof technical degrees of freedom of the underlying proof method.

4.1 Conceptual Aspects of Intelligent Proof Engineering

A rather simple but important observation is the following. Whenever we want to prove some property of an algorithm whose specification involves supplementary functions we usually have to exploit auxiliary knowledge about these underlying functions. This comes true in an even stronger way when we proceed in a top-down design style. Then the supplementary functions are constructed such that certain intended properties are indeed satisfied. Of course these properties have to be kept in mind and probably made use of when trying to establish some verification condition of the main algorithm. For illustration let us reconsider from this point of view the sorting algorithms dealt with. First we recall the central specification parts:

isort(e) = e
isort(c(n,l)) = ins(n,isort(l))

minsort(e) = e
minsort(c(n,l)) = c(min(n,l),minsort(del(min(n,l),c(n,l))))

bsort(e) = e
bsort(c(n,l)) = c(last(bubble(c(n,l))),bsort(all-but-last(bubble(c(n,l)))))

qsort(e) = e
qsort(c(n,l)) = app(qsort(lowers(n,l),c(n,qsort(greaters(n,l)))))

msort(e) = e
msort(c(n,e)) = c(n,e)
msort(c(m,c(n,l))) = merge(msort(c(m,split1(l))),msort(c(n,split2(l)))).

Consider now for instance the specification of isort with the auxiliary function ins. For proving

$ord(isort(l)) = t$

it is clear that we have to rely on the intended behaviour of ins. Namely, it should insert a given element into an ordered list such that this property remains invariant. In slightly generalized form this is represented by the auxiliary lemma

$ord(ins(n,l)) = ord(l).$

For verifying

$ord(msort(l)) = t$

we have to exploit the fact that the function merge was indeed defined such that merging two ordered lists in turn produces an ordered list which – again in slightly generalized form – is caught by the auxiliary lemma

$ord(merge(l_1,l_2)) = ord(l_1) \wedge ord(l_2).$

The main design decision for qsort was to decompose the sorting process for a non-empty list c(n,l) into a preprocessing step computing the smaller lists lowers(n,l) and greaters(n,l) and to concatenate the results of recursively sorting them where the splitting element is placed in between. The properties closely corresponding to this design decision are

$ord(app(l_1,l_2)) = ord(l_1) \wedge ord(l_2) \wedge l_1 \leq_{ll} l_2$
$lowers(n,l) \leq_{ln} n = t$
$n \leq_{nl} greaters(n,l) = t$ and
$lowers(n,l) \leq_{ll} greaters(n,l) = t$

which play an central role in verifying

$ord(qsort(l)) = t.$

Analaogous considerations for minsort and bsort have led to the introduction of the lemmas

$min(n,l) \leq n = t$ and
$min(n,l) \leq_{nl} l = t.$

For bsort the intention behind making bubble small elements to the right is (partially) reflected by the lemma

$last(bubble(c(n,l))) = min(n,l).$

For successfully verifying the ordered property for the sorting algorithms considered these straightforward supplementary lemmas do not yet suffice. We have to exploit additionally the second correctness property, namely permutative equivalence of a list and its sorted version. Moreover we must use the knowledge that the result of comparisons like $m \leq_{nl} l$, $l \leq_{ln} m$ and $l_1 \leq_{ll} l_2$ does not depend on the order of the elements in l, l_1, l_2. Note that this knowledge is implicitely visible within the definitions of $\leq_{nl}$, $\leq_{ln}$ and $\leq_{ll}$ due to the AC-property of boolean conjunction. The combination of these issues clearly motivates and justifies the introduction of subsidiary lemmas like

$$m \leq_{nl} \text{minsort}(l) = m \leq_{nl} l$$
$$m \leq_{nl} \text{bsort}(l) = m \leq_{nl} l \quad \text{or}$$
$$\text{qsort}(l_1) \leq_{ll} l_2 = l_1 \leq_{ll} l_2 .$$

And finally, it is not surprising that additional knowledge about the properties of $\leq_{nl}$, $\leq_{ln}$ and $\leq_{ll}$ and their interplay is needed for some cases, for instance transitivity properties like

$$m \leq n \wedge m \leq_{nl} l \wedge n \leq_{nl} l = m \leq n \wedge n \leq_{nl} l.$$

The situation concerning the verification of the permutative equivalence property is quite similar. In an analogous style most auxiliary lemmas used can be explained and motivated. In particular, the design decision to model permutative equivalence via a counting function oc for element occurrences is closely related to the nature of the problem to be solved. This connection is due to the presence of the AC-operator + within the definition of oc which corresponds to the fact that the order of elements is irrelevant when counting occurrences.

4.2 Technical Aspects of Intelligent Proof Engineering

Even in the case that a specification and verification problem has been carefully modelled and structured the remaining proof tasks may be quite challenging. In fact, a couple of proof technical aspects and details have turned out to be crucial for practically successful verification with a system like UNICOM. These issues are now discussed.

4.2.1 Improving Simplification by Using AC-Rewriting

When we started this case study in verifying sorting algorithms simplification in UNICOM was implemented essentially by ordinary rewriting. As a consequence the mechanical proofs of even simple inductive lemmas often were quite tedious and complicated, if not impossible, in particular concerning boolean and natural number reasoning. This problem was due to the fact that the AC-properties of operators like $\wedge$, $\vee$ and + could not be mechanically exploited for simplification. The integration of AC-rewriting into UNICOM by means of an earlier developed and implemented efficient algorithm for AC-matching based on constraint propagation techniques (cf. [GD 88]) did

indeed solve this problem. An important practical aspect when incorporating AC-rewriting into the simplication mechanism concerns so-called extension rules which are well-known from completion modulo AC-theories (cf. [PS 81]). Assume for illustration that we want to prove the inductive conjecture

(C) $l = r$

where l is of the form $f(s_1,s_2)$ with f an AC-operator, e.g. the boolean conjunction. Then within the process of computing critical pairs with (C) and simplifying their left and right hand sides we may get an intermediate result of the form

(*) $f(\sigma s_1, f(\sigma s_2, s_3))$

for some substitution σ. Now we would like to apply (C) for further simplifying this term but (C) is neither applicable at top position nor to the subterm $f(\sigma s_2, s_3)$. We first have to transform the term using the AC-axioms for f, for instance into

$$f(f(\sigma s_1, \sigma s_2), s_3)$$

before being able to apply (C). Technically this difficulty can be overcome by using the extended version of (C), namely

(C_{ext}) $f(x,l) = f(x,r)$

where x is a new variable. (C_{ext}) can be directly applied to reduce (*) at top level by means of AC-matching. Thus explicit internal preprocessing transformations using AC-steps for enabling reduction at certain subterms can be avoided by using such extension rules. What does this mean in practice for completion based inductive theorem proving? Essentially we have two possibilities.

Instead of the inductive conjecture (C) we try to prove (C) and (C_{ext}) together using the simple version of AC-rewriting. Indeed, within our domain of interest when starting only with (C_{ext}) the original version (C) is usually generated after one appropiate critical pair computation and simplification step. Consider for instance an extended conjecture

(C_{ext}) $b \wedge (s_1 \wedge s_2) = b \wedge s_3$.

The top position of the left hand side obviously is inductively complete and the corresponding critical pairs with the definition rules for $\wedge$ are

$s_1 \wedge s_2 = t \wedge s_3$ and $f = f \wedge s_3$.

The latter one simplifies to the trivial equation $f = f$ and the first one exactly to (C). The case for boolean disjunction $\vee$ is completely analogous. For the natural number AC-operator + (as well as

for multiplication) we again have the same situation when using for simplication the additional non-equational cancellation law

$$s(m) = s(n) \Rightarrow m = n$$

for the free constructor s (for multiplication the cancellation rule $m+n = m+p \Rightarrow n = p$ would be needed). In all these cases the equations $f(x,y) = f(x,z)$ and $y = z$ are equivalent concerning inductive validity whereas in general only the implication $y = z \Rightarrow f(x,y) = f(x,z)$ is valid.

The second possibility is to integrate the extension rule mechanism automatically into the simplification process. Thus the user only has to provide the original conjecture (C) whereas the system independently uses the internally constructed corresponding extension rule for AC-simplification. This variant is of course more comfortable for the human user because he does not have to provide explicitely extended versions of conjectures. In UNICOM both techniques have been implemented.

It should be mentioned that sometimes extension rules are needed in a modified distributed form. For instance, when we want to prove

$$(*) \quad b' \wedge b \vee b' \wedge \neg b = b'$$

knowing that $(**)$ $b \vee \neg b = t$ holds, (*) is easily obtained from (**) by conjunctively adjoining b' on both sides and subsequent simplification by distributing $\wedge$ over $\vee$ (on the left hand side). That means it may be useful to (automatically) construct (and simplify) extension rules where the top operator of the extension rule is different from the original top operator.

4.2.2 Non-Reducing Proof Simplification Steps

According to the main theorem underlying completion based inductive theorem proving (cf. chapter 2) the essence of the method consists in assuring (ground) simplifiability of critical proofs obtained by overlapping definition rules into conjectures. The easiest way to establish this simplifiability property clearly consists in trying to find rewrite proofs for the corresponding critical pairs by means of ordinary reduction to normal forms that coincide. A significant practical improvement concerning simplification power is obtained by using AC-rewriting which can still be automatically performed. But even this enhanced technique does not solve the problem in all cases. The reason is that it is sometimes necessary to transform an intermediate result by applying a rewrite rule in the inverse direction in order to enable a next reducing simplification step. Let us illustrate this phenomenon via an interesting example mentioned in 3.3.1. For verifying the correctness of minsort we had to use the auxiliary lemmas

(L_1) $\min(n,l) \leq n = t$ and

(L_2) $\min(n,l) \leq_{nl} l = t$

which could not be proved mechanically by UNICOM. In fact we need two more properties for establishing (L_1), (L_2), namely

(L_3) $n \leq m \wedge \min(n,l) \leq m = n \leq m$ and
(L_4) $n < m \wedge \min(n,l) \leq m = n < m$.

The dependency graph is shown in figure 6.

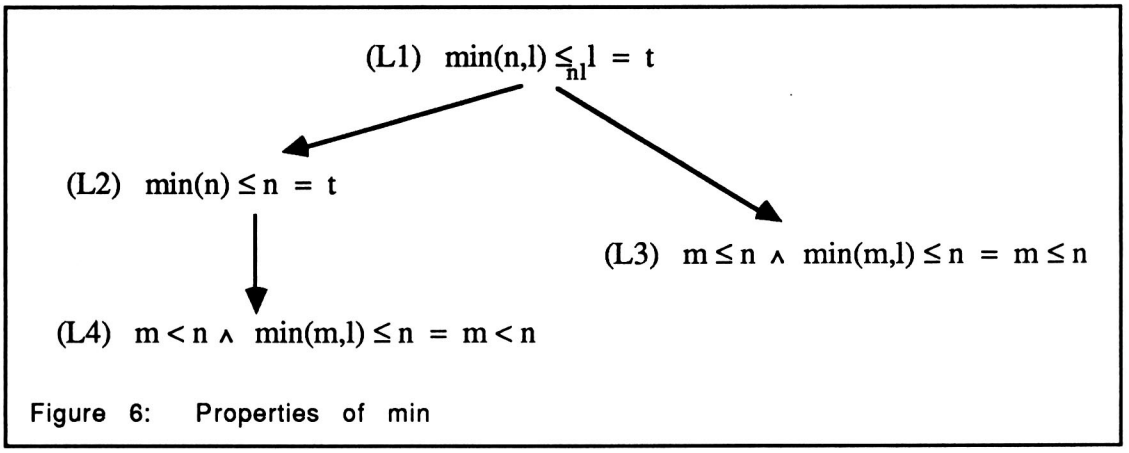


Figure 6: Properties of min

Let us have a closer look on (L_3). Here the interesting case is the critical peak obtained by overlapping the recursive definition rule for min into (L_3). The resulting critical pair is

$n \leq m \wedge \text{if-nat}(n \leq p, \min(n,l), \min(p,l)) \leq m = n \leq m$

with corresponding superposition term $n \leq m \wedge \min(n, c(p,l)) \leq m$. Reducing the left hand side leads to

$n \leq m \wedge \text{if-bool}(n \leq p, \min(n,l) \leq m, \min(p,l) \leq m)$

and

$n \leq m \wedge n \leq p \wedge \min(n,l) \leq m \vee n \leq m \wedge p < n \wedge \min(p,l) \leq m$.

Applying (L_3) using AC-matching with implicit extension rules we get

$n \leq m \wedge n \leq p \vee n \leq m \wedge p < n \wedge \min(p,l) \leq m$.

Now we would like to apply (L_3) to the second component of the disjunction, too. But this is impossible since the subterm $p \leq m$ is missing. Nevertheless $p \leq m$ is implied by the conjuncts $n \leq m$ and $p < n$, which is formalized by the transitivity lemma

(L_5) $p < n \wedge n \leq m \wedge p \leq m = p < n \wedge n \leq m$.

Thus, by applying this transitivity rule from right to left the missing subterm can be introduced yielding

$$(*) \quad n \leq m \wedge n \leq p \;\vee\; n \leq m \wedge p < n \wedge p \leq m \wedge \min(p,l) \leq m .$$

Now, (L_3) is indeed applicable leading to

$$n \leq m \wedge n \leq p \;\vee\; n \leq m \wedge p < n \wedge p \leq m .$$

Elimination of $p \leq m$ by another application of (L_5) in the "right" direction and boolean simplification finally produce the desired result

$$n \leq m$$

which coincides with the right hand side of the critical pair. Hence, we are done, provided that the intermediate "peak result" (*) is smaller than the original superposition term w.r.t .the underlying reduction ordering. This condition is indeed satisfied when choosing an appropriate recursive path ordering. Note that the interesting step of applying (L_5) in reverse direction essentially is a goal-directed non-trivial guessing step which enables further simplification. Currently such a kind of reasoning with non-reducing simplification steps cannot be performed mechanically by UNICOM. In fact the difficulties involved are quite obvious. From a theoretical point of view it has to be verified that the overall complexity of the constructed proof is smaller than that of the critical peak itself. And practically the question arises when and how such non-reducing guessing steps should be performed. To be theoretically precise it must be noted that the problem of verifying proof complexities already occurs when using AC-rewriting. Here we need AC-compatible reduction orderings ensuring theoretical correctness. Such orderings are currently not available in UNICOM. Nevertheless by inspecting the proofs produced it can often be verified by hand that the relevant conditions are indeed satisfied.

4.2.3 Eliminating Simplification Indeterminism

Another practically very important and subtle point concerning completion based inductive theorem proving is the question of how to control the simplification process. Let us again illustrate the problem via an example. Within the proof of the correctness property

$$\text{ord}(\text{bsort}(l)) = t$$

we had to simplify the intermediate result

$$\text{last}(\text{bubble}(c(n,l)) \leq_{nl} \text{all-but-last}(\text{bubble}(c(n,l)))$$

to t. Besides the corresponding definition rules a couple of auxiliary lemmas including

(L_3) $last(l) \leq_{nl} all\text{-}but\text{-}last(l) = l \leq_{nl} all\text{-}but\text{-}last(l)$
(L_4) $m \leq_{nl} bubble(l) = m \leq_{nl} l$
(L_5) $last(bubble(c(n,l)) = min(n,l)$
(L_6) $min(n,l) \leq n = t$
(L_7) $min(n,l) \leq_{nl} l = t$

were available. In fact we chose (L_3) for the next simplification step yielding

$last(bubble(c(n,l))) \leq_{nl} bubble(c(n,l))$

which could finally be reduced to t as desired. Consider now what would have happened if instead of (L_3) we had chosen (L_5) for the next simplification step. Indeed, the result would have been

$min(n,l) \leq_{nl} all\text{-}but\text{-}last(bubble(c(n,l)))$

which is irreducible with respect to the corresponding definition rules and available lemmas. Thus the proof attempt would have failed. Obviously the final result of successive simplifications does not have to be unique. This phenomenon is due to the fact that the rules which are used for simplification do not constitute a confluent system in general. Whereas this property usually holds for the set of definition rules (in most cases it is not only ground confluent but even confluent) it is in general violated if additional inductive lemmas and conjectures are taken into account. Hence it is very important to perform the simplification steps in an intelligent goal-directed way. One general heuristic which has turned out to be very useful in many examples and which is implemented in UNICOM roughly proceeds as follows. The rules available for simplification are partitioned into definition rules, inductive lemmas and the actual conjecture. The highest priority for simplification is assigned to the conjecture itself which corresponds to the intuition that an induction hypothesis should be applied as early and as often as possible. Next it is checked whether a lemma can be used for simplification. If this is not possible, too, then it is attempted to apply a definition rule for one of the function symbols involved. Moreover the available lemmas are ordered decreasingly with respect to their estimated importance. This latter priority mechanism is currently realized implicitely in UNICOM. The priority of a lemma is determined by its position within the actual specification hierarchy which is processed bottom up.

4.2.4 How to Choose Inductively Complete Positions

For a given inductive conjecture UNICOM computes all positions which are inductively complete, i.e. suffice for constructing critical pairs. Depending on a system parameter either all corresponding proof attempts are then automatically developed in parallel or the user is asked to choose one for continuing. Whereas the fully automatic variant proceeding in parallel is theoretically quite elegant it usually causes severe efficiency problems in practice. In many cases certain inductively complete

positions are completely inappropiate for the intended goal whereas other choices seem to be more promising. For instance, when trying to prove the inductive conjecture

$$m \leq n \wedge m \leq_{nl} \text{greaters}(n,l) = m \leq n$$

within the verification process for the quick-sort algorithm it is hopeless to choose the inductively complete position of the left hand side subterm $m \leq n$ for critical pair construction. Instead one should try an induction over the structure of the list l which corresponds to choosing the position of greaters(n,l). Within the framework of classical inductive theorem proving much work has been devoted to recursion analysis for finding appropriate induction terms and induction schemas for a given conjecture (cf. [BM 79], [Bu 88]). We suspect that such a sophisticated analysis can be carried over to the completion based approach and provide useful heuristics for supporting or even automating an intelligent selection process for inductively complete positions. But the details concerning this transfer of results and techniques from classical inductive theorem proving into our context still have to be worked out.

4.2.5 Handling Conditional Reasoning

Within the presented case study in specifying and verifying a couple of sorting algorithms conditional reasoning clearly has played an important role. Our decision to model conditional properties and actions by encoding via ternary if-then-else operations was mainly motivated by pragmatic reasons. First of all there still exist various severe theoretical and practical problems concerning rewriting and completion techniques for properly conditional systems. And secondly, we actually wanted to find out what can be achieved within the purely equational approach when using a powerful implementation incorporating various refinements and optimizations of the basic method. And indeed, by making extensive use of numerous basic schematic properties of if-then-else operations encouraging experimental results have been obtained. Nevertheless we feel that for many problems a properly conditional approach separating the condition from the conclusion part would be more adequate and natural. Roughly speaking the main problem concerning the coding approach lies in its uniform mixture of conditional and unconditional information. For instance, within a "conditional term" of the form $\text{if-s}(b,s_1,s_2)$ the conditional knowledge $b = t$ or $b = f$ cannot be directly made use of when trying to simplify s_1 or s_2, respectively. Further research is needed to develop a theoretically well-founded and practically applicable extension of the underlying completion based approach for properly conditional systems which is specialized to inductive theorem proving. Some progress along this line of research has already been achieved, e.g. in [Ga 87], [KR 87].

Without claiming to be complete let us finally mention some perspectives for future research, in particular from a practical point of view. We think that much work remains to be done concerning a well-considered design of the whole specification and verification environment. Since a fully automatic inductive theorem prover is not a realistic perspective the efforts should rather be directed towards interactive systems with a high degree of mechanical support. This mechanical

support should especially include modern techniques for constructing, maintaining, modifying and structuring complex data and knowledge bases from which relevant information can be easily extracted. Moreover the system design should deal with an appropiate formalism for specifying and incorporating special heuristics and strategies for controlling the proof process which seem to be promising. More generally spoken higher concepts for knowledge based proof planning should be taken into account.

5. Conclusion

We have investigated the problems arising when trying to specify and verify various sorting algortihms within an equational framework. Using the mechanical support of UNICOM, a completion based inductive theorem prover, we have indeed been able to prove partial correctness of these algortihms. It has turned out that for succeeding in such non-trivial verification tasks a substantial amount of intelligent engineering work is needed. On the one hand side this concerns a careful modelling of the verification properties to be established as well as an appropiate structuring and decomposition of the resulting proof problems. The original specification design decisions have to be taken into account in order to provide enough basic knowledge for the main proofs. On the other hand a couple of practically important proof technical and heuristic issues have been pointed out and exemplified.

References

[Au 79] Aubin, R.: Mechanizing Structural Induction, Part I: Formal System, Part II: Strategies, TCS, Vol. 9, 1979

[AGG 87] Avenhaus, J., Göbel, R., Gramlich, B., Madlener, K, Steinbach, J.: TRSPEC: A Term Rewriting Based System for Algebraic Specifications, Proc. of the 1st International Workshop on Conditional Term Rewriting Systems, Orsay, France, 1987, LNCS 308, eds. S. Kaplan, J.-P. Jouannaud, 1988

[Ba 87] Bachmair, L: Proof Methods for Equational Theories, PhD Thesis, Univ. of Illinois, Urbana Champaign, 1987

[Ba 88] Bachmair, L.: Proof by Consistency in Equational Theories, Proc. of LICS, pp. 228-233, 1988

[BM 79] Boyer, R., Moore, J.: A Computational Logic, Academic Press, 1979

[Bu 69] Burstall, R.: Proving prpoerties of programs by structural induction, Computer Journal 12/1, pp. 41-48, 1969

[Fr 86] Fribourg, L.: A strong restriction of the inductive completion procedure, Proc. 13th ICALP, Rennes, France, LNCS 226, pp. 105-116, 1986

[Ga 87] Ganzinger, H.:Ground term confluence in parametric conditional equational specifications, Proc. 4th STACS, LNCS 247, pp. 286-298, 1987

[Go 80] Goguen, J.A.: How to prove algebraic inductive hypotheses without induction, Proc. of 5th CADE, ed. W. Bibel and R. Kowalski, LNCS 87, pp. 356-373, 1980

[Gö 85] Göbel, R.: Completion of Globally Finite Term Rewriting Systems for Inductive Proofs, Proc. of GWAI 85, Springer Verlag, 1985

[Gö 87] Göbel, R.: Ground Confluence, Proc. 2nd RTA, Bordeaux, France, LNCS 256, 1987

[Gö 88] Göbel, R.: A Completion Procedure for Generating Ground Confluent Term Rewriting Systems, Dissertation, FB Informatik, Universität Kaiserslautern, Feb. 1988

[Gr 89] Gramlich, B.: Inductive Theorem Proving Using Refined Unfailing Completion Techniques, SEKI-Report SR-89-14, FB Informatik, Universität Kaiserslautern

[GD 88] Gramlich, B., Denzinger, J.: Efficient AC-Matching Using Constraint Propagation, SEKI-Report SR-88-14, FB Informatik, Universität Kaiserslautern

[Hu 80] Huet, G.: Confluent reductions: abstract properties and applications to term rewriting systems, JCSS 25, pp. 239-266, 1982

[HH 80] Huet, G., Hullot, J.: Proofs by Induction in Equational Theories withConstructors, Proc. 21st FOCS, pp. 96-107, 1980, also in JCSS 25(2), pp. 239-266, 1982

[HO 80] Huet, G., Oppen, D.C.: Equations and rewrite rules: A survey, in Formal Language Theory: Perspectives and Open Problems, pp. 349-405, ed. R. Book, New York, Academic Press, 1980

[JK 86] Jouannaud, J.-P., Kounalis, E.: Automatic Proofs by Induction in Equational Theories Without Constructors, Proc. Symb. Logic in Computer Science, pp. 358-366, Boston, Massachusetts, 1986, also in Information and Computation, vol. 82/1, pp. 1-33, 1989

[JK 86b] Jouannaud, J.-P., Kirchner, H.: Completion of a set of rules modulo a set of equations, SIAM J. Comp., 15/4, pp. 1155-1194, 1986

[KL 80] Kamin, S., Levy, J.-J.: Two generalizations of recursive path orderings, Unpublished Note, Dept. of Comp. Sci., Univ. of Illinois, Urbana, IL, 1980

[KR 87] Kaplan, S., Remy, J.L.: Completion Algorithms for Conditional Rewriting Systems, Colloquium on the Resolution of Equations in Algebraic Structures, Austin, 1987

[Kü 87] Küchlin, W.: Inductive Completion by Ground Proof Transformation, Proc. CREAS, Lakeway, Texas, 1987

[Mu 80] Musser, D.: On proving inductive properties of abstract data types, Proc. 7th ACM Symp. on Principles of Programming Languages, pp. 154-162, Las Vegas, Nevada, USA, 1980

[Pa 84] Paul, E.: Proof by induction in equational theories with relations between constructors, Proc. of 9th CAAP, ed. B. Courcelle, Cambridge Univ. Press, 1984

[Pl 85] Plaisted, D.A.: Semantic confluence tests and completion methods, Information and Control 65, pp. 182-215,1985

[PS 81] Peterson, G.E., Stickel, M.E.: Complete Sets of Reductions for Some Equational Theories, JACM 28/2, pp.233-264, 1981

[Sc 88] Scherer, R.: UNICOM: Ein verfeinerter Rewrite-basierter Beweiser für induktive Theoreme, Diplomarbeit, FB Informatik, Universität Kaiserslautern, 1988

Appendix

In this appendix we provide a complete listing of the specifications dealt with. Let us first explain some notational conventions used subsequently. If not marked by (↔) equations should always be read from left to right, i.e. as rewrite rules. Inductive properties marked by (*) are considered to be accepted by UNICOM without proof. The inductive lemmas are organized top-down in layers corresponding to the structure of the hierarchical specifications as given to UNICOM. That means when verifying some inductive property of a certain layer all lemmas from layers below have already been proved and are available for simplification. Specification components include a name (SPEC), used subspecifications (USE), sort names (SORTS), defined operations with their arity (OPS), constructors (CONS), definition rules (DEF) and inductive conjectures (IND).

****

SPEC boolean-algebra

SORTS bool

OPS t, f : → bool

¬ : bool → bool

∧, ∨ : bool x bool → bool

if-bool : bool x bool x bool → bool

CONS t, f

DEF ¬ f = t

¬ t = f

f ∧ b = f

t ∧ b = b

f ∨ b = b

t ∨ b = t

if-bool(t,b_1,b_2) = b_1

if-bool(f,b_1,b_2) = b_2

IND

$b \vee b = b$ | $b \wedge b = b$

$b \vee \neg b = t$ | $b \wedge \neg b = f$

$\neg (b_1 \wedge b_2) = \neg b_1 \vee \neg b$ | $\neg (b_1 \vee b_2) = \neg b_1 \wedge \neg b_2$

$b_1 \wedge (b_2 \wedge b_3) = (b_1 \wedge b_2) \vee (b_1 \wedge b_2)$ | $b_1 \wedge b_2 \vee \neg b_1 \wedge b_2 = b_2$

$\text{if-bool}(b_1,b_2,b_3) = (b_1 \wedge b_2) \vee (\neg b_1 \wedge b_2)$ | $\text{if-bool}(b_1,b_2,b_2) = b_2$

$b_1 \wedge b_2 = b_2 \wedge b_1 \ (\leftrightarrow)$ | $b_1 \wedge (b_2 \wedge b_3) = (b_1 \wedge b_2) \wedge b_3 \ (\leftrightarrow)$

$b_1 \vee b_2 = b_2 \vee b_1 \ (\leftrightarrow)$ | $b_1 \vee (b_2 \vee b_3) = (b_1 \vee b_2) \vee b_3 \ (\leftrightarrow)$

****

SPEC natural-numbers

USE boolean algebra

SORTS nat

OPS $o : \rightarrow nat$

$s : nat \rightarrow nat$

$+ : nat \times nat \rightarrow nat$

$=_{nat} : nat \times nat \rightarrow bool$

$\leq : nat \times nat \rightarrow bool$

$\text{if-nat} : nat \times nat \rightarrow bool$

CONS o, s

DEF

$o + n = n$ | $\text{if-nat}(f,m,n) = n$

$s(m) + n = s(m + n)$ | $\text{if-nat}(t,m,n) = m$

$o =_{nat} o = t$ | $o \leq n = t$

$s(m) =_{nat} o = f$ | $s(m) \leq o = f$

$o =_{nat} s(n) = f$ | $s(m) \leq s(n) = m \leq n$

$s(m) =_{nat} s(n) = m =_{nat} n$

IND

$m + n = n + m \ (\leftrightarrow)$ | $n =_{nat} n = t$

$(m + n) + p = m + (n + p) \ (\leftrightarrow)$ | $n \leq_{nat} n = t$

$\text{if-nat}(b,n,n) = n$

$m \leq n \wedge m =_{nat} n = m \leq n$ | $\neg (m \leq n) \wedge m =_{nat} n = f$

$m \leq n \vee m =_{nat} n = m \leq n$ | $m \leq n \wedge \neg (n \leq m) = \neg (n \leq m)$

$m \leq n \vee n \leq m = t$ | $\neg (m \leq n) \wedge \neg (n \leq m) = f$

$\neg (m \leq n) \vee \neg (n \leq m) \vee m =_{nat} n = t$

$m \leq n \wedge n \leq p \wedge m \leq p = m \leq n \wedge n \leq p$

$m \leq n \wedge \neg (p \leq n) \wedge m \leq p = m \leq n \wedge \neg (p \leq n)$

$m \leq n \wedge \neg (p \leq n) \wedge \neg (p \leq m) = m \leq n \wedge \neg (p \leq n)$

$\neg (n \leq m) \wedge n \leq p \wedge m \leq p = \neg (n \leq m) \wedge n \leq p$

$\neg (n \leq m) \wedge n \leq p \wedge \neg (p \leq m) = \neg (n \leq m) \wedge n \leq p$

$\neg (n \leq m) \wedge \neg (p \leq n) \wedge m \leq p = \neg (n \leq m) \wedge \neg (p \leq n)$

$\neg (n \leq m) \wedge \neg (p \leq n) \wedge \neg (p \leq m) = \neg (n \leq m) \wedge \neg (p \leq n)$

****

SPEC lists-of-natural-numbers
USE natural-numbers
SORTS list
OPS e : → list
c : nat x list → list
app : list x list → list
if-list : bool x list x list → list
CONS e, c

DEF app(e,l) = l
app(c(n,l_1),l_2) = c(n,app(l_1,l_2))
if-list(f,l_1,l_2) = l_2
if-list(t,l_1,l_2) = l_1

IND if-list(b,l,l) = l
app(l,e) = l
app(app(l_1,l_2),l_3) = app(l_1,app(l_2,l_3))

****

SPEC insertion-sort
USE lists-of-natural-numbers
OPS isort : list x list → list
ins : nat x list → list
ord : list → bool
perm : list x list → bool
∈ : nat x list → bool
del : nat x list → list

DEF isort(e) = e
isort(c(n,l)) = ins(n,isort(l))
ins(n,e) = c(n,e)
ins(n,c(m,l)) = if-list(n ≤ m, c(n,c(m,l)), c(m,ins(n,l)))
ord(e) = t
ord(c(m,e)) = t
ord(c(m,(c(n,l))) = m ≤ n ∧ ord(c(n,l))
perm(e,e) = t
perm(e,c(m,l)) = f
perm(c(m,l_1),l_2) = m ∈ l_2 ∧ perm(l_1,del(m,l_2))
m ∈ e = f
m ∈ c(n,l) = m =$_{nat}$ n ∨ m ∈ l
del(m,e) = e

$del(m,c(n,l)) = \text{if-list}(m =_{nat} n, l, c(n,del(m,l)))$

IND $ord(isort(l)) = t$

$ord(ins(n,l)) = ord(l)$

$ord(c(n,ins(m,l))) = n \leq m \wedge ord(c(n,l))$

$m \leq n \wedge \neg p \leq n \vee m \leq p \wedge p \leq n = m \leq n \wedge m \leq p$

$ord(c(n,\text{if-list}(b,l_1,l_2)) = \text{if-bool}(b,ord(c(n,l_1)),ord(c(n,l)))$

$perm(l,isort(l)) = t$

$del(n,ins(n,l)) = l$

$n \in ins(n,l) = t$

$del(n,\text{if-list}(b,l_1,l_2)) = \text{if-list}(b,del(n,l_1),del(n,l_2))$

$n \in \text{if-list}(b,l_1,l_2) = \text{if-bool}(b,n \in l_1,n \in l_2)$

$\text{if-list}(m \leq n, l_1, \text{if-list}(m =_{nat} n, l_2, l_3)) = \text{if-list}(m \leq n, l_1, l_3)$

****

SPEC minimum-sort

USE lists-of-natural-numbers

OPS $minsort : list \rightarrow list$

$min : nat \times list \rightarrow nat$

$ord : list \rightarrow bool$

$\leq_{nl} : nat \times list \rightarrow bool$

$\leq_{ll} : list \times list \rightarrow bool$

$perm : list \times list \rightarrow bool$

$del : nat \times list \rightarrow list$

$\in : nat \times list \rightarrow bool$

DEF $minsort(e) = e$

$minsort(c(n,l)) = c(min(n,l),minsort(del(min(n,l),c(n,l))))$

$min(n,e) = n$

$min(n,c(m,l)) = \text{if-nat}(n \leq m, min(n,l),min(m,l))$

$ord(e) = t$

$ord(c(n,l)) = n \leq_{nl} l \wedge ord(l)$

$e \leq_{ll} l = t$

$c(n,l_1) \leq_{ll} l_2 = n \leq_{nl} l_2 \wedge l_1 \leq_{ll} l_2$

$n \leq_{nl} e = t$

$n \leq_{nl} c(m,l) = n \leq m \wedge n \leq_{nl} l$

perm(e,e) = t

perm(e,c(n,l)) = f

perm(c(n,l_1), l_2) = n ∈ l_2 ∧ perm(l_1,del(n,l_2))

m ∈ e = f

m ∈ c(n,l) = m =$_{nat}$ n ∨ m ∈ l

del(m,e) = e

del(m,c(n,l)) = if-list(m =$_{nat}$ n,l,c(n,del(m,l)))

IND ord(minsort(l)) = t

---

m ≤$_{nl}$ minsort(l) = m ≤$_{nl}$ l

m ≤$_{nl}$ del(m,l) = m ≤$_{nl}$ l

min(m,l) ≤ m = t (*)

min(m,l) ≤$_{nl}$ l = t (*)

---

m ≤ n ∧ m ≤$_{nl}$ del(n,l) = m ≤ n ∧ m ≤$_{nl}$ l

m ≤ min(n,l) = m ≤ n ∧ m ≤$_{nl}$ l

---

m ≤$_{nl}$ if-list(b,l_1,l_2) = if-bool(b, m ≤$_{nl}$ l_1, m ≤$_{nl}$ l_2)

m =$_{nat}$ n ∧ b ∨ ¬ (m =$_{nat}$ n) ∧ m ≤ n ∧ b = m ≤ n ∧ b

m ≤ if-nat(b,n,p) = if-bool(b, m ≤ n, m ≤ p)

m ≤ n ∧ n ≤ p ∧ b ∨ m ≤ p ∧ ¬ (n ≤ p) ∧ b = m ≤ n ∧ m ≤ p ∧ b

---

perm(minsort(l),l) = t

---

min(n,l) ∈ c(n,l) = t

---

if-nat(b,m,n) ∈ l = if-bool(b, m ∈ l, n ∈ l)

if-nat(b,m,n) =$_{nat}$ p = if-bool(b, m =$_{nat}$ p, n =$_{nat}$ p)

if-bool(b,b_1,b_2) ∨ if-bool(b,b_3,b_4) = if-bool(b, b_1 ∨ b_3, b_2 ∨ b_4)

****

SPEC bubble-sort

USE lists-of-natural-numbers

OPS bsort, bubble, all-but-last : list → list

last : list → nat

ord : list → bool

≤$_{nl}$: nat x list → bool

oc : nat x list → nat

DEF bsort(e) = e

bsort(c(n,l)) = c(last(bubble(c(n,l))), bsort(all-but-last(bubble(c(n,l)))))

bubble(e) = e

bubble(c(n,e)) = c(n,e)

bubble(c(m,c(n,l))) = if-list(m ≤ n, c(n,bubble(c(m,l))), c(m,bubble(c(n,l))))

last(e) = 0

last(c(n,e)) = n

last(c(m,c(n,l))) = last(c(n,l))

all-but-last(e) = e

all-but-last(c(n,e)) = e

all-but-last(c(m,c(n,l))) = c(m,all-but-last(c(m,l)))

ord(e) = t

ord(c(m,l)) = $m \leq_{nl} l \wedge$ ord(l)

$m \leq_{nl} e = t$

$m \leq_{nl} c(n,l) = m \leq n \wedge m \leq_{nl} l$

oc(m,e) = 0

oc(m,c(n,l)) = oc(m,l) + if-nat($m =_{nat} n$,s(o),o)

IND ord(bsort(l)) = t

$m \leq_{nl}$ bsort(l) = $m \leq_{nl} l$

last(l) $\leq_{nl}$ all-but-last(l) = last(l) $\leq_{nl}$ l

last(bubble(c(n,l))) = min(n,l)

min(m,l) ≤ m = t

$\min(m,l) \leq_{nl} l = t$

$m \leq$ last(bubble(c(n,l))) $\wedge$ $m \leq_{nl}$ all-but-last(bubble(c(n,l))) = $m \leq_{nl}$ bubble(c(n,l))

---

last(c(m,bubble(c(n,l)))) = last(bubble(c(n,l)))

$m \leq_{nl}$ bubble(l) = $m \leq_{nl} l$

$m \leq_{nl}$ if-list(b,l_1,l_2) = if-bool(b, $m \leq_{nl} l_1$, $m \leq_{nl} l_2$)

m ≤ if-nat(b,n,p) = if-bool(b,m ≤ n, m ≤ p)

last(if-list(b,l_1,l_2)) = if-nat(b,last(l_1), last(l_2))

all-but-last(if-list(b,l_1,l_2)) = if-list(b,all-but-last(l_1), all-but-last(l_2))

c(m,if-list(b,l_1,l_2))) = if-list(b,c(m,l_1),c(m,l_2))

---

oc(n,bsort(l)) = oc(n,l)

---

oc(n,c(last(bubble(c(m,l))),e)) + oc(n,all-but-last(bubble(c(m,l)))) = oc(n,bubble(c(m,l)))

---

oc(n,bubble(l)) = oc(n,l)

oc(n,c(last(if-list(b,l_1,l_2)),l_3)) + oc(n,all-but-last(if-list(b,l_1,l_2)))

$= \text{if-nat}(b, oc(n,c(last(l_1),l_3)) + oc(n,all\text{-}but\text{-}last(l_1)),$
$oc(n,c(last(l_2),l_3)) + oc(n,all\text{-}but\text{-}last(l_2)))$
$oc(n,\text{if-list}(b,l_1,l_2)) = \text{if-nat}(b,oc(n,l_1),oc(n,l_2))$

****

SPEC quick-sort

USE lists-of-natural-numbers

OPS $qsort : list \to list$
$lowers, greaters : nat \times list \to list$
$ord : list \to bool$
$\leq_{nl} : nat \times list \to bool$
$\leq_{ln} : list \times nat \to bool$
$\leq_{ll} : list \times list \to bool$
$oc : nat \times list \to nat$

DEF $qsort(e) = e$
$qsort(c(n,l)) = app(qsort(lowers(n,l)),c(n,qsort(greaters(c(n,l)))))$
$lowers(n,e) = e$
$lowers(n,c(m,l)) = \text{if-list}(m \leq n,c(m,lowers(n,l)),lowers(n,l))$
$greaters(n,e) = e$
$greaters(n,c(m,l)) = \text{if-list}(m \leq n,greaters(n,l),c(m,greaters(n,l)))$
$ord(e) = t$
$ord(c(m,l)) = m \leq_{nl} l \wedge ord(l)$
$m \leq_{nl} e = t$
$m \leq_{nl} c(n,l) = m \leq n \wedge m \leq_{nl} l$
$e \leq_{ln} m = t$
$c(n,l) \leq_{nl} m = n \leq m \wedge l \leq_{ln} m$
$e \leq_{ll} l = t$
$c(m,l_1) \leq_{ll} l_2 = m \leq_{nl} l_2 \wedge l_1 \leq_{ll} l_2$
$oc(m,e) = o$
$oc(m,c(n,l)) = oc(m,l) + \text{if-nat}(m =_{nat} n,s(o),o)$

IND $ord(qsort(l)) = t$

$qsort(l) \leq_{ln} n = l \leq_{ln} n$
$qsort(l_1) \leq_{ll} l_2 = l_1 \leq_{ll} l_2$
$l_1 \leq_{ll} qsort(l_2) = l_1 \leq_{ll} l_2$
$lowers(n,l) \leq_{ln} n = t$
$n \leq_{nl} greaters(n,l) = t$
$lowers(n,l) \leq_{ll} greaters(n,l) = t$

$n \leq_{nl} qsort(l) = n \leq_{nl} l$

$lowers(m,l) \leq_{ln} n \wedge greaters(m,l) \leq_{ln} n = l \leq_{ln} n$

$n \leq_{nl} lowers(m,l) \wedge n \leq_{nl} greaters(m,l) = n \leq_{nl} l$

$lowers(m,l_1) \leq_{ll} l_2 \wedge greaters(m,l_1) \leq_{ll} l_2 = l_1 \leq_{ll} l_2$

$m \leq n \wedge m \leq_{nl} greaters(n,l) = m \leq n$

$\neg (m \leq n) \wedge lowers(n,l) \leq_{ln} m = \neg (m \leq n)$

$ord(app(l_1,l_2)) = ord(l_1) \wedge ord(l_2) \wedge l_1 \leq_{ll} l_2$

$l_1 \leq_{ll} c(n,l_2) = l_1 \leq_{ln} n \wedge l_1 \leq_{ll} l_2$

$l_1 \leq_{ll} app(l_2,l_3) = l_1 \leq_{ll} l_2 \wedge l_1 \leq_{ll} l_3$

$app(l_1,l_2) \leq_{ll} l_3 = l_1 \leq_{ll} l_3 \wedge l_2 \leq_{ll} l_3$

$m \leq_{nl} app(l_1,l_2) = m \leq_{nl} l_1 \wedge m \leq_{nl} l_2$

$app(l_1,l_2) \leq_{ln} m = l_1 \leq_{ln} m \wedge l_2 \leq_{ln} m.$

$if\text{-}list(b,l_1,l_2) \leq_{ln} m = if\text{-}bool(b,l_1 \leq_{ln} m, l_2 \leq_{ln} m)$

$m \leq_{nl} if\text{-}list(b,l_1,l_2) = if\text{-}bool(b, m \leq_{nl} l_1, m \leq_{nl} l_2)$

$if\text{-}list(b,l_1,l_2) \leq_{ll} l_3 = if\text{-}bool(b,l_1 \leq_{ll} l_3, l_2 \leq_{ll} l_3)$

$if\text{-}bool(b,b_1,b_2) \wedge if\text{-}bool(b,b_3,b_4) = if\text{-}bool(b, b_1 \wedge b_3, b_2 \wedge b_4)$

$oc(n,qsort(l)) = oc(n,l)$

$oc(n,app(l_1,l_2)) = oc(n,l_1) + oc(n,l_2)$

$oc(n,lowers(m,l)) + oc(n,greaters(m,l)) = oc(n,l)$

$oc(n,if\text{-}list(b,l_1,l_2)) = if\text{-}nat(b,oc(n,l_1),oc(n,l_2))$

$if\text{-}nat(b,m,n) + if\text{-}nat(b,p,q) = if\text{-}nat(b,m + p,n + q)$

****

SPEC merge-sort

USE lists-of-natural-numbers

OPS msort , split1, split2 : list → list

merge : list x list → list

ord : list → bool

$\leq_{nl}$: nat x list → bool

oc : nat x list → bool

DEF msort(e) = e

msort(c(n,e)) = c(n,e)

msort(c(m,c(n,l))) = merge(msort(c(m,split1(l))),msort(c(n,split2(l))))

merge(e,l) = l

merge(l,e) = l

$merge(c(m,l_1),c(n,l_2)) = if\text{-}list(m \leq n,c(m,merge(l_1,c(n,l_2))),c(n,merge(c(m,l_1),l_2)))$

$split1(e) = e$

$split1(c(m,e)) = c(m,e)$

$split1(c(m,c(n,l))) = c(m,split1(l))$

$split2(e) = e$

$split2(c(m,e)) = e$

$split2(c(m,c(n,l))) = c(n,split2(l))$

$ord(e) = t$

$ord(c(m,l)) = m \leq_{nl} l \wedge ord(l)$

$m \leq_{nl} e = t$

$m \leq_{nl} c(n,l) = m \leq n \wedge m \leq_{nl} l$

$oc(m,e) = o$

$oc(m,c(n,l)) = oc(m,l) + if\text{-}nat(m =_{nat} n, s(o), o)$

IND $ord(msort(l)) = t$

$ord(merge(l_1,l_2)) = ord(l_1) \wedge ord(l_2)$

$m \leq_{nl} merge(l_1,l_2) = m \leq_{nl} l_1 \wedge m \leq_{nl} l_2$

$m \leq n \wedge m \leq_{nl} l \wedge n \leq_{nl} l = m \leq n \wedge n \leq_{nl} l$

$\neg\, m \leq n \wedge m \leq_{nl} l \wedge n \leq_{nl} l = \neg\, m \leq n \wedge m \leq_{nl} l$

$ord(if\text{-}list(b,l_1,l_2)) = if\text{-}bool(b, ord(l_1), ord(l_2))$

$m \leq_{nl} if\text{-}list(b,l_1,l_2) = if\text{-}bool(b, m \leq_{nl} l_1, m \leq_{nl} l_2)$

$oc(n,msort(l)) = oc(n,l)$

$oc(n,merge(l_1,l_2)) = oc(n,l_1) + oc(n,l_2)$

$oc(n,split1(l)) + oc(n,split2(l)) = oc(n,l)$

$oc(n,if\text{-}list(b,l_1,l_2)) = if\text{-}nat(b,oc(n,l_1),oc(n,l_2))$

****